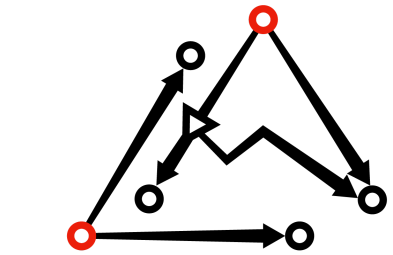

Least Authority

PRIVACY MATTERS

Serai DKG Implementation

Security Audit Report

Magic Grants

Final Audit Report: 13 July 2026

Table of Contents

Overview

Background

Magic Grants has requested that Least Authority perform a security audit of the implementation of Serai's Distributed Key Generation, which is premised on Publicly Verifiable Secret Sharing.

Project Dates

- **June 9, 2026 - June 25, 2026:** Initial Code Review *(Completed)*
- **June 29, 2026:** Delivery of Initial Audit Report *(Completed)*
- **July 13, 2026:** Verification Review
- **July 13, 2026:** Delivery of Final Audit Report

Review Team

- George Gkitsas, Security / Cryptography Researcher and Engineer
- Miguel Quaresma, Security Researcher and Engineer
- Burak Atasoy, Project Manager
- Jessy Bissal, Technical Editor

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the implementation of Serai's Distributed Key Generation, which is premised on Publicly Verifiable Secret Sharing, followed by issue reporting, along with mitigation and remediation instructions as outlined in this report.

The following code repository was considered in scope for the review:

- `Serai`: https://github.com/serai-dex/serai/tree/1ae4797da43f9798365785877e3ccca8e2e3f874/crypto/dkg/evrf

Specifically, we examined the following Git revision for our initial review:

- `1ae4797da43f9798365785877e3ccca8e2e3f874`

For the verification, we examined the following Git revision:

- `4c34222fcee860d720a8f39da0484cea4644f456`

For the review, this repository was cloned for use during the audit and for reference in this report:

- `serai-dex-serai`: https://github.com/LeastAuthority/serai-dex-serai

All file references in this document use Unix-style paths relative to the project's root directory.

In addition, any dependency and third-party code, unless specifically mentioned as in scope, were considered out of scope for this review.

Supporting Documentation

The following documentation was available to the review team:

- eVRF DKG: https://github.com/LeastAuthority/serai-dex-serai/tree/audit/crypto/dkg/evrf

In addition, this audit report references the following documents:

- D. Boneh, I. Haitner, Y. Lindell, and G. Segev, "Exponent-VRFs and Their Applications." *IACR Cryptology ePrint Archive*, 2024, [BHL+24]

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;
- Vulnerabilities within each component and whether the interaction between the components is secure;
- Whether requests are passed correctly to the network core;
- Key management, including secure private key storage and management of encryption and signing keys;
- Denial of Service (DoS) and other security exploits that would impact the intended use or disrupt the execution;
- Protection against malicious attacks and other ways to exploit;
- Inappropriate permissions and excess authority;
- Data privacy, data leaking, and information integrity; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

Serai is a decentralized exchange (DEX) that allows users to trade assets across heterogeneous blockchains through a liquidity-pool (AMM) trading model. Its defining characteristic is that it does not use a custodial bridge or wrapped-token middleman. User funds are held in a secured threshold-multisig wallet collectively controlled by a rotating validator set, so no single party can move funds.

The `crypto/` directory is a cryptographic stack built on the standard Rust `ff/group` elliptic-curve traits. Its main functionality is threshold signing used in multisig. This includes a distributed key generation using an eVRF-based DKG. The DKG uses verifiable encryption with zero-knowledge cryptography built on generalized Bulletproofs.

In this context, our team reviewed the correctness of the eVRF DKG, the soundness and correctness of the zero-knowledge circuits, and the codebase for general vulnerabilities. Our findings focused on implementation correctness, proof and transcript handling, and areas where edge cases or future curve configurations could affect reliability.

During our analysis, we found a divergence between the eVRF DKG specification and the implementation, where the verifiable-encryption mask omitted the randomized extractor key (Issue B). We also identified an issue in the transcript length calculation that results from an assumption about the size of the group elements, which could cause valid proofs to be rejected (Issue A).

System Design

Our team examined Serai's Distributed Key Generation implementation and found that it demonstrates careful cryptographic implementation, strong theoretical grounding, and well-supported design rationale.

Dependencies

Our team did not identify any vulnerabilities in the implementation's use of dependencies.

Code Quality

We performed a combined manual and AI-assisted review of the repositories in scope and found the code to be well-organized and aligned with development best practices.

Tests

The repositories in scope include tests with sufficient coverage levels. Branch coverage is approximately 85%, and lines-of-code (LOC) coverage is approximately 77%.

Documentation and Code Comments

The project documentation clearly outlines the project's purpose and sufficiently describes the system's intended functionality. Additionally, the codebase includes descriptive comments, which aid in understanding the intended behavior of the relevant components.

Scope

The scope of this review was sufficient and included all security-critical components.

Specific Issues & Suggestions

We list the issues and suggestions identified during the review in descending order of severity. In most cases, remediation of an issue is preferable, but mitigation is suggested as another option for cases where a trade-off could be required.

ISSUE / SUGGESTION	SEVERITY	STATUS
Issue A: Transcript Length Calculation Assumes All Proof Points Use the ToweringCurve Size	Low	Resolved
Issue B: Verifiable-Encryption Mask Omits Randomized Extractor Key	Low	Resolved

Issue A: Transcript Length Calculation Assumes All Proof Points Use the ToweringCurve Size

Location

`crypto/dkg/evrf/src/proof/mod.rs#L332`

`crypto/dkg/evrf/src/proof/mod.rs#L640`

crypto/dkg/evrf/src/lib.rs#L63

Synopsis

The function `transcript_len`, used to calculate the length of a transcript, assumes every serialized group element in the proof has the byte length of `C::ToweringCurve`. However, the transcript also contains ECDH commitments serialized as `C::EmbeddedCurve` points. If a future curve configuration uses embedded and towering curves with different encoded point sizes, `transcript_len` will return an incorrect proof size. This can cause participation deserialization or proof verification to fail for otherwise valid proofs.

Preconditions

The issue requires a curve configuration where `C::EmbeddedCurve` and `C::ToweringCurve` group encodings have different serialized lengths.

Likelihood

Medium.

The issue is not triggered by the currently used curve configuration, but it would be triggered deterministically if the eVRF code were instantiated with a curve configuration where `C::EmbeddedCurve` and `C::ToweringCurve` group encodings have different serialized lengths.

Impact

Low.

A successful trigger would cause proof parsing or verification failure due to insufficient or extra transcript bytes. This would most likely result in valid participations being rejected or deserialization becoming misaligned before encrypted shares are read.

Severity

Low.

Technical Details

The `transcript_len` function computes the serialized proof length by counting group elements and scalar elements, then multiplying all group elements by the encoded size of `C::ToweringCurve` group elements:

```rust
group_elements *  <<C::ToweringCurve as WrappedGroup>::G as
                   GroupEncoding>::Repr::default().as_ref().len()
```

However, during proving, the transcript also includes two ECDH commitment points of type `C::EmbeddedCurve` per participant:

```rust
for ecdh_commitment in ecdh_commitments {
  transcript.push_point(&ecdh_commitment);
```

```
}
```

When reading a participation, the `Participation::read` function relies on the size computed by `transcript_len(t, n)` to determine how many proof bytes to consume before reading encrypted secret shares. If `C::EmbeddedCurve` points serialize to a different length than `C::ToweringCurve` points, the reader may consume too few or too many bytes for the proof. This can shift the stream boundary between the proof and encrypted shares, or leave the verifier with a proof byte slice that does not match the actual transcript layout.

Remediation

We recommend computing transcript length using separate counters for towering-curve points and embedded-curve points, or explicitly adding the ECDH commitment byte contribution using `C::EmbeddedCurve`'s encoded point length.

Status

The Serai DEX team has resolved the issue as recommended.

Verification

Resolved.

Issue B: Verifiable-Encryption Mask Omits Randomized Extractor Key

Location

`crypto/dkg/evrf/src/proof/mod.rs#L174-L212`

`crypto/dkg/evrf/src/proof/mod.rs#L552-L586`

`crypto/dkg/evrf/src/lib.rs#L217-L221`

`crypto/dkg/evrf/src/shares.rs#L84-L92`

Synopsis

The implementation encrypts each posted secret share with a mask computed as `x1 + x2`, where `x1` and `x2` are x-coordinates from two ECDH outputs. The referenced eVRF DKG construction [BHL+24] uses a randomized extractor, `k' * x1 + x2`, to argue that the result is statistically close to uniform. The implementation omits `k'`, so the hiding argument for publicly posted encrypted shares does not directly apply.

Preconditions

An attacker must be able to observe DKG participations, including encrypted secret shares, ECDH commitments, and proofs. The issue also requires the distribution of `x1 + x2` for the supported embedded curve to be distinguishably biased in an exploitable form.

Likelihood

Low.

A practical exploit would require a concrete distinguisher or estimator for the distribution of the sum of two embedded-curve x-coordinates.

Impact

Medium.

Exploitation of this issue could result in information leakage about publicly posted encrypted secret shares. Sufficient leakage across shares could weaken the secrecy margin of the DKG output.

Severity

Low.

Technical Details

In the `verifiable_encryption` function, the circuit accumulates the two ECDH x-coordinates with coefficient ONE:

```rust
lincomb = lincomb.term(<C::ToweringCurve as WrappedGroup>::F::ONE, point.x());
```

The resulting linear combination is constrained to equal the committed encryption mask. In the prover, the same value is computed by adding each ECDH x-coordinate:

```rust
*encryption_key += dh_x;
```

The share is then encrypted as:

```rust
*share + *encryption_key;
```

In the full DDH eVRF construction, the extractor is `k' * x1 + x2`, with `k'` acting as the random universal-hash key used by the leftover-hash argument. The paper [BHL+24] also notes that deterministic two-source extraction from the two biased x-coordinate sources is not sufficient without additional assumptions about the structure of the x-coordinate set. Therefore, the implementation relies on the unproven assumption that the bare sum `x1 + x2` is sufficiently close to uniform for the supported curve instantiation.

Remediation

We recommend implementing the proven randomized extractor form for the verifiable-encryption mask. This can be achieved by sampling or deriving a uniform public `k'` independently of the ECDH x-coordinate samples, binding it into the relevant transcript or verification context, and computing:

```rust
mask = k' * x1 + x2
```

The circuit, prover, encryption, and decryption paths should all use the same expression.

Status

The Serai DEX team has resolved the issue as recommended.

Verification

Resolved.

About Least Authority

Founded in 2011 and based in Berlin, Germany, since 2016, Least Authority provides security consulting for privacy-preserving technologies and decentralized systems. We are committed to transparency, openness, and advancing secure digital infrastructure.

We believe that people have a fundamental right to privacy and that secure solutions enable people to use the internet and other connected technologies more freely. Our security consulting services help teams make their solutions more resistant to unauthorized access and unintended system manipulation. We support teams from the design phase through production launch and beyond.

Our team has experience reviewing code for security vulnerabilities and specific attack vectors in multiple languages, including Rust, Go, Python, C, C++, Solidity, Cairo, Circom, Noir, Haskell, OCaml, TypeScript, Swift, ZoKrates, and others. We have reviewed implementations of cryptographic protocols and distributed system architecture in blockchains, payments, smart contracts, zero-knowledge protocols, and consensus protocols. Additionally, we use various tools to scan code and networks and build custom tools as necessary.

For more information about our security consulting, please visit:
https://leastauthority.com/security-consulting.

Our Methodology

We use a transparent review process that provides the project team visibility into our findings, assumptions, and remediation considerations throughout the audit. The goals of our security audits are to improve the quality of the systems we review and support sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Code Review

In manually reviewing all of the code, we identify potential issues, including those related to code logic, error handling, cryptographic implementations, and correctness with respect to specifications, standards, and best practices. We also evaluate areas where more defensive programming could reduce the risk of future mistakes and accelerate future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

When appropriate, we also use artificial intelligence (AI) to support the efficiency of our security audits without reducing rigor or reliability. AI-assisted outputs are carefully guided, reviewed, and validated by our team so that conclusions remain accurate and defensible. This supports stronger audit decisions, not only faster review.

Vulnerability Analysis

Our audit techniques include manual code analysis, user interface interaction, and white box penetration testing, supplemented by various tools, including AI, that support the code review process. Discussions with the development team help clarify their vision for the software. Installing and using the relevant software helps us understand user interactions and roles, while also informing our analysis of threat models and attack surfaces. Design documentation, relevant papers, other audit results, similar projects, source code dependencies, and open issue tickets may also be reviewed to inform the analysis. We identify potential vulnerabilities, and, when appropriate, create issue entries. For each entry, we follow the issue investigation and remediation process described below.

Documenting Results

We follow a conservative and transparent process for analyzing potential security vulnerabilities and supporting successful remediation. Whenever a potential issue is identified, we raise it with the client through a dedicated secure communication channel to discuss its validity and impact, to the extent these can be determined at that stage. This process allows potentially relevant findings to be reviewed with the client early, before they are documented in greater detail in the initial audit report. For each confirmed issue, we assess impact and feasibility and use those ratings to determine the overall severity, following the OWASP Risk Severity Matrix.

Suggested Solutions

We identify immediate and comprehensive mitigations available to live deployments and then suggest remediation requirements for future releases. When code is already deployed in production, we include mitigation strategies in the report to provide interim support until appropriate remediation is available. The mitigation and remediation recommendations should be reviewed by the development team and deployment engineers. After delivering the Initial Audit Report, we remain available to support implementation of the proposed mitigations and remediations before the verification review.

Development teams should address findings in an order and with an urgency that prioritizes the security of their users. Findings should be addressed as soon as possible, but remediation should be carried out pragmatically and should not cause unnecessary disruption to project timelines.

Although issues must be handled on a case-by-case basis, we work toward a resolution timeline that balances the impact on users and the needs of the project team.

Resolutions & Publishing

Once the findings are comprehensively addressed, we complete a verification review to assess whether the issues and suggestions have been sufficiently addressed. When this analysis is complete, we update the report and provide a Final Audit Report that can be published in full. If any critical issues remain, we recommend that the Final Audit Report not be published and that users and other stakeholders be alerted to the impact. We encourage all findings to be addressed and the Final Audit Report to be shared publicly to support transparency and advance security knowledge within the industry.