Audit of FROST

Chainflip.io

04 May 2022
Version: 2.0

Presented by:
Kudelski Security Research Team
Kudelski Security – Nagravision SA

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

DOCUMENT PROPERTIES

Version:	2.0
File Name:	Audit_FROST
Publication Date:	04 May 2022
Confidentiality Level:	For public release
Document Owner:	Tommaso Gagliardoni
Document Recipient:	Tom Nash
Document Status:	Approved

TABLE OF CONTENTS

TABLE OF FIGURES

EXECUTIVE SUMMARY

Kudelski Security ("Kudelski", "we"), the cybersecurity division of the Kudelski Group, was engaged by Chainflip ("the Client") to conduct an external security assessment in the form of a code audit of the FROST implementation developed by the Client.

The assessment was conducted remotely by the Kudelski Security Research Team. The audit took place in February 2022 and focused on the following objectives:

- To provide a professional opinion on the maturity, adequacy, and efficiency of the software solution in exam.
- To check compliance with existing standards.
- To identify potential security or interoperability issues and include improvement recommendations based on the result of our analysis.

This report summarizes the analysis performed and findings. It also contains detailed descriptions of the discovered vulnerabilities and recommendations for remediation.

1.1 Engagement Scope

The scope of the audit was the repository located at:

https://github.com/chainflip-io/chainflip-backend

branch **develop in multisig directory** on commit:
`c0a1c3dc42dd3fb284bc0cb157e13b6a32874097.`

1.2 Engagement Analysis

The engagement consisted of a ramp-up phase where the necessary documentation about the technological standards and design of the solution in exam was acquired, followed by a manual inspection of the code provided by the Client and the drafting of this report.

As a result of our work, we identified **5 Medium**, **2 Low**, and **7 Informational** findings.

Most of these findings and observation are related to missing validations, error mismanagement and poor coding practice, but there is also an issue in the distributed key generation algorithm that could lead to a DoS attack. Notice that the same key generation engine is also used in many other MPC protocols and products.

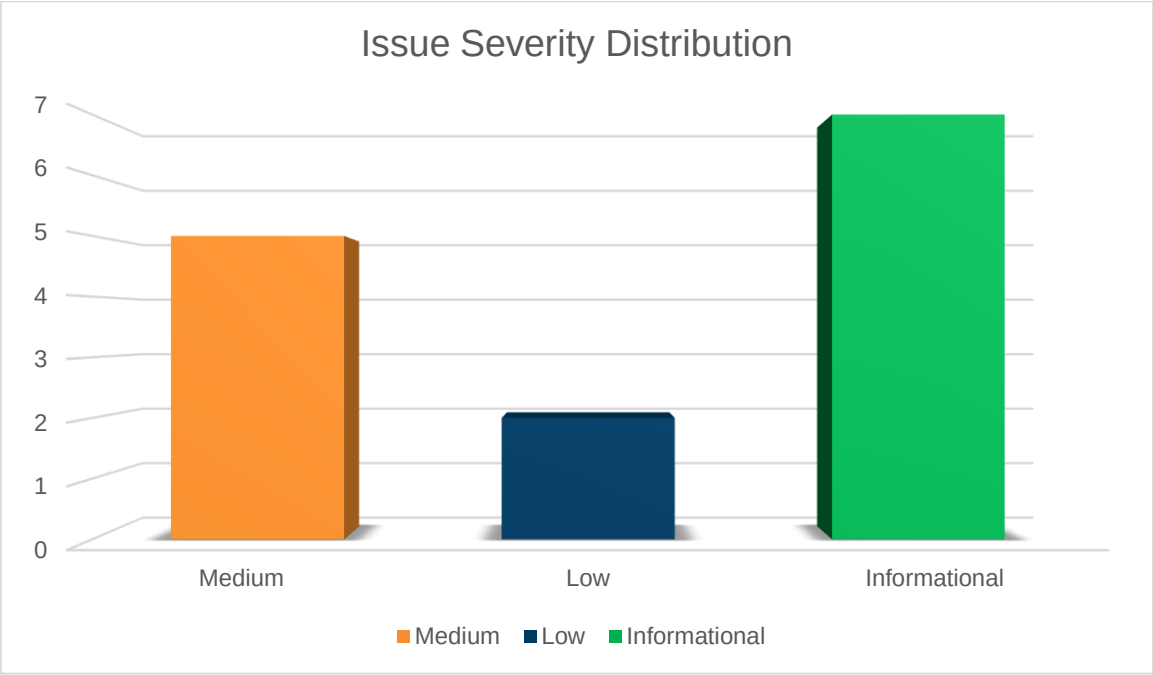

Figure 1 Issue Severity Distribution

1.3 Observations

The library we audited is the Client's implementation of FROST for threshold Schnorr signatures. The FROST cryptographic scheme is recent and challenging to implement correctly, so the fact that we identified issues in the current, version of the code should not be seen as a surprise.

