BLOCKSEC

# Security Audit
# Report for Frost Adaptor Signature

**Date:** August 26, 2025 **Version:** 1.0

**Contact:** contact@blocksec.com

# Contents

## Report Manifest

| Item | Description |
| --- | --- |
| Client | Bitway Labs |
| Target | Frost Adaptor Signature |

## Version History

| Version | Date | Description |
| --- | --- | --- |
| 1.0 | August 26, 2025 | First release |

## Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2025-09-01 02:39:45+0000

# Chapter 1 Introduction

## 1.1 About Audit Target

| Information | Description |
|---|---|
| Type | Software Library |
| Language | Rust |
| Approach | Semi-automatic and manual verification |

The target of this audit is the code repository [1] of Frost Adaptor Signature of Bitway Labs.

Frost Adaptor Signature is a library that implements the Schnorr adaptor signature scheme, compatible with both FROST and Taproot. Specifically, it builds upon ZCash's FROST implementation and customizes two signature methods to suit specific project requirements.

Note this audit only focuses on the files in the following directory:

- frost-adaptor-signature/src/*

Other files are not within the scope of the audit. Additionally, all dependencies of the files within the audit scope are considered reliable in terms of both functionality and security, and are therefore not included in the audit scope.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report.

| Project | Version | Commit Hash |
|---|---|---|
| frost-adaptor-signature | `Version 1` | `ed111b03d7581f7919270e1e16c1d12ddd740adb` |
| | `Version 2` | `b6e4318bafc62049ecef2c9091a5862f88b2965e` |

## 1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the audit target, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of audit targets.

[1] https://github.com/bitwaylabs/frost-adaptor-signature.git

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

## 1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan audit targets with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of audit targets and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

### 1.3.1 Security Issues

* Access control
* Permission management
* Whitelist and blacklist mechanisms
* Initialization consistency
* Improper use of the proxy system
* Reentrancy
* Denial of Service (DoS)
* Untrusted external call and control flow
* Exception handling
* Data handling and flow
* Events operation
* Error-prone randomness
* Oracle security
* Business logic correctness
* Semantic and functional consistency
* Emergency mechanism
* Economic and incentive impact

### 1.3.2 Additional Recommendation

* Gas optimization
* Code quality and style

**Note** *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

## 1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

**Table 1.1:** Vulnerability Severity Classification

| Impact \ Likelihood | *High* | *Low* |
|---|---|---|
| *High* | High | Medium |
| *Low* | Medium | Low |

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following five categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Partially Fixed** The item has been confirmed and partially fixed by the client.
- **Fixed** The item has been confirmed and fixed by the client.

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

# Chapter 2 Findings

In total, we found **four** potential security issues. Besides, we have **five** recommendations and **two** notes.

- High Risk: 1
- Medium Risk: 2
- Low Risk: 1
- Recommendation: 5
- Note: 2

| ID | Severity | Description | Category | Status |
|---|---|---|---|---|
| 1 | High | Incomplete validation when the feature `cheater-detection` is disabled | Software Security | Fixed |
| 2 | Medium | Lack of a threshold validation in the module `refresh` | Software Security | Confirmed |
| 3 | Medium | Lack of consistency validation between the inputs `group_commitment` and `signer_nonces` | Software Security | Confirmed |
| 4 | Low | Lack of an aggregated signature verification | Software Security | Fixed |
| 5 | - | Implement a threshold validation in the function `sign_with_dkg_nonce()` | Recommendation | Confirmed |
| 6 | - | Avoid unsafe error handling with `unwrap()` | Recommendation | Fixed |
| 7 | - | Remove redundant operations in `sign_with_group_commitment()` | Recommendation | Fixed |
| 8 | - | Remove the redundant code | Recommendation | Fixed |
| 9 | - | Revise the improper error message | Recommendation | Fixed |
| 10 | - | Modifications to the `FROST` protocol are assumed to be safe | Note | - |
| 11 | - | Inconsistency between the key refresh documentation and the implementation | Note | - |

The details are provided in the following sections.

## 2.1 Security Issue

### 2.1.1 Incomplete validation when the feature `cheater-detection` is disabled

**Severity** High

**Status** Fixed in `Version 2`

**Introduced by** `Version 1`

**Description** The function `aggregate_with_group_commitment()` contains inconsistent identifier validation logic that depends on the `cheater-detection` feature flag. When the feature is enabled, the validation checks that identifiers exist in both `signature_shares` and `pubkeys.veri-`

fying_shares(). However, when the feature is disabled, it only validates against signature_shares, potentially allowing invalid identifiers to pass validation.

```
267pub fn aggregate_with_group_commitment(
268    signing_package: &SigningPackage,
269    signature_shares: &BTreeMap<Identifier, round2::SignatureShare>,
270    pubkeys: &keys::PublicKeyPackage,
271    group_commitment: &VerifyingKey,
272) -> Result<Signature, Error> {
273    // Check if signing_package.signing_commitments and signature_shares have
274    // the same set of identifiers, and if they are all in pubkeys.verifying_shares.
275    if signing_package.signing_commitments().len() != signature_shares.len() {
276        return Err(Error::UnknownIdentifier);
277    }
278
279    if !signing_package.signing_commitments().keys().all(|id| {
280        #[cfg(feature = "cheater-detection")]
281        return signature_shares.contains_key(id) && pubkeys.verifying_shares().contains_key(id);
282        #[cfg(not(feature = "cheater-detection"))]
283        return signature_shares.contains_key(id);
284    }) {
285        return Err(Error::UnknownIdentifier);
286    }
```

**Listing 2.1:** frost-adaptor-signature/src/lib.rs

**Impact** This inconsistency may allow invalid identifiers to circumvent the validation.

**Suggestion** Implement complete identifier validation regardless of feature flags to ensure consistent security behavior.

### 2.1.2 Lack of a threshold validation in the module refresh

**Severity** Medium

**Status** Confirmed

**Introduced by** Version 1

**Description** The module refresh enables signers to refresh their secret shares and maintain the same group public key using ZCash's FROST implementation. In the implementation, the threshold (i.e., min_signers) is not allowed to be decreased since decreasing the threshold alters the real secret while keeping the group public key unchanged. However, the usage is problematic since it lacks threshold validation during refresh operations. Specifically, the function lacks a validation for the threshold. When the threshold decreases, the refresh operation succeeds, while the real secret does not correspond to the group's public key. As a result, this leads to failed signature verification even when the number of signers reaches the new threshold.

```
24      pub fn refresh_dkg_part1(identifier: Identifier, max_signers: u16, min_signers: u16) ->
            Result<(round1::SecretPackage, round1::Package), frost_core::Error<frost_secp256k1_tr
            ::Secp256K1Sha256TR>> {
25          let rng = rand::rngs::OsRng;
```

```
26          frost_core::keys::refresh::refresh_dkg_part_1(identifier, max_signers, min_signers, rng
                )
27      }
```

**Listing 2.2:** frost-adaptor-signature/src/lib.rs

```
361   let public_key_package = PublicKeyPackage {
362       header: old_pub_key_package.header,
363       verifying_shares: new_verifying_shares,
364       verifying_key: old_pub_key_package.verifying_key,
365   };
366
367   let key_package = KeyPackage {
368       header: Header::default(),
369       identifier: round2_secret_package.identifier,
370       signing_share,
371       verifying_share,
372       verifying_key: public_key_package.verifying_key,
373       min_signers: round2_secret_package.min_signers,
374   };
375
376   Ok((key_package, public_key_package))
```

**Listing 2.3:** frost/frost-core/src/keys/refresh.rs

**Impact** This results in signature verification failing when the number of signers equals the threshold.

**Suggestion** Implement validation to ensure the new threshold is greater than or equal to the current threshold during refresh operations.

**Feedback from the project** We have verified the same `min_signers` in applications.

### 2.1.3 Lack of consistency validation between the inputs `group_commitment` and `signer_nonces`

**Severity** Medium

**Status** Confirmed

**Introduced by** `Version 1`

**Description** The function `sign_with_group_commitment()` computes the signature share based on the formula: $\lambda_i \cdot hiding + G.ScalarMult(PK_i, challenge \cdot \lambda_i)$. This approach differs significantly from the standard FROST protocol, which uses the formula: $hiding_i + binding_i \cdot binding\_factor_i + G.ScalarMult(PK_i, challenge * \lambda_i)$. While both functions use the `group_commitment` to compute the `challenge`, standard FROST derives the `group_commitment` deterministically using the formula: $\sum_i (hiding_i \cdot G + binding\_factor_i \cdot binding_i \cdot G)$. This derivation indicates that the group commitment has a one-to-one mathematical correspondence with the input `signer_nonces` (i.e., `hiding` and `binding` components). This correspondence is fundamental for ensuring signature verifiability.

However, the function `sign_with_group_commitment()` accepts an externally specified value `group_commitment` without validating its correspondence with `signer_nonces`. This implementation is extremely sensitive to parameter accuracy, where any slight deviation in input parameters could lead to signature verification failures.

```
120   pub(crate) fn sign_with_group_commitment(
121       signing_package: &SigningPackage,
122       signer_nonces: &round1::SigningNonces,
123       key_package: &keys::KeyPackage,
124       group_commitment: &VerifyingKey,
125       // binding_factor: BindingFactor<Secp256K1Sha256TR>,
126       nonces_with_lambda: bool
127   ) -> Result<SignatureShare, Error> {
128       // Compute Lagrange coefficient.
129       let lambda_i = derive_interpolating_value(key_package.identifier(), signing_package)?;
130
131       // Multiply nonces by lambda if nonces_with_lambda is true
132       let signer_nonces = if nonces_with_lambda {
133           let hiding = round1::Nonce::from_scalar(lambda_i * signer_nonces.hiding().to_scalar());
134           let binding = round1::Nonce::from_scalar(lambda_i * signer_nonces.binding().to_scalar()
                  );
135
136           round1::SigningNonces::from_nonces(hiding, binding)
137       } else {
138           signer_nonces.clone()
139       };
140
141       let (signing_package, _, key_package) =
142           Secp256K1Sha256TR::pre_sign(signing_package, &signer_nonces, key_package)?;
143
144       // Compute the per-message challenge.
145       let challenge = <Secp256K1Sha256TR as Ciphersuite>::challenge(
146           &group_commitment.to_element(),
147           key_package.verifying_key(),
148           signing_package.message(),
149       )?;
150
151       // Compute the signature share.
152       // let signature_share = Secp256K1Sha256TR::compute_signature_share(
153       //     &GroupCommitment::<Secp256K1Sha256TR>::from_element(group_commitment.to_element()),
154       //     &signer_nonces,
155       //     binding_factor,
156       //     lambda_i,
157       //     &key_package,
158       //     challenge,
159       // );
160       let signer_nonces = if group_commitment.has_even_y() {
161           signer_nonces.clone()
162       } else {
163           negate_nonces(&signer_nonces)
164       };
165
166       let z_share = Secp256K1ScalarField::deserialize(signer_nonces.hiding().serialize()[..].
```

```
            try_into().unwrap()).unwrap()
167       + (lambda_i * key_package.signing_share().to_scalar() * challenge.to_scalar());
168
169       round2::SignatureShare::deserialize(&Secp256K1ScalarField::serialize(&z_share)[..])
170
171       // Ok(signature_share)
172   }
173}
```

**Listing 2.4:** frost-adaptor-signature/src/lib.rs

**Impact** Inconsistency between `group_commitment` and `signer_nonces` could lead to signature verification failures.

**Suggestion** Implement a validation to ensure the inputs `group_commitment` and `signer_nonces` are properly aligned.

**Feedback from the project** It is designed to use group commitments specifically for decentralized DLC Oracle signers.

### 2.1.4 Lack of an aggregated signature verification

**Severity** Low

**Status** Fixed in `Version 2`

**Introduced by** `Version 1`

**Description** The function `aggregate_with_group_commitment()` aggregates signature shares into a final signature. However, the implementation is problematic due to the lack of aggregated signature verification. Specifically, according to the `FROST` standard requirement, this aggregated signature should be verified using the group public key before publishing or releasing the signature. At the same time, this function does not perform such verification. This could allow invalid signatures to propagate through the network.

**Impact** The function may return invalid signatures that could be propagated throughout the network.

**Suggestion** Implement signature verification using the group public key before returning the aggregated signature to ensure its validity.

## 2.2 Recommendation

### 2.2.1 Implement a threshold validation in the function `sign_with_dkg_nonce()`

**Status** Confirmed

**Introduced by** `Version 1`

**Description** The function `sign_with_group_commitment()` does not validate whether the number of signing commitments meets the minimum threshold requirement. When the participant count falls below the required threshold, this may lead to invalid signature generation, resulting in unnecessary computational resource consumption. It is recommended that proper threshold validation be implemented at the function's entry point.

```
120   pub(crate) fn sign_with_group_commitment(
121       signing_package: &SigningPackage,
122       signer_nonces: &round1::SigningNonces,
123       key_package: &keys::KeyPackage,
124       group_commitment: &VerifyingKey,
125       // binding_factor: BindingFactor<Secp256K1Sha256TR>,
126       nonces_with_lambda: bool
127   ) -> Result<SignatureShare, Error> {
```

**Listing 2.5:** frost-adaptor-signature/src/lib.rs

**Suggestion** Implement a check at the beginning of the function to ensure `signing_package.signing_commitments().len() >= threshold` before proceeding with sign.

### 2.2.2 Avoid unsafe error handling with `unwrap()`

**Status** Fixed in `Version 2`

**Introduced by** `Version 1`

**Description** The codebase contains extensive use of the function `unwrap()`, particularly within critical cryptographic operations (e.g., the function `sign_with_group_commitment()`).

```
166       let z_share = Secp256K1ScalarField::deserialize(signer_nonces.hiding().serialize()[..].
              try_into().unwrap()).unwrap()
167       + (lambda_i * key_package.signing_share().to_scalar() * challenge.to_scalar());
```

**Listing 2.6:** frost-adaptor-signature/src/lib.rs

The function `unwrap()` will cause the program to panic when called on a `Result` type that is `Err` or an `Option` type that is `None`. The prevalent use of `unwrap()` in a cryptographic library's core functionalities poses a severe risk to application stability and reliability.

**Suggestion** Replace the `unwrap()` with explicit error handling mechanisms like the `?` operator or match expressions.

### 2.2.3 Remove redundant operations in `sign_with_group_commitment()`

**Status** Fixed in `Version 2`

**Introduced by** `Version 1`

**Description** The function `sign_with_group_commitment()` currently performs an inefficient and logically redundant serialization and deserialization of a scalar value. Specifically, the scalar derived from `signer_nonces.hiding()` is first serialized into a byte array, only to be immediately deserialized back into a `Secp256K1ScalarField` for a subsequent arithmetic operation, as shown in the snippet below:

```
166       let z_share = Secp256K1ScalarField::deserialize(signer_nonces.hiding().serialize()[..].
              try_into().unwrap()).unwrap()
167       + (lambda_i * key_package.signing_share().to_scalar() * challenge.to_scalar());
```

**Listing 2.7:** frost-adaptor-signature/src/lib.rs

In cryptographic libraries, functions (e.g., `sign_with_group_commitment()`) should ideally receive inputs that are already in their correct, deserialized form. Deserialization, along with any error handling, belongs before the function call, not within it. Performing these conversions inside the signing function adds unnecessary overhead and complexity. This redundancy not only impacts performance but also hinders code readability, making the operation less direct. Cryptographic libraries typically operate directly on primitives, as exemplified by Zcash FROST's `sign()` function, which avoids such intermediate serialization.

**Suggestion** Refactor the function `sign_with_group_commitment()` to directly access the underlying scalar value of `signer_nonces.hiding()` for arithmetic operations.

