Least Authority

PRIVACY MATTERS

Groth16 Verifier in EVM Smart Contract

Security Audit Report

Worldcoin

Final Audit Report: 25 August 2023

Table of Contents

Overview

Background

Worldcoin has requested that Least Authority perform a security audit of the Groth16 Verifier in the EVM Smart Contract.

Project Dates

- **August 16, 2023 - August 18, 2023:** Initial Code Review *(Completed)*
- **August 22, 2023:** Delivery of Initial Audit Report *(Completed)*
- **August 25, 2023 :** Verification Review *(Completed)*
- **August 25, 2023:** Delivery of Final Audit Report *(Completed)*

Review Team

- Nathan Ginnever, Security Researcher and Engineer
- Jasper Hepp, Security Researcher and Engineer
- Ahmad Jawid Jamiulahmadi, Security Researcher and Engineer
- Mirco Richter, Cryptography Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the Groth16 Verifier in the EVM Smart Contract followed by issue reporting, along with mitigation and remediation instructions as outlined in this report.

The following PR is considered in scope for the review:
- Consensys / Gnark:
  https://github.com/Consensys/gnark/pull/810

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

- `30141357125254002fbac1319c3b6843e46e2b91`

For the verification, we examined the Git revision:

- `c14ec3381860e1c61f8a5a6431f7dd377a514e13`

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:
- Website:
  https://worldcoin.org

In addition, this audit report references the following documents:

- J. Groth, "On the Size of Pairing-based Non-interactive Arguments." *IACR Cryptology ePrint Archive*, 2016, [Groth16]
- G. Wood, "Ethereum: A Secure Decentralised Generalised Transaction Ledger." *Ethereum*, 2023, [Wood23]
- Least Authority "The Moon Math Manual": https://leastauthority.com/community-matters/moonmath-manual
- BN254 Point Compression for L2s: https://xn--2-umb.com/23/bn254-compression/#bn254-compression
- Gnark Tests: https://github.com/kustosz/gnark-tests/blob/feef966f84fd72acd5a8e193855d38a5abe7646f/solidity/solidity_groth16_test.go
- EIP 197: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-197.md
- NatSpec: https://docs.soliditylang.org/en/v0.8.21/natspec-format.html
- Golang data-driven code generator: https://pkg.go.dev/text/template

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;
- Potential misuse and gaming of the smart contracts;
- Attacks that impacts funds, such as the draining or manipulation of funds;
- Mismanagement of funds via transactions;
- Vulnerabilities in the smart contracts' code;
- Protection against malicious attacks and other ways to exploit the smart contracts;
- 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

Our team performed a review of a PR, which modifies the Golang contract template of the [Groth16] proof verifier in the `gnark` library to optimize the verifier for both L1 and L2 gas consumption. The PR reimplements the function `verifyProof` in a gas optimized way in Solidity assembly and implements a new function, `verifyCompressedProof`, that adds the possibility to pass in compressed proofs.

In our review, we examined the mathematical description of compressing a curve point in the context of [Groth16]. We checked the constants in the `solidity.go` file and verified their correctness with sage math. We also checked the implementation in `solidity.go` against the mathematical description.

Additionally, we reviewed the code for proper initialization and storage of data to verify whether the precompile functions were used correctly for pairing operations. We checked the tests and the implemented `compress` function against the mathematical description and could not identify any issues.

Code Quality

The code is well-organized and generally adheres to Solidity best practices. The code is generated by a Golang data-driven code generator, which outputs textual data that can then be compiled and deployed to EVM-based chains. However, we found that error handling in the implementation could be improved, as the error messages returned do not provide useful information about the cause of the error. We recommend creating custom error types with the appropriate error messages (Suggestion 1).

Test

Tests are implemented in a branch of the general `gnark` test repository here. The tests are passing and are sufficient. However, since the `compress` function is only implemented in the tests, we recommend improving the accessibility to the `compress` function by adding it to an existing library or implementing it as a `view` function in the smart contract itself (Suggestion 2).

Documentation

The project documentation provided for this review offers a generally sufficient overview of the system and its intended behavior.

Code comments

We found that the code is sufficiently commented. However, the functions present in the codebase do not follow the guidelines for Solidity code comments. Using NatSpec comments would help code reviewers and users easily understand the inputs and functionality of every method present, and therefore aid in identifying any potential issues (Suggestion 3).

Scope

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

Specific Issues & Suggestions

We list the issues and suggestions found during the review, in the order we reported them. 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	STATUS
Suggestion 1: Use Custom Error Messages	Resolved
Suggestion 2: Collaborate With Gnark To Implement the Compress Function	Resolved
Suggestion 3: Adhere to Natspec Guidelines	Resolved

Suggestions

Suggestion 1: Use Custom Error Messages

Location

`https://github.com/Consensys/gnark/pull/810/files#diff-55042c3c4e5c5fd9d9a0d329cb769ef950d087be5e65afd73d81384e309df37d`

Synopsis

The Go template `solidity.go` of the verifier smart contract checks specific values with the solidity function `require`. It does this without returning any error messages. In contrast, the original contract template contained descriptive error messages.

Mitigation

We recommend adding `reverts` or `requires` that return accurate error messages to improve code usage.

Status

The Worldcoin team has added `reverts` with two distinct error messages, `PublicInputNotInField` and `ProofInvalid`, across the code.

Verification

Resolved.

Suggestion 2: Collaborate With Gnark To Implement the Compress Function

Location

`solidity/solidity_groth16_test.go#L162`

`solidity/solidity_groth16_test.go#L176`

Synopsis

At the moment, the `compress` functions for curve points only exist in a test file. These are complex implementations that are difficult to implement.

Mitigation

We recommend collaborating with the maintainers of the `gnark` library to implement the `compress` functions in a way that allows developers to easily call it as part of a library or as a `view` function in the smart contract.

Status

The Worldcoin team has implemented the `compress` functions as a `view` function in the Smart Contract template `soliditiy.go`. The tests have been updated as well.

Verification

Resolved.

Suggestion 3: Adhere to NatSpec Guidelines

Location

`https://github.com/Consensys/gnark/pull/810/files#diff-55042c3c4e5c5fd9d9a0d329cb769ef950d087be5e65afd73d81384e309df37d`

Synopsis

While the code comments are descriptive, it is standard to follow NatSpec guidelines when commenting on function descriptions, inputs, outputs, and any other notable information about the developer usage of the functions.

Mitigation

We recommend strictly adhering to NatSpec guidelines.

Status

The Worldcoin team has implemented the mitigation as recommended.

Verification

Resolved.

About Least Authority

We believe that people have a fundamental right to privacy and that the use of secure solutions enables people to more freely use the Internet and other connected technologies. We provide security consulting services to help others make their solutions more resistant to unauthorized access to data and unintended manipulation of the system. We support teams from the design phase through the production launch and after.

The Least Authority team has skills for reviewing code in multiple Languages, such as C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, JavaScript, ZoKrates, and circom, for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture in cryptocurrency, blockchains, payments, smart contracts, zero-knowledge protocols, and consensus protocols. Additionally, the team can utilize various tools to scan code and networks and build custom tools as necessary.

Least Authority was formed in 2011 to create and further empower freedom-compatible technologies. We moved the company to Berlin in 2016 and continue to expand our efforts. We are an international team that believes we can have a significant impact on the world by being transparent and open about the work we do.

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

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up 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.

Vulnerability Analysis

Our audit techniques include manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's website to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. As we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation. We hypothesize what vulnerabilities may be present and possibly resulting in Issue entries, then for each, we follow the following Issue Investigation and Remediation process.

Documenting Results

We follow a conservative and transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even before having verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this, we analyze the feasibility of an attack in a live system.

Suggested Solutions

We search for immediate and comprehensive mitigations that live deployments can take, and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our Initial Audit Report, and before we perform a verification review.

Before our report, including any details about our findings and the solutions are shared, we like to work with your team to find reasonable outcomes that can be addressed as soon as possible without an overly negative impact on pre-existing plans. Although the handling of issues must be done on a case-by-case basis, we always like to agree on a timeline for a resolution that balances the impact on the users and the needs of your project team.

Resolutions & Publishing

Once the findings are comprehensively addressed, we complete a verification review to assess that the issues and suggestions are sufficiently addressed. When this analysis is completed, we update the report and provide a Final Audit Report that can be published in whole. If there are critical unaddressed issues, we suggest the report not be published and the users and other stakeholders be alerted of the impact. We encourage that all findings be dealt with and the Final Audit Report be shared publicly for the transparency of efforts and the advancement of security learnings within the industry.