In general, we found that the implementation maintains a high standard and we believe that all the identified vulnerabilities can be easily addressed. Moreover, we did not find evidence of any hidden backdoor or malicious intent in the code.

1.4 Issue Summary List

The following security issues were found:

ID	SEVERITY	FINDING	STATUS
KS-CFR-F-01	Medium	Possible DoS Attack in FROST KeyGen	Remediated
KS-CFR-F-02	Medium	Unsuccessful Zeroization of Random Values	Remediated
KS-CFR-F-03	Medium	Secret Key Shares Stored in Cleartext in Database	Acknowledged
KS-CFR-F-04	Medium	Various Vulnerabilities Found by `cargo-audit`	Acknowledged
KS-CFR-F-05	Medium	Use of Outdated, Unmaintained, or Deprecated Crates	Acknowledged
KS-CFR-F-06	Low	64-bit Values Clipped to 32-bit Values	Remediated
KS-CFR-F-07	Low	Hash Values Not Mapped to Correct Group	Acknowledged

The following are non-security observations related to general design and optimization:

ID	SEVERITY	FINDING	STATUS
KS-CFR-O-01	Informational	Parameters Control Left to Blockchain Governance Layer	Informational
KS-CFR-O-02	Informational	On the Constraint of Public Key x-Coordinate	Informational
KS-CFR-O-03	Informational	Whitepaper is Outdated	Informational
KS-CFR-O-04	Informational	On Error Management	Informational
KS-CFR-O-05	Informational	Compilation Error in `cargo test`	Informational
KS-CFR-O-06	Informational	Test `can_subscribe_infura` Fails	Informational
KS-CFR-O-07	Informational	Warnings from `cargo clippy`	Informational

2. METHODOLOGY

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

Figure 2 Methodology Flow

2.1 Kickoff

The project was kicked off when all of the sales activities had been concluded. We 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. It was an opportunity for both sides to ask questions and get to know each other. By the end of the kickoff there was an understanding of the following:

- Designated points of contact
- Communication methods and frequency
- Shared documentation
- Code and/or any other artifacts necessary for project success
- Follow-up meeting schedule, such as a technical walkthrough
- Understanding of timeline and duration

2.2 Ramp-up

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

- Reviewing previous work in the area including academic papers
- Reviewing programming language constructs for the languages used in the code
- Researching common flaws and recent technological advancements

2.3 Review

The review phase is where a majority of the work on the engagement was performed. In this phase we analyzed the project for flaws and issues that could impact the security posture. This included an analysis of the architecture, a review of the code, and a specification matching to match the architecture to the implemented code.

In this code audit, we performed the following tasks:

1. Security analysis and architecture review of the original protocol
2. Review of the code written for the project

3. Assessment of the cryptographic primitives used
4. Compliance of the code with the provided technical documentation

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 were used to assist the reviewer during the testing. We discuss our methodology in more detail in the following subsections.

Code Safety

We analyzed the provided code, checking for issues related to the following categories:

- General code safety and susceptibility to known issues
- Poor coding practices and unsafe behavior
- Leakage of secrets or other sensitive data through memory mismanagement
- Susceptibility to misuse and system errors
- Error management and logging

This is a general and not comprehensive list, meant only to give an understanding of the issues we have been looking for.

Cryptography

We analyzed the cryptographic primitives and components as well as their implementation. We checked 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

Technical Specification Matching

We analyzed the provided documentation and checked that the code matches the specification. We checked for things such as:

- Proper implementation of the documented protocol phases
- Proper error handling
- Adherence to the protocol logical description

2.4 Reporting

Kudelski delivered to the Client a preliminary report in PDF format that contained an executive summary, technical details, and observations about the project, which is also the general structure of the current final report.

The executive summary contains an overview of the engagement, including the number of findings as well as a statement about our general risk assessment of the project as a whole.

In the report we not only point out security issues identified but also informational findings for improvement categorized into several buckets:

- High
- Medium
- Low
- Informational

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

As we performed the audit, we also identified issues that are not security related, but are general best practices and steps, that can be taken to lower the attack surface of the project.

As an optional step, we can agree on the creation of a public report that can be shared and distributed with a larger audience.

2.5 Verify

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

The output of this phase was the current, final report with any mitigated findings noted.

2.6 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.

Correct memory management is left to Rust and was therefore not in scope. Zeroization of secret values from memory was considered in scope and we tried to identify areas of concern in this sense.

While assessment 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 **Appendix C** of this document.

3. TECHNICAL DETAILS OF SECURITY FINDINGS

This section contains the technical details of our findings as well as recommendations for mitigation.

3.1 Possible DoS Attack in FROST KeyGen

Finding ID: KS-CFR-F-01

Severity: Medium

Status: Remediated

Location: `src/multisig/client/keygen/keygen_stages.rs`

Description and Impact Summary

The 2-round FROST KeyGen procedure is a modification of the Pedersen DKG scheme with the addition of a zero-knowledge proof (ZKP) on the low-degree coefficient of a user's local polynomial in order to protect against rogue-key attacks against that coefficient, which would allow an adversary to influence the low-degree coefficient of the resulting final polynomial, i.e., the long-term secret of the scheme. The procedure is explained in Fig. 1 of the FROST paper:

FROST KeyGen

Round 1

1. Every participant P_i samples t random values $(a_{i0}, \ldots, a_{i(t-1)}) \xleftarrow{\$} \mathbb{Z}_q$, and uses these values as coefficients to define a degree $t-1$ polynomial $f_i(x) = \sum_{j=0}^{t-1} a_{ij} x^j$.
2. Every P_i computes a proof of knowledge to the corresponding secret a_{i0} by calculating $\sigma_i = (R_i, \mu_i)$, such that $k \xleftarrow{\$} \mathbb{Z}_q$, $R_i = g^k$, $c_i = H(i, \Phi, g^{a_{i0}}, R_i)$, $\mu_i = k + a_{i0} \cdot c_i$, with Φ being a context string to prevent replay attacks.
3. Every participant P_i computes a public commitment $\vec{C_i} = \langle \phi_{i0}, \ldots, \phi_{i(t-1)} \rangle$, where $\phi_{ij} = g^{a_{ij}}, 0 \leq j \leq t-1$
4. Every P_i broadcasts $\vec{C_i}, \sigma_i$ to all other participants.
5. Upon receiving $\vec{C_\ell}, \sigma_\ell$ from participants $1 \leq \ell \leq n, \ell \neq i$, participant P_i verifies $\sigma_\ell = (R_\ell, \mu_\ell)$, aborting on failure, by checking $R_\ell \stackrel{?}{=} g^{\mu_\ell} \cdot \phi_{\ell 0}^{-c_\ell}$, where $c_\ell = H(\ell, \Phi, \phi_{\ell 0}, R_\ell)$.
   Upon success, participants delete $\{\sigma_\ell : 1 \leq \ell \leq n\}$.

Round 2

1. Each P_i securely sends to each other participant P_ℓ a secret share $(\ell, f_i(\ell))$, deleting f_i and each share afterward except for $(i, f_i(i))$, which they keep for themselves.
2. Each P_i verifies their shares by calculating: $g^{f_\ell(i)} \stackrel{?}{=} \prod_{k=0}^{t-1} \phi_{\ell k}^{i^k \bmod q}$, aborting if the check fails.
3. Each P_i calculates their long-lived private signing share by computing $s_i = \sum_{\ell=1}^{n} f_\ell(i)$, stores s_i securely, and deletes each $f_\ell(i)$.
4. Each P_i calculates their public verification share $Y_i = g^{s_i}$, and the group's public key $Y = \prod_{j=1}^{n} \phi_{j0}$. Any participant can compute the public verification share of any other participant by calculating $Y_i = \prod_{j=1}^{n} \prod_{k=0}^{t-1} \phi_{jk}^{i^k \bmod q}$.

Figure 1: **KeyGen.** A distributed key generation (DKG) protocol that builds upon the DKG by Pedersen [23]. Our variant includes a protection against rogue key attacks by requiring each participant to prove knowledge of their secret value commits, and requires aborting on misbehaviour.

The Client's implementation correctly follows this procedure. However, we notice the following: it is still possible for a malicious adversary to deviate from the scheme for what concerns polynomial coefficients of degree greater than zero (which are not protected by the ZKP). By crafting arbitrary shares for the other players which are disconnected from the committed coefficient values, a single malicious party is able to complete the key generation protocol with corrupted shares, so that at the end it would not be possible for any combination of parties to perform correct signing, and this would not be immediately detected by the protocol.

Recommendation

Ideally, ZKPs should be added (and checked) for all the coefficients, not only for the low degree one. However, this would probably be too expensive. A simple modification of the protocol to mitigate the effects of the possibility of a DoS attack is as follows:

1) Before Step 4 of Round 1, each participant `i` also computes a commitment of all their secret polynomial, for example as $H(a_{1,0} \,||\, \ldots \,||\, a_{i,(t-1)})$, where `H` is a secure hash function, and `||` denotes concatenation of strings which represent unique encodings of the coefficients. This value is also broadcast together with the others at step 4.
2) At the end of the DKG protocol, a test round is performed where the participants sign a dummy "OK" message and verify that the signature is valid. If the verification fails, each participant is required to reveal their secret polynomials (as they would be invalidated anyway), so that the culprit can be identified and banned from the next iteration of the protocol.

Another simple extra verification step (which can be performed locally by each party without modifying the exchanged elements, so it does not add too much overhead) is the following: At step 5 of round 1, in addition to checking the ZKP of the coefficient of degree zero, also check that the product of all $\Phi_{j(t-1)}$ (including the checking party's own one) is not the identity point. This ensures that the underlying polynomial will actually be of degree t-1, and not lower. We observe that the possibility of this happening by chance are negligible. However, given its low cost this check should still be introduced as an extra defense in depth, and also in order to formally validate the protocol. Notice in fact that a lower degree final polynomial would result in a threshold lower than expected, so it would be a pretty dangerous occurrence.

Status Details

This has been fixed in PR#1434 by adding the sanity check against the point at infinity, and by adding a final round of dummy signing to check that there is enough shares to reconstruct the key.

3.2 Unsuccessful Zeroization of Random Values

Finding ID: KS-CFR-F-02

Severity: Medium

Status: Remediated

Location: `src/multisig/crypto.rs` @ line 128

Description and Impact Summary

The function `random()` generates a new random scalar, but in so doing it creates a variable which is a temporary copy of the generated value.

```
    pub fn random(mut rng: &mut Rng) -> Self {
        use curv::elliptic::curves::secp256_k1::SK;

        let scalar = secp256k1::SecretKey::new(&mut rng);

        let scalar = Secp256k1Scalar::from_underlying(Some(SK(scalar)));
        Scalar(scalar)
    }
```

This way, the output value can be managed (and zeroized) by the calling code, but the internal variable will be deallocated without proper zeroization, which might result in secret values left in memory.

Recommendation

We recommend removing the need for the copy variable. This should be faster than zeroizing the copy before the end of the function.

Status Details

Client corrected the problem in PR #1437.

3.3 Secret Key Shares Stored in Cleartext in Database

Finding ID: KS-CFR-F-04

Severity: Medium

Status: Acknowledged

Location: `src/multisig/client/key_store.rs` @ line 37

Description and Impact Summary

The KeyDB stores key share in clear.

```
    // Save `key` under key `key_id` overwriting if exists
    pub fn set_key(&mut self, key_id: KeyId, key: KeygenResultInfo) {
        self.db.update_key(&key_id, &key);
        self.keys.insert(key_id, key);
    }
```

An attacker with access to the node machine could recover the secret share of the node.

Recommendation

We recommend encrypting and authenticating shares before storing them in the database.

Status Details

The client roadmap contains the key database encryption at rest.

3.4 Various Vulnerabilities Found by `cargo-audit`

Finding ID: KS-CFR-F-04

Severity: Medium

Status: Acknowledged

Location: `various`

Description and Impact Summary

The automated tool `cargo-audit` finds the following vulnerabilities in the dependencies used by the project:

- https://rustsec.org/advisories/RUSTSEC-2020-0159

```
Crate:         chrono
Version:       0.4.19
Title:         Potential segfault in `localtime_r` invocations
Date:          2020-11-10
ID:            RUSTSEC-2020-0159
URL:           https://rustsec.org/advisories/RUSTSEC-2020-0159
Solution:      No safe upgrade is available!
Dependency tree:
chrono 0.4.19
```

- https://rustsec.org/advisories/RUSTSEC-2021-0079

```
Crate:         hyper
Version:       0.10.16
Title:         Integer overflow in `hyper`'s parsing of the `Transfer-
Encoding` header leads to data loss
Date:          2021-07-07
ID:            RUSTSEC-2021-0079
URL:           https://rustsec.org/advisories/RUSTSEC-2021-0079
Solution:      Upgrade to >=0.14.10
Dependency tree:
hyper 0.10.16
```

- https://rustsec.org/advisories/RUSTSEC-2021-0078

```
Crate:         hyper
Version:       0.10.16
Title:         Lenient `hyper` header parsing of `Content-Length` could
allow request smuggling
Date:          2021-07-07
ID:            RUSTSEC-2021-0078
URL:           https://rustsec.org/advisories/RUSTSEC-2021-0078
Solution:      Upgrade to >=0.14.10
```

- https://rustsec.org/advisories/RUSTSEC-2021-0130

```
Crate:         lru
Version:       0.6.6
Title:         Use after free in lru crate
Date:          2021-12-21
ID:            RUSTSEC-2021-0130
URL:           https://rustsec.org/advisories/RUSTSEC-2021-0130
Solution:      Upgrade to >=0.7.1
```

- https://rustsec.org/advisories/RUSTSEC-2020-0071

```
Crate:         time
Version:       0.1.44
Title:         Potential segfault in the time crate
Date:          2020-11-18
ID:            RUSTSEC-2020-0071
URL:           https://rustsec.org/advisories/RUSTSEC-2020-0071
Solution:      Upgrade to >=0.2.23
Dependency tree:
time 0.1.44
```

- https://rustsec.org/advisories/RUSTSEC-2021-0124

```
Crate:         tokio
Version:       0.2.25
Title:         Data race when sending and receiving after closing a
`oneshot` channel
Date:          2021-11-16
ID:            RUSTSEC-2021-0124
URL:           https://rustsec.org/advisories/RUSTSEC-2021-0124
Solution:      Upgrade to >=1.8.4, <1.9.0 OR >=1.13.1
Dependency tree:
tokio 0.2.25
```

- https://rustsec.org/advisories/RUSTSEC-2021-0110

```
Crate:         wasmtime
Version:       0.27.0
Title:         Multiple Vulnerabilities in Wasmtime
Date:          2021-09-17
ID:            RUSTSEC-2021-0110
URL:           https://rustsec.org/advisories/RUSTSEC-2021-0110
Solution:      Upgrade to >=0.30.0
```

Recommendation

We recommend following `cargo-audit`'s pointers to proposed solutions where applicable.

Status Details

Acknowledged by the client.

3.5 Use of Outdated, Unmaintained, or Deprecated Crates

Finding ID: KS-CFR-F-05

Severity: Medium

Status: Acknowledged

Location: `Cargo.toml`

Description and Impact Summary

`cargo-audit` shows that the following crates are either deprecated or unmaintained. Deprecated and unmaintained dependencies can create security issues due to unpatched vulnerabilities that remain persistent in the code. The impact of these vulnerabilities could be severe depending on the location and criticality of the functions used.

- `tempdir`

```
Crate:         tempdir
Version:       0.3.7
Warning:       unmaintained
Title:         `tempdir` crate has been deprecated; use `tempfile`
instead
Date:          2018-02-13
ID:            RUSTSEC-2018-0017
URL:           https://rustsec.org/advisories/RUSTSEC-2018-0017
```

- `net2`

```
Crate:         net2
Version:       0.2.37
Warning:       unmaintained
Title:         `net2` crate has been deprecated; use `socket2`
instead
Date:          2020-05-01
ID:            RUSTSEC-2020-0016
URL:           https://rustsec.org/advisories/RUSTSEC-2020-0016
Dependency tree:
```

- `failure`

```
Crate:         failure
Version:       0.1.8
Warning:       unmaintained
Title:         failure is officially deprecated/unmaintained
Date:          2020-05-02
ID:            RUSTSEC-2020-0036
URL:           https://rustsec.org/advisories/RUSTSEC-2020-0036
```

- `difference`

```
Crate:          difference
Version:        2.0.0
Warning:        unmaintained
Title:          difference is unmaintained
Date:           2020-12-20
ID:             RUSTSEC-2020-0095
URL:            https://rustsec.org/advisories/RUSTSEC-2020-0095
```

Moreover, at `Cargo.toml` line 56 Crate `sha2` is used in version 0.9.5 which is not the latest and detected as yanked.

Recommendation

Alternatives should be used if possible.

Status Details

Acknowledged by the client.

3.6 64-bit Values Clipped to 32-bit Values

Finding ID: KS-CFR-F-06

Severity: Low

Status: Remediated

Location: `src/multisig/crypto.rs` @ line 141

`src/multisig/client/common/broadcast_verification.rs` @ line 50

Description and Impact Summary

For example:

```
    pub fn from_usize(a: usize) -> Self {
        Scalar(ECScalar::from_bigint(&BigInt::from(a as u32)))
    }
```

This might lead to unexpected behavior.

Recommendation

We recommend using the `usize` type.

Status Details

Corrected by the client in PR #1438.

3.7 Hash Values Not Mapped to Correct Group

Finding ID: KS-CFR-F-07

Severity: Low

Status: Acknowledged

Location: `src/multisig/client/keygen/keygen_frost.rs` @ line 78

`src/multisig/client/signing/frost.rs` @ line 151

Description and Impact Summary

Hash values (256 bits) should map to elements of Z_q or Z^*_q but this reduction is not performed.

Recommendation

We recommend mapping hash values to correct ranges by using procedures to avoid modular bias, as explained for example at https://research.kudelskisecurity.com/2020/07/28/the-definitive-guide-to-modulo-bias-and-how-to-avoid-it/ . More in detail, the recommended procedure (also suggested for example in https://datatracker.ietf.org/doc/draft-irtf-cfrg-hash-to-curve/) is to perform a modular reduction of a larger value, in order to reduce the bias to a negligible amount. Compared to rejection sampling, this has the advantage that the procedure is guaranteed to halt in a predictable time, and it adds little overhead.

Status Details

The Client acknowledges the issue but prefers to maintain the same approach used also in Bitcoin and other similar secp256k1 applications. The reason is that the order of this curve is sufficiently close to 2^{256} to make any bias unobservable in practice.

4. OTHER OBSERVATIONS

This section 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.

4.1 Parameters Control Left to Blockchain Governance Layer

Observation ID: KS-CFR-O-01

Description and Impact Summary

We notice that several undesirable behaviors are only avoided by the guarantees that the higher level blockchain governance layer explicitly avoids certain configurations. Examples:

- A large number of parties will make the validator to crash. This is avoided by a mechanism in place that ensures that the number of messages processed at each stage is no larger than the number of ceremony participants (which is determined by the blockchain rules and limited to some reasonable number, 150 in this case).
- Reuse of `CeremonyIds` is possible, making ceremony context hash not unique. It is possible for the blockchain components to request ceremonies with previously used ceremonies id, but only if the blockchain governance made the chain do so: governance has total control over the chain's behavior, it's a collection of stakeholders who vote to do things, i.e., upgrade the chain's behavior. Another case is if the chain locally reorganizes, e.g., a blockchain node realizes it is on a fork on the chain and corrects its state. According to the Client, neither of these aspects are typical behavior. The Client acknowledged that old ceremony ids would be accepted even if there is no reason to do it and probably they shouldn't do it.
- A single signer can join multiple times a ceremony. Also in this case, the chain governance is responsible for deduplication,

Recommendation

We recommend enforcing safeguards at a lower level in order to avoid misconfiguration issues.

4.2 On the Constraint of Public Key x-Coordinate

Observation ID: KS-CFR-O-02

Description and Impact Summary

In `src/multisig/client/keygen/keygen_stages.rs :110` the public key coordinate x is chosen to be less than the curve order divided by two. This reduces the number of possible valid public keys.

```
/// Check if the public key's x coordinate is smaller than "half secp256k1's order",
/// which is a requirement imposed by the Key Manager contract
pub fn is_contract_compatible(pk: &secp256k1::PublicKey) -> bool {
    let pubkey = cf_chains::eth::AggKey::from(pk);

    let x = BigInt::from_bytes(&pubkey.pub_key_x);
    let half_order = BigInt::from_bytes(&secp256k1::constants::CURVE_ORDER) / 2 + 1;

    x < half_order
}
```

Recommendation

Reducing the entropy pool of allowed keys is in general an undesirable behavior. The need for such requirement should be better documented.

Notes

This is a necessity of the client because of the usage of Ethereum `ecrecover`.

4.3 Whitepaper is Outdated

Observation ID: KS-CFR-O-03

Description and Impact Summary

The whitepaper at https://chainflip.io/whitepaper.pdf does not mention the use of FROST.

Recommendation

Should be updated if possible.

Notes

The Client confirmed that this document is out of date, as it was written before the decision to adopt FROST. The client is working on an updated version.

4.4 On Error Management

Observation ID: KS-CFR-O-04

Description and Impact Summary

Error management is sometimes not properly addressed, leaving the application to crash or panic on unexpected input rather than trying to recover from the error. Some examples:

- Cases where `expect/unwrap` is called in a function that returns a `Result`. The `Result` type is not utilized for handling all the errors of the function, causing a panic. For example in `src/multisig/client/signing/frost.rs:257` and `src/multisig/db/persistent.rs:67`.
- In `src/multisig/client/state_runner.rs:59`, a request coming from the State Chain with a ceremony id different from the one previously set can make the validator crash. This would make the node to restart, losing all the current ceremony states. The protocol will react by banning or suspending nodes, but this could still lead to a temporary DoS.
- In general, in most `expect`, `unwrap`, and `assert` calls.

Recommendation

We recommend trying to recover from errors whenever possible instead of leaving the implementation crash via a panic, particularly if this allows a selective shutdown of validators. Furthermore, log these critical events before taking action in a persistent way.

4.5 Compilation Error in `cargo test`

Observation ID: KS-CFR-O-05

Description and Impact Summary

When running `cargo test` the following error appears:

```
error: internal compiler error: encountered incremental compilation error
with evaluate_obligation(d0144488feafe20e-26a80b73469dc7f0)
  |
  = help: This is a known issue with the compiler. Run `cargo clean -p
chainflip_node` or `cargo clean` to allow your project to compile
  = note: Please follow the instructions below to create a bug report with
the provided information
  = note: See <https://github.com/rust-lang/rust/issues/84970> for more
information

thread 'rustc' panicked at 'Found unstable fingerprints for
evaluate_obligation(d0144488feafe20e-26a80b73469dc7f0):
Ok(EvaluatedToOkModuloRegions)',
/rustc/b3d11f95cc5dd687fdd185ce91e02ebe40e6f46b/compiler/rustc_query_system
/src/query/plumbing.rs:624:9
note: run with `RUST_BACKTRACE=1` environment variable to display a
backtrace

error: internal compiler error: unexpected panic

note: the compiler unexpectedly panicked. this is a bug.

note: we would appreciate a bug report: https://github.com/rust-
lang/rust/issues/new?labels=C-bug%2C+I-ICE%2C+T-compiler&template=ice.md

note: rustc 1.55.0-nightly (b3d11f95c 2021-07-04) running on x86_64-
unknown-linux-gnu

note: compiler flags: -C embed-bitcode=no -C debuginfo=2 -C incremental --
crate-type lib

note: some of the compiler flags provided by cargo are hidden
```

Recommendation

A possible solution is to increase the `ulimit` parameter with `ulimit -n 1000000`.

4.6 Test `can_subscribe_infura` Fails

Observation ID: KS-CFR-O-06

Description and Impact Summary

When running `cargo test`:

```
test test_all_key_manager_events ... FAILED

failures:

---- test_all_key_manager_events stdout ----
[DEBG]  chainflip_engine::eth - Connecting new web3 client to
ws://fake.rinkeby.endpoint/flippy1234
thread 'test_all_key_manager_events' panicked at 'Couldn't create
EthRpcClient: at engine/src/eth/mod.rs line 193 column 30

Caused by:
    0: IO error: failed to lookup address information: Name or service not
known
    1: failed to lookup address information: Name or service not known',
engine/tests/can_subscribe_infura.rs:21:10
note: run with `RUST_BACKTRACE=1` environment variable to display a
backtrace


failures:
    test_all_key_manager_events

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.04s

error: test failed, to rerun pass '-p chainflip-engine --test
can_subscribe_infura'
```

Recommendation

Debug the failure.

4.7 Warnings from `cargo clippy`

Observation ID: KS-CFR-O-07

Description and Impact Summary

When running `cargo clippy` we obtained the following warnings:

```
warning: use of irregular braces for `write!` macro
  --> engine/src/eth/mod.rs:52:10
   |
52 | #[derive(Error, Debug)]
   |          ^^^^^
   |
   = note: `#[warn(clippy::nonstandard_macro_braces)]` on by default
help: consider writing `Error`
  --> engine/src/eth/mod.rs:52:10
   |
52 | #[derive(Error, Debug)]
   |          ^^^^^
   = help: for further information visit https://rust-lang.github.io/rust-
clippy/master/index.html#nonstandard_macro_braces
   = note: this warning originates in the derive macro `Error` (in Nightly builds,
run with -Z macro-backtrace for more info)


warning: local variable doesn't need to be boxed here
   --> state-chain/pallets/cf-witnesser/src/lib.rs:160:4
    |
160 |             call: Box<<T as Config>::Call>,
    |             ^^^^
    |
    = note: `#[warn(clippy::boxed_local)]` on by default
    = help: for further information visit https://rust-lang.github.io/rust-
clippy/master/index.html#boxed_local

warning: local variable doesn't need to be boxed here
   --> state-chain/pallets/cf-witnesser/src/lib.rs:186:4
    |
186 |             call: Box<<T as Config>::Call>,
    |             ^^^^
    |
    = help: for further information visit https://rust-lang.github.io/rust-
clippy/master/index.html#boxed_local


warning: local variable doesn't need to be boxed here
   --> state-chain/pallets/cf-governance/src/lib.rs:283:4
    |
283 |             call: Box<<T as Config>::Call>,
    |             ^^^^
    |
    = note: `#[warn(clippy::boxed_local)]` on by default
    = help: for further information visit https://rust-lang.github.io/rust-
clippy/master/index.html#boxed_local
```

Recommendation

Debug these warnings.

APPENDIX A: ABOUT KUDELSKI SECURITY

Kudelski Security is an innovative, independent Swiss provider of tailored cyber and media security solutions to enterprises and public sector institutions. Our team of security experts delivers end-to-end consulting, technology, managed services, and threat intelligence to help organizations build and run successful security programs. Our global reach and cyber solutions focus is reinforced by key international partnerships.

Kudelski Security is a division of Kudelski Group. For more information, please visit https://www.kudelskisecurity.com.

Kudelski Security

Route de Genève, 22-24

1033 Cheseaux-sur-Lausanne

Switzerland

Kudelski Security

5090 North 40th Street

Suite 450

Phoenix, Arizona 85018

APPENDIX B: DOCUMENT HISTORY

VERSION	STATUS	DATE	AUTHOR	COMMENTS
0.1	Draft	28 February 2022	Tommaso Gagliardoni	
0.2	Draft	3 March 2022	Nathan Hamiel	Fixed typos
0.3	Draft	9 March 2022	Tommaso Gagliardoni	Removed one finding, added code snippet
0.4	Draft	18 March 2022	Tommaso Gagliardoni	Updated status of findings according to Client's feedback
0.5	Draft	22 March 2022	Tommaso Gagliardoni	Modified severity of one finding
1.0	Final (Proposal)	13 April 2022	Tommaso Gagliardoni	Added Client's feedback
2.0	Final	26 April 2022	Tommaso Gagliardoni	Client accepted feedback

REVIEWER	POSITION	DATE	VERSION	COMMENTS
Nathan Hamiel	Head of Security Research	3 March 2022	0.1	
Tommaso Gagliardoni	Senior Cryptography Expert	9 March 2022	0.2	
Nathan Hamiel	Head of Security Research	9 March 2022	0.3	
Nathan Hamiel	Head of Security Research	18 March 2022	0.4	
Nathan Hamiel	Head of Security Research	23 March 2022	0.5	
Nathan Hamiel	Head of Security Research	13 April 2022	1.0	

APPROVER	POSITION	DATE	VERSION	COMMENTS
Nathan Hamiel	Head of Security Research	28 February 2022	0.1	
Nathan Hamiel	Head of Security Research	3 March 2022	0.2	
Nathan Hamiel	Head of Security Research	9 March 2022	0.3	
Nathan Hamiel	Head of Security Research	18 March 2022	0.4	
Nathan Hamiel	Head of Security Research	23 March 2022	0.5	
Nathan Hamiel	Head of Security Research	23 March 2022	1.0	

APPENDIX C: SEVERITY RATING DEFINITIONS

Kudelski Security uses a custom approach when determining criticality of identified issues. This is meant to be simple and fast, providing customers with a quick at a glance view of the risk an issue poses to the system. As with anything risk related, these findings are situational. We consider multiple factors when assigning a severity level to an identified vulnerability. A few of these include:

- Impact of exploitation
- Ease of exploitation
- Likelihood of attack
- Exposure of attack surface
- Number of instances of identified vulnerability
- Availability of tools and exploits

SEVERITY	DEFINITION
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	Informational findings are best practice steps that can be used to harden the application and improve processes.