### 2.2.4 Remove the redundant code

**Status** Fixed in `Version 2`

**Introduced by** `Version 1`

**Description** There are extensive blocks of commented-out or "dead" code throughout the codebase. While useful for temporary debugging, leaving such inactive code in the main branch creates clutter.

```
175// /// Compute a signature share, negating if required by BIP340.
176// fn compute_signature_share(
177//     signer_nonces: &round1::SigningNonces,
178//     group_commitment: <Secp256K1Group as Group>::Element,
179//     lambda_i: <<Secp256K1Group as Group>::Field as Field>::Scalar,
180//     key_package: &frost::keys::KeyPackage<S>,
181//     challenge: Challenge<S>,
182//     sig_params: &SigningParameters,
183// ) -> round2::SignatureShare {
184//     let mut sn = signer_nonces.clone();
185//     if group_commitment.to_affine().y_is_odd().into() {
186//         sn.negate_nonces();
187//     }
188
189//     let mut kp = key_package.clone();
190//     let public_key = key_package.verifying_key();
191//     let pubkey_is_odd: bool = public_key.y_is_odd();
192//     let tweaked_pubkey_is_odd: bool =
193//         tweaked_public_key(public_key, sig_params.tapscript_merkle_root.as_ref())
194//             .to_affine()
195//             .y_is_odd()
196//             .into();
197//     if pubkey_is_odd != tweaked_pubkey_is_odd {
198//         kp.negate_signing_share();
199//     }
200
201//     let z_share = lambda_i * Secp256K1ScalarField::deserialize(&sn.hiding().serialize()).unwrap
      ()
202//     + (lambda_i * kp.signing_share().to_scalar() * challenge.to_scalar());
203
204//     round2::SignatureShare::deserialize(Secp256K1ScalarField::serialize(&z_share)).unwrap()
205// }
```

**Listing 2.8:** frost-adaptor-signature/src/lib.rs

**Suggestion** Remove all identified dead or commented-out code from the codebase.

### 2.2.5 Revise the improper error message

**Status** Fixed in `Version 2`

**Introduced by** `Version 1`

**Description** The `aggregate_with_adaptor_point()` and `aggregate_with_group_commitment()` functions return `Error::UnknownIdentifier` when the lengths don't match. However, this error message is misleading as it suggests an identifier-related issue when the actual problem is a mismatch between the number of signing commitments and signature shares.

```
216    if signing_package.signing_commitments().len() != signature_shares.len() {
217        return Err(Error::UnknownIdentifier);
218    }
```

**Listing 2.9:** frost-adaptor-signature/src/lib.rs

```
275    if signing_package.signing_commitments().len() != signature_shares.len() {
276        return Err(Error::UnknownIdentifier);
277    }
```

**Listing 2.10:** frost-adaptor-signature/src/lib.rs

**Suggestion** Use a specific error type.

## 2.3 Note

### 2.3.1 Modifications to the `FROST` protocol are assumed to be safe

**Introduced by** `Version 1`

**Description** The project uses ZCash's `FROST` implementation and introduces two signing modes, namely `sign_with_adaptor_point()` and `sign_with_dkg_nonce()`. These modes modify the signature schema of the underlying FROST protocol. This audit verifies that the project functions correctly with the modifications, but does not provide mathematical proof of security. This audit assumes that both the ZCash `FROST` dependency and the custom modifications are cryptographically secure and do not introduce security vulnerabilities at the project level.

### 2.3.2 Inconsistency between the key refresh documentation and the implementation

**Introduced by** `Version 1`

**Description** According to the project documentation[1], the description for the key refresh module, "Add or remove signers as needed, without regenerating vaults or migrating funds", conflicts with the legitimate application of ZCash's `FROST`. Specifically, the implementation of

key refresh is designed solely to refresh existing shares and cannot introduce new participant identifiers (as this would fundamentally alter the shared secret). While the documentation's claim directly contradicts this technical limitation. Adding new participants with new identifiers unequivocally requires a new Distributed Key Generation (DKG) ceremony.

[1] https://docs.bitway.com/tct/major-features/feature-key-refresh#f0f5


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS