Lelantus

Summary Report

July 20, 2020

Prepared For:
Reuben Yap | *Zcoin*
reuben@zcoin.io

Prepared By:
Jim Miller | *Trail of Bits*
james.miller@trailofbits.com

Will Song | *Trail of Bits*
will.song@trailofbits.com

Suha Hussain | *Trail of Bits*
suha.hussain@trailofbits.com

Review Summary

From July 13 to July 20, 2020, Trail of Bits performed an assessment of Lelantus with two engineers and reported three issues in total, ranging from informational to high severity. The assessment was performed on commit 7f7e64e5 of the `lelantus` repository with a primary focus on `liblelantus` via manual review of the codebase. We also integrated some static analysis tools, such as `cppcheck`, `rats`, and `flawfinder`, and reviewed their output, which did not result in any additional findings.

We closely reviewed `liblelantus` and compared it to the Lelantus protocol to ensure that the implementation complies with this specification. Aside from a few typos in the paper and some underspecified areas, we found that the implementation largely complies with its specification. We report no security issues related to the implementation matching the protocol.

We also sought to answer various questions about the security of Lelantus. We focused on flaws that would allow an attacker to:

- Create coins out of thin air.
- De-anonymize other users.
- Double-spend coins.

We reported one informational and one high-severity finding related to overflows in the balancing equation, which in turn relates to creating coins out of thin air. The high-severity finding was introduced in the most recent commit, which had not gone through the normal review process, according to Zcoin. In addition to this, we reported one medium-severity issue related to the generation of challenges in the Fiat-Shamir transformation.

During the week of July 20, 2020, Trail of Bits reviewed fixes for the reported issues. We concluded that the fixes resolve the issues raised and did not introduce any new issues.

On the following page, we review the maturity of the codebase and the likelihood of future issues. In each area of control, we rate the maturity from strong to weak, or missing, and give a brief explanation of our reasoning. Zcoin should consider these steps to improve their security maturity:

- Address the findings presented in this report.
- Write additional negative testing to verify that bad inputs, such as empty vectors and the point at infinity, are caught during verification.

Code Maturity Evaluation

Category Name	Description
Access Controls	**Strong.** We do not report any issues related to access controls.
Arithmetic	**Moderate.** We report two issues related to potential integer overflows, one of which could be exploited.
Assembly Use	**Not Reviewed.**
Centralization	**Satisfactory.** We found that most of the issues discovered tend to be partially or entirely protected by controls in other parts of the system.
Upgradeability	**Not Reviewed.**
Function Composition	**Strong.** We found the codebase to be separated into functions with clear purpose. We have no concerns in this area.
Front-Running	**Not Reviewed.**
Key Management	**Strong.** We do not report any issues related to key management.
Monitoring	**Not Reviewed.**
Specification	**Satisfactory.** We report a few typos in the paper and a few other areas that seem to be underspecified. Aside from this, we found the specification to be clear and detailed, and the implementation closely aligned with it.
Testing & Verification	**Moderate.** Overall, there was pretty good testing coverage. However, we think the system could benefit from additional negative testing. For example, we recommend adding tests to confirm that bad inputs are caught by the membership checks of each verifier.

Appendix A. Code Maturity Classifications

Code Maturity Classes	
Category Name	**Description**
Access Controls	Related to the authentication and authorization of components.
Arithmetic	Related to the proper use of mathematical operations and semantics.
Assembly Use	Related to the use of inline assembly.
Centralization	Related to the existence of a single point of failure.
Upgradeability	Related to contract upgradeability.
Function Composition	Related to separation of the logic into functions with clear purpose.
Front-Running	Related to resilience against front-running.
Key Management	Related to the existence of proper procedures for key generation, distribution, and access.
Monitoring	Related to use of events and monitoring procedures.
Specification	Related to the expected codebase documentation.
Testing & Verification	Related to the use of testing techniques (unit tests, fuzzing, symbolic execution, etc.).

Rating Criteria	
Rating	**Description**
Strong	The component was reviewed and no concerns were found.
Satisfactory	The component had only minor issues.
Moderate	The component had some issues.
Weak	The component led to multiple issues; more issues might be present.

Missing	The component was missing.
Not Applicable	The component is not applicable.
Not Considered	The component was not reviewed.
Further Investigation Required	The component requires further investigation.

Appendix B. Non–Security-Related Findings

This appendix contains findings that do not have immediate or obvious security implications.

- **The Lelantus paper contains some typos.** See this issue for more details.

- **There are a few areas of the implementation that appear to be underspecified or that differed from the specification but do not have implications for security:**

    - `Sigmaextended_verifier.cpp` performs a batch verification of the `A`, `B`, `C`, and `D` values.
    - `Verify_rangeproof` pads the output coin vector with the point at infinity to obtain a size of `m`.

- **There are a few areas where additional guardrails could be added for the prover.** It is possible (but admittedly unlikely) for an honest prover to generate random values (e.g., calls to `randomize` and `generate_challenge`) that will fail membership checks upon verification. Consider adding these membership checks to the prover side.