RAILGUN_

SMART CONTRACT AUDIT

zokyo

December 21st 2022 | v. 1.0

Security Audit Score

PASS

Zokyo Security has concluded that this smart contract passes security qualifications to be listed on digital asset exchanges.

TECHNICAL SUMMARY

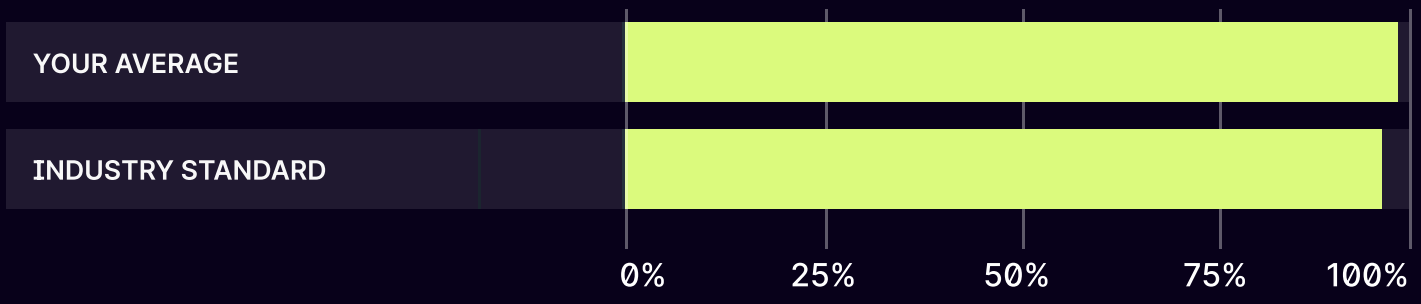
This document outlines the overall security of the Railgun smart contracts evaluated by the Zokyo Security team.

The scope of this audit was to analyze and document the Railgun smart contract codebase for quality, security, and correctness.

Contract Status

There were 0 critical issues found during the audit. (See Complete Analysis

Testable Code



100% of the code is testable, which is above the industry standard of 95%.

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract but rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that can withstand the Ethereum network's fast-paced and rapidly changing environment, we recommend that the Railgun team put in place a bug bounty program to encourage further active analysis of the smart contract.

Table of Contents

Auditing Strategy and Techniques Applied 3

Executive Summary 4

Structure and Organization of the Document 5

Complete Analysis 6

Code Coverage and Test Results for all files written by Zokyo Security 11

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the Railgun repository: https://github.com/Railgun-Privacy/contract/commit

Last commit: 4385ec73bf7d0da283123e8a3ac3900216cfc3f0

Within the scope of this audit, the team of auditors reviewed the following contract(s):

- logic/Commitments.sol
- logic/RailgunLogic.sol
- logic/RailgunSmartWallet.sol

Throughout the review process, care was taken to ensure that contracts:

- Documentation and code comments match logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices inefficient use of gas, without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the latest vulnerabilities;
- Whether the code meets best practices in code readability, etc.

Zokyo Security has followed best practices and industry-standard techniques to verify the implementation of Railgun smart contracts. To do so, the code was reviewed line by line by our smart contract developers, who documented even minor issues as they were discovered. Part of this work includes writing a test suite using the Truffle testing framework. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

01	Due diligence in assessing the overall code quality of the codebase.
02	Cross-comparison with other, similar smart contracts by industry leaders.
03	Testing contracts logic against common and uncommon attack vectors.
04	Thorough manual review of the codebase line by line.

Executive Summary

Zokyo auditing team has run a deep investigation of given code. The contracts are in good condition, well written and structured.

During the auditing process, there were some issues with medium severity and informational issues found. After the technical review of fixes from Railgun team, we can state issues are resolved, unresolved and invalid in the doc accordingly.

All the mentioned findings may have an effect only in case of specific conditions performed by the contract owner and the investors interacting with it.

STRUCTURE AND ORGANIZATION OF THE DOCUMENT

For the ease of navigation, the following sections are arranged from the most to the least critical ones. Issues are tagged as “Resolved” or “Unresolved” or “Invalid” depending on whether they have been fixed or addressed. The issues that are tagged as “Verified” contain unclear or suspicious functionality that either needs explanation from the Client or remains disregarded by the Client. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

Critical

The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.

High

The issue affects the ability of the contract to compile or operate in a significant way.

Medium

The issue affects the ability of the contract to operate in a way that doesn’t significantly hinder its behavior.

Low

The issue has minimal impact on the contract’s ability to operate.

Informational

The issue has no impact on the contract’s ability to operate.

COMPLETE ANALYSIS

Findings Summary

#	Title	Risk	Status
1	Unchecked possible zero address	Medium	Resolved
2	Possible re-entrancy	Medium	Resolved
3	Unnecessary contract name specified	Informational	Unresolved
4	Add information about 4 slots gap	Informational	Invalid
5	Storage slots amount increased	Informational	Invalid

MEDIUM | RESOLVED

Transaction validation is out of the scope

File: RailgunLogic.sol
Function: validateTransaction
Details:
The current implementation of the in-scope functions does not validate unshieldPreimage to contain a correct value. Without such validation, anyone could call a function providing its own address and get all tokens.

Recommendation:
make sure Verifier.verify function is audited and does all necessary verifications and validations."

MEDIUM | RESOLVED

Possible re-entrancy

File: RailgunSmartWallet.sol
Function: transact
Details:
In the case there'll be an ERC20 token with the callback functionality within the transfer method or an ERC721 within the transferFrom method, it's possible to have a re-entrancy attack.

Recommendation:
make sure that the same Transaction could not be sent twice withing one blockchain tx. This could be done, for example, by marking the hash as processed before sending any tokens.

INFORMATIONAL | UNRESOLVED

Unnecessary contract name specified

File: RailgunLogic.sol
Function: transferTokenIn
Details:
While calling the getFee function on the line#258 the third argument is specified as: RailgunLogic.shieldFee. We double-checked the code and there is no any reason of clarifying the contract name for the given state variable

Recommendation:
replace the argument with simple shieldFee

INFORMATIONAL | INVALID

Add information about 4 slots gap

File: RailgunLogic.sol
Details:
Because the contract was moved upper in the storage by 40 slots and 3 more slots were added in the GAP for the currently unused variables (depositFee, withdrawFee, transferFee) it would be nice to add a comment or even put them as the separate variable to make sure none in the newest versions (let's say in a year or two) will use those slots.

Recommendation:
define __gap as the array of 40 elements and add additional variable after to contain the dirty 3 slots

Storage slots amount increased

"File: RailgunLogic.sol
Details:
Decreasing the __gap in the Commitments by 40 and adding the __gap as an array of 43 elements moved the three previously used variable data into the end of the __gap. But then adding 6 additional variables that occupy 5 slots created an overflow in the storage layout. It should be good if you didn't inhere that contract and should not overlap any data.

Recommendation:
Decrease the __gap in the RailgunLogic by 5 elements"

	logic/Commitments.sol logic/RailgunLogic.sol logic/RailgunSmartWallet.sol
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

CODE COVERAGE AND TEST RESULTS FOR ALL FILES

Tests written by Zokyo Security

As a part of our work assisting Railgun in verifying the correctness of their contract code, our team was responsible for writing integration tests using the Truffle testing framework.

The tests were based on the functionality of the code, as well as a review of the Railgun contract requirements for details about issuance amounts and how the system handles these.

Commitments

#doubleInit
✓ fails if twice initializing
#hashLeftRight
✓ hashLeftRight
#getInsertionTreeNumberAndStartingIndex
✓ create new tree
#insertLeaves
✓ check tree number and starting index after inserting leaves
✓ check tree number and starting index after inserting leaves with loop
#changeFee
✓ check event after changing fee
✓ fails if caller is not the owner
✓ fails if shield fee exceeds 50%
✓ fails if unshield fee exceeds 50%
#getFee
✓ check fee
#getTokenID
✓ get ERC20 TokenID
✓ get ERC721 TokenID
#hashCommitment
✓ hashCommitment ERC20
✓ hashCommitment ERC721
#validateCommitmentPreimage
✓ validateCommitmentPreimage ERC20
✓ fails validateCommitmentPreimage ERC20 if npk doesnt exist
✓ return false if token in blocklist
✓ fails validateCommitmentPreimage ERC20 if value equals 0
✓ validateCommitmentPreimage ERC721

#addVector
✓ check vector after adding
✓ fails if caller is not the owner
#removeVector
✓ check vector after removing
✓ fails if caller is not the owner
#sumCommitments
✓ sumCommitments
#accumulateAndNullifyTransactionStub
✓ accumulateAndNullifyTransactionStub
#transferTokenIn
✓ transferTokenIn ERC20
✓ transferTokenIn ERC721
✓ transferTokenIn ERC1155
#transferTokenOut
✓ transferTokenOut ERC20
✓ transferTokenOut ERC721
✓ transferTokenOut ERC1155
#validateTransaction
✓ validateTransaction when UnshieldType is NONE
✓ validateTransaction when UnshieldType is NORMAL
✓ fails with error 8
✓ fails with error 7
✓ fails with error 6
✓ validateTransaction with nullfiller
✓ fails with error 4
✓ fails with error 1
✓ fails with error 2
✓ fails with error 3

RailgunSmartWallet
✓ check lastEventBlock equals transaction blockNumber after transact NORMAL
✓ fails if transaction isn't valid
#shield
✓ success shield ERC-20
✓ Invalid preimage data
#transact
✓ check lastEventBlock equals transaction blockNumber after transact ERC-20

48 passing (31s)

The resulting code coverage (i.e., the ratio of tests-to-code) is as follows:

FILE	% STMTS	% BRANCH	% FUNCS	% LINES	% Uncovered Lines
Commitments.sol	100	100	100	100	
Globals.sol	100	100	100	100	
RailgunLogic.sol	100	100	100	100	
RailgunSmartWallet.sol	100	100	100	100	
All Files	**100**	**100**	**100**	**100**	

We are grateful for the opportunity to work with the Railgun team.

The statements made in this document should not be interpreted as an investment or legal advice, nor should its authors be held accountable for the decisions made based on them.

Zokyo Security recommends the Railgun team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo