RAILGUN_

RAILGUN

SMART CONTRACT AUDIT

zokyo

September 14th, 2022 | v. 1.0

Security Audit Score

PASS

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

TECHNICAL SUMMARY

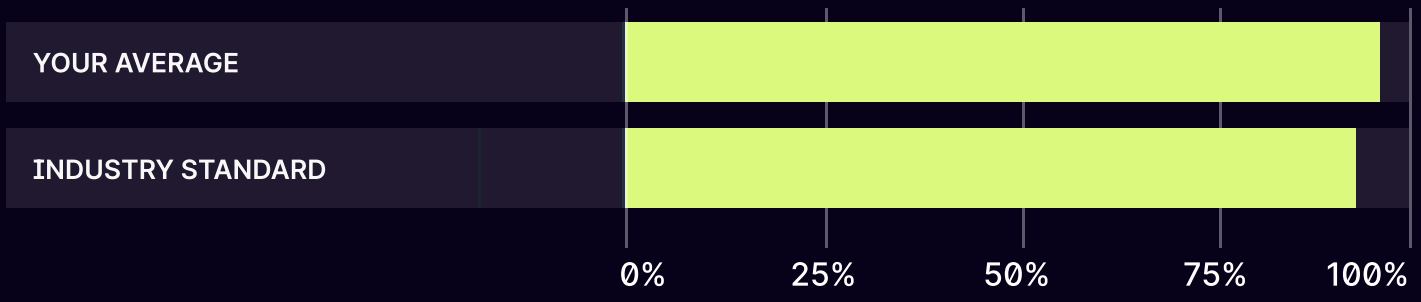
This document outlines the overall security of the RAILGUN smart contracts, evaluated by Zokyo's Blockchain Security team.

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

Contract Status

There was 0 critical issue found during the audit. (See Complete Analysis)

Testable Code



The testable code is 100%, which is above the industry standard of 95%. (See Complete Analysis)

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

Table of Contents

AUDITING STRATEGY AND TECHNIQUES APPLIED

The Smart contract's source code was taken from the RAILGUN repository.

Repository: https://github.com/Railgun-Privacy/contract

Last commit: d73c1da62c624fe083417342ce3e64748572bde9

Contracts under the scope:

- Voting.sol

Throughout the review process, Zokyo Security ensures that the contract:

- Implements and adheres to existing Token standards appropriately and effectively;
- 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's Security Team has followed best practices and industry-standard techniques to verify the implementation of RAILGUN smart contracts. To do so, the code is reviewed line-by-line by our smart contract developers, documenting any issues as they are discovered. Part of this work includes writing a unit 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 contract 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 RAILGUN's smart contract. The contract is in good condition, well written and structured.

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

Based on the conducted audit, we give a score of 99 to the aforementioned contracts.

STRUCTURE AND ORGANIZATION OF DOCUMENT

For ease of navigation, sections are arranged from most critical to least critical. Issues are tagged “Resolved” or “Unresolved” depending on whether they have been fixed or addressed. Acknowledged means that the issue was sent to the client team and the client team are aware of it, but they have chosen to not solved it. 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

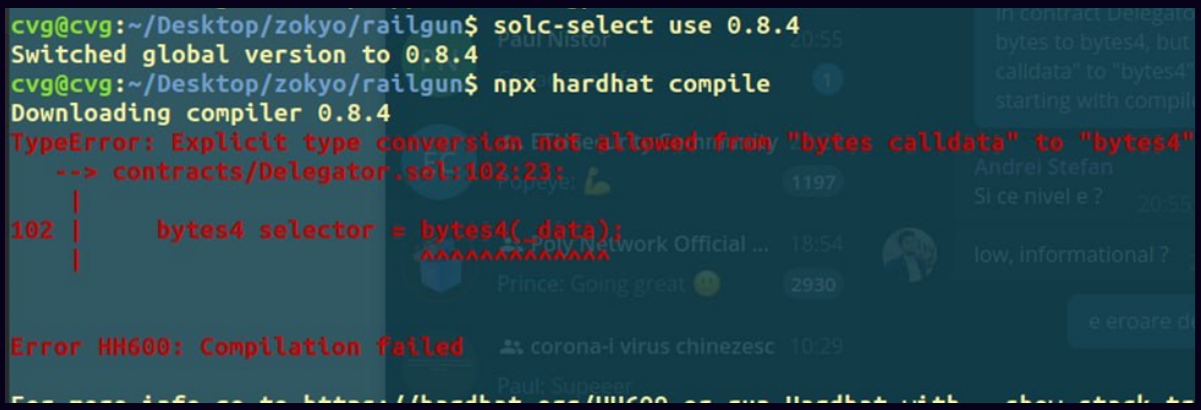
The issue has no impact on the contract’s ability to operate.

COMPLETE ANALYSIS

LOW | RESOLVED

The solidity pragma version is not locked for both Voting.sol and Delegator.sol which is imported in the Voting.sol contract, in the hardhat config file right now the version selected is 0.8.12, but by having a pragma version that is not locked this can be prone to errors, an example of such an error is that in the moment the contracts get compiled with a version lower then 0.8.7, there will be a compilation error because of conversion from bytes to bytes4 at line 102 in Delegator.sol

Recommendation:
Lock the pragma version to 0.8.12



LOW | ACKNOWLEDGED

In contract Voting.sol, there are two for loops functions, one at line 207 and one at line 408, it is recommended that inside the loop functions to implement a break condition that will check the gasLeft() and will break the function iteration if the gas is lesser then 20 000 units to not run into the out of gas error, in the case of an “out of gas error”, the gas it is not returned but in a case of reverting from an require or revert function or from execution breaking the gas it is returned.

Recommendation:
Implement a check inside the loops that will check for the gas left and will intrerupt the execution of the function using the break keyword to not be able to run in the “out of gas error”

INFORMATIONAL | ACKNOWLEDGED

In contract Voting.sol, at line 102, variable STAKING_CONTRACT is of type Staking, and the variable it is used inside the contract to do external calls to a certain address at lines 204, 250, 320, 352. To better optimize the gas usage, it is recommended that STAKING_CONTACT to be of type address, and do an in-place initialization using an interface where you want to do an external call, this will greatly reduce the gas costs, because you will only read from the storage the address instead of an object of type Staking which is greatly more expensive.

Recommendation:
Change the type of STAKING_CONTRACT to address and do an in-place initialization using the interface where you are doing external calls at lines 204, 250, 320 and 352.

INFORMATIONAL | ACKNOWLEDGED

In contract Voting.sol, at line 103, variable DELEGATOR_CONTRACT is of type Delegator, and the variable it is used inside the contract to do external calls to a certain address at line 407 inside a for loop. To better optimize the gas usage, it is recommended that DELEGATOR_CONTRACT to be of type address, and do an in-place initialization using an interface where you want to do an external call, this will greatly reduce the gas costs, because you will only read from the storage the address instead of an object of type Staking which is greatly more expensive.

Recommendation:
Change the type of DELEGATOR_CONTRACT to address and do an in-place initialization using the interface at line 407

INFORMATIONAL | RESOLVED

In contract Voting.sol, at line 60, the struct ProposalStruct it is defined, to be in accordance with best practices and to optimize the gas usage it would be recommended to implement variable packing inside structures definitions.

Recommendation:
Refactor the structure so it will be in accordance with the variable packing patter.

INFORMATIONAL | ACKNOWLEDGED

In contract Voting.sol, at line 207, there is a for loop that could be better optimized by changing the i++ inside the for loop to an unchecked arithmetic because the i variable is of type uint256, and you will run out of gas before being able to iterate enough for an overflow

Recommendation:
Change the structure of the i++ instruction to an unchecked one to better optimize the gas.

```
205
206     // Loop over actions and copy manually as solidity doesn't support copying structs
207     for (uint256 i = 0; i < _actions⬆.length;) {
208       proposal.actions.push(Call(
209         _actions⬆[i].callContract,
210         _actions⬆[i].data,
211         _actions⬆[i].value
212       ));
213       unchecked {
214         ++i;
215       }
216     }
```

INFORMATIONAL | ACKNOWLEDGED

In contract Voting.sol, at line 408, there is a for loop that could be better optimized by changing the i++ inside the for loop to an unchecked arithemthic because the i variable is of type uint256 and you will run out of gas before being able to iterate enough for an overflow.

Recommendation:
Change the structure of the i++ instruction to an unchecked one to better optimize the gas.

```
404
405   Call[] memory actions = proposal.actions; //todo: put actions variable in-memory to save gas in the for loop
406
407   // Loop over actions and execute
408   for (uint256 i = 0; i < actions.length;) {
409     // Execute action
410     (bool successful, bytes memory returnData) = IDelegator(DELEGATOR_CONTRACT).callContract(
411       actions[i].callContract,
412       actions[i].data,
413       actions[i].value
414     );
415
416     // If an action fails to execute, catch and bubble up reason with revert
417     if (!successful) {
418       revert ExecutionFailed(i, returnData);
419     }
420     unchecked {
421       ++i;
422     }
423   }
```

(Note: this img also represents multiple changes from recommendations about this lines of code that are place in the document above for better gas optimization)

INFORMATIONAL | RESOLVED

In contract Voting.sol, at line 405, there is a for loop that could be better optimized by changing the the variable actions from storage to an in memory variable, because the gets used in the for loop and at every use the value will be read again from the storage, which can become expensive after a lot of operations.

Recommendation:
Change the variable actions form in storage to an in memory variable. (The modification can be observed in the image from above)

	Voting.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 are written by the RAILGUN DAO Project team

Governance/Voting

- ✓ Should go through vote lifecycle correctly (1090ms)
- ✓ Should not be able to sponsor after the sponsor window (85ms)
- ✓ Should not not execute failed proposal (265ms)
- ✓ Should execute proposals correctly (240ms)
- ✓ Should only be able to sponsor once per week (189ms)
- ✓ Should only allow voting key to call (134ms)

6 passing (7s)

Code Coverage

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

FILE	% STMTS	% BRANCH	% FUNCS	% LINES	% UNCOVERED LINES
governance/	92.31	83.87	69.23	93.15	
voting.sol	92.31	83.87	69.23	93.15	... 155,167,415
All files	**92.31**	**83.87**	**69.23**	**93.15**	

CODE COVERAGE AND TEST RESULTS FOR ALL FILES

Tests are written by Zokyo Secured team

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

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.

Voting

- ✓ Should be deployed correctly (134ms)
- ✓ Should be able to create a proposal (482ms)
- ✓ Should be able to get number of proposals (180ms)
- ✓ Should be able to get actions in a proposal (182ms)
- ✓ Should allow voting key to be set (518ms)
- ✓ Should allow sponsorship of proposals (744ms)
- ✓ Should be able to get sponsored amount given by account (179ms)
- ✓ Should allow proposals to be unsponsored (498ms)
- ✓ Should allow votes to be called (429ms)
- ✓ Should allow nay votes on proposal (639ms)
- ✓ Should allow yes votes on proposals (359ms)
- ✓ Should be able to get number of votes cast by voter (388ms)
- ✓ Should allow proposals to be executed (828ms)
- ✓ Should not allow proposals to be executed with wrong variables (882ms)

14 passing (13s)

Code Coverage

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

FILE	% STMTS	% BRANCH	% FUNCS	% LINES	% UNCOVERED LINES
governance/	100	100	100	100	
Voting.sol	100	100	100	100	
All files	**100**	**100**	**100**	**100**	

We are grateful to have been given the opportunity to work with the RAILGUN DAO Project 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's Security Team recommends that the RAILGUN DAO Project team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo