RAILGUN_

SMART CONTRACT AUDIT

zokyo

March 9th 2023 | 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

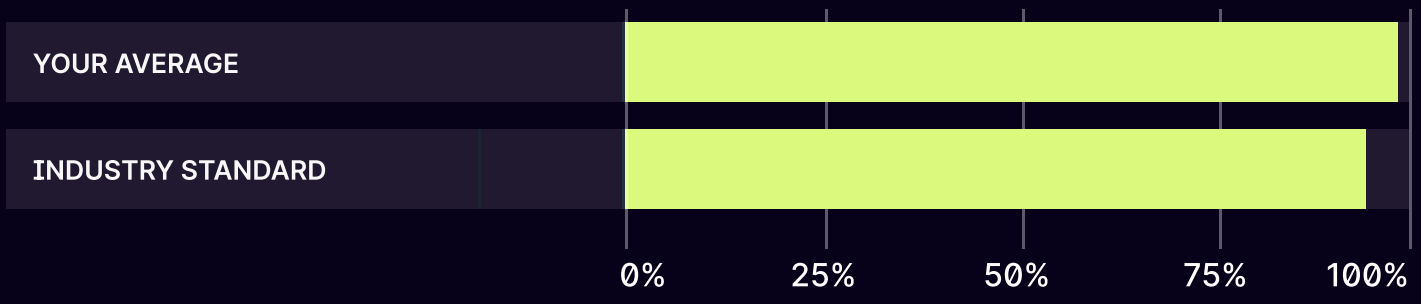
This document outlines the overall security of the RailGun smart contracts evaluated by the Zokyo Security team.

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

Contract Status

There was no critical issue found during the audit. (See Complete Analysis)

Testable Code



96.43% 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 contracts 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 contracts.

Table of Contents

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the RailGun repository:
https://github.com/railgun-privacy/contract

Last commit: 5076fe437c8ae6d481a5694a4e1c143e2983cea8

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

- RailgunLogic.sol
- RailgunSmartWallet.sol

During the audit, Zokyo Security ensured that the contract:

- Implements and adheres to the existing standards appropriately and effectively;
- The documentation and code comments match the logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices, efficiently using resources without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the most resent vulnerabilities;
- 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 Hardhat 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.	03	Testing contracts logic against common and uncommon attack vectors.
02	Cross-comparison with other, similar smart contracts by industry leaders.	04	Thorough manual review of the codebase line by line.

Executive Summary

There was no critical issue found during the audit, alongside three with low severity and some of informational issues. 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. They are described in detail in the “Complete Analysis” section.

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 “Acknowledged” depending on whether they have been fixed or addressed. Acknowledged means that the issue was sent to the RailGun team and the RailGun team is aware of it, but they have chosen to not solved it. 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	Nullifier checks not executed when _transaction.nullifiers is empty	Low	Resolved
2	Payable Functions in RailgunSmartWallet Contract Do Not Use Ether and Have No Extraction Method	Low	Resolved
3	Adapt Contract Not Verified as Contract	Low	Resolved
4	Inaccurate Error Message in RailgunLogic Contract for Unsupported Token Types	Informational	Acknowledged
5	Solidity Values Are Automatically Initialized to `0`	Informational	Acknowledged
6	Fix solidity version	Informational	Acknowledged
7	Save gas for increments in the loop	Informational	Acknowledged
8	Error objects to save gas	Informational	Acknowledged

LOW | RESOLVED

Nullifier checks not executed when _transaction.nullifiers is empty

Contract: RailgunLogic.sol
Line(s): 491

The accumulateAndNullifyTransaction function in the RailgunLogic contract allows for nullifier checks to be skipped when the _transaction.nullifiers array is empty. This can occur when a user provides an empty array as input to the function, causing the for loop to never execute.

Impact: If the nullifier checks are not executed, it could allow for the double spending of notes, resulting in loss of funds for users of the system.

Here's an example POC in Hardhat and Ethers that demonstrates the difference in flag and secondFlag values in the loopThroughArrays function of the EmptyArrayTest contract. This POC shows that if an empty array is looped through, the loop will not execute, and the value of flag will remain 0. On the other hand, if a non-empty array is looped through, the loop will execute, and the value of secondFlag will equal the length of the array (which is 3 in this case).

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract EmptyArrayTest {
    uint256 public flag;
    uint256 public secondFlag;

    function loopThroughArrays() public {
        uint256[] memory emptyArray;
        uint256[3] memory myArray;

        for (uint256 i = 0; i < emptyArray.length; i++) {
            flag++;
        }

        for (uint256 i = 0; i < myArray.length; i++) {
            secondFlag++;
        }
    }
}
```

```
// POC to demonstrate the difference in flag and secondFlag values in the EmptyArrayTest contract
const { ethers } = require("hardhat");

async function main() {
  const EmptyArrayTest = await ethers.getContractFactory("EmptyArrayTest");
  const emptyArrayTest = await EmptyArrayTest.deploy();
  await emptyArrayTest.deployed();

  console.log(`flag value before calling loopThroughArrays: ${await emptyArrayTest.flag()}`);
  console.log(`secondFlag value before calling loopThroughArrays: ${await emptyArrayTest.secondFlag()}`);

  await emptyArrayTest.loopThroughArrays();

  console.log(`flag value after calling loopThroughArrays: ${await emptyArrayTest.flag()}`);
  console.log(`secondFlag value after calling loopThroughArrays: ${await emptyArrayTest.secondFlag()}`);
}

main();
```

When we run this script, we can see that the value of flag is `0` after calling loopThroughArrays, and the value of secondFlag is 3.

```
flag value before calling loopThroughArrays: 0
secondFlag value before calling loopThroughArrays: 0
flag value after calling loopThroughArrays: 0
secondFlag value after calling loopThroughArrays: 3
```

This demonstrates that if we try to loop through an empty array in Solidity, the loop contents will not execute. If we loop through a non-empty array, the loop contents will execute for each element in the array.

Recommendation:
A fix for this issue could involve adding an additional check before the for loop to ensure that the _transaction.nullifiers array is not empty.

Note:
Responding from feedback from the Railgun team: we can confirm that the call to validateTransaction() and accumulateAndNullifyTransaction() are performed on the same transaction object and that nullifiers are bound to the SNARK proof, adding, removing, or modifying nullifier values will cause the SNARK proof to fail.

Additional Comments from the Railgun team: An addition to issue 1, an internal review of it has found that while the nullifiers length is enforced through the SNARK, there is a possibility where the commitment array length enforced by the SNARK can be 1, and if that is marked as a withdraw note the final commitment array length is `0`. This results in a situation where the commitment event is emitted incorrectly causing issues for front-ends.

LOW | RESOLVED

Payable Functions in RailgunSmartWallet Contract Do Not Use Ether and Have No Extraction Method

Contract: RailgunSmartWallet.sol
Line(s): 21, 68

The RailgunSmartWallet contract includes two functions, shield and transact, that are payable. This means that they can accept payments in any cryptocurrency supported by the contract. However, these functions do not use Ether, the native cryptocurrency of the Ethereum network. Moreover, there is no way to extract Ether if a user mistakenly sends it to these functions.

Impact: This bug has no impact on the security of the contract or the Ethereum network, as the contract does not support Ether and there is no way for the Ether to be used or extracted. However, users who mistakenly send Ether to the contract may lose their funds permanently.

Recommendation:
In summary, while the payable modifier can be useful in certain cases, it is important to consider whether it is necessary and to carefully design the contract to handle ether properly if it is used. However, if the team intends to future-proof the code by making it payable from the beginning, it could also be a useful strategy to prevent significant refactoring later on.

Comments from Railgun team: No functions should have been marked payable. Payable modifiers are removed.

Adapt Contract Not Verified as Contract

Contract: RailgunLogic.sol
Line(s): 427

The Railgun smart contract includes a feature that allows the user to specify an "Adapt Contract" to execute additional behavior when certain transactions are executed. However, the current implementation does not include a check to verify that the Adapt Contract is actually a contract. This means that if an external account were to specify an invalid Adapt Contract that is not a contract, the smart contract would not throw an error and would simply execute the transaction as normal, potentially leading to unexpected behavior. Recommendation: To prevent this vulnerability, it is recommended to add a check to verify that the Adapt Contract is a contract before executing any behavior associated with that contract.

Recommendation:
One possible solution would be to add a check for the code size of the Adapt Contract, which should be greater than zero if it is a valid contract. The following code could be added to the validateTransaction function:

```
if (_transaction.boundParams.adaptContract != address(0) && !isContract(_transaction.boundParams.adaptContract))
return (false, "Invalid Adapt Contract as Sender");
```

And the following helper function could be defined in the contract:

```
function isContract(address addr) internal view returns (bool) {
  uint size;
  assembly { size := extcodesize(addr) }
  return size > 0;
}
```

Comments from Railgun team: The adapt contract field can also be used to lock transactions to a specific EOA, this is allowed behavior, though the most common use case is to lock it to contracts.

INFORMATIONAL | ACKNOWLEDGED

Inaccurate Error Message in RailgunLogic Contract for Unsupported Token Types

Contract: RailgunLogic.sol
Line(s): 304

In the current implementation of the RailgunLogic contract, the error message for an unsupported token type is hardcoded to "RailgunLogic: ERC1155 not yet supported". This could be problematic as a user could input an invalid token address that is not an ERC1155 token, resulting in the same error message being returned.

Recommendation:
To avoid confusion for users, the error message could be made more general, such as "Token type not supported", to indicate that the issue could be caused by an invalid token address or an unsupported token type.

INFORMATIONAL | ACKNOWLEDGED

Solidity Values Are Automatically Initialized to `0` Description

In Solidity, variables of value types (such as integers, booleans, and addresses) are automatically initialized to `0` when declared. This means that you do not have to explicitly assign the value of `0` to a variable when declaring it.

Recommendation:
However, it is good practice to initialize all variables in your Solidity code, even if they are automatically initialized to `0`. This can help make your code more readable and avoid potential bugs. This finding is not a bug, but rather a feature of the language. It is good to be aware of this behavior in order to write efficient and readable Solidity code.

INFORMATIONAL | ACKNOWLEDGED

Fix solidity version

Lock the pragma to a specific version since not all the EVM compiler versions support all the features, especially the latest one, which are kind of beta version, So the intended behavior written in the code might not be executed as expected. Locking the pragma helps ensure that contracts do not accidentally get deployed using, for example, the latest compiler, which may have higher risks of undiscovered bugs.

Recommendation:
fix version to 0.8.7

INFORMATIONAL | ACKNOWLEDGED

Save gas for increments in the loop

The increments in for loops at lines 29, 77 and 98 in RailgunSmartWallet and lines 402,491 and 514 in **RailgunLogic** can be updated to save gas as follows:

```
for (uint i = 0; i < len; ) {
    ...

    unchecked {
        ++i;
    }
}
```

Recommendation:
Update the code in contracts for loop increments as shown above.

Comments from Railgun team: Gas savings are not large enough to justify reduced code readability, leaving as is.

INFORMATIONAL | ACKNOWLEDGED

Error objects to save gas

Starting from solidity 0.8.4 (project uses 0.8.17) - solidity developers are recommended to use custom error objects to save gas and also better error information as explained here in solidity blog.

Recommendation:
Use custom errors instead of require statements to revert.

Comments from Railgun team: Custom errors would be ideal, but current frontends/SDKs depend on string errors, as the Railgun contract is usually not the top-level call. Switching to custom errors will be done as part of an overall effort to reduce bytecode size, but isn't practical as of right now.

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

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

Contract: RailGunLogic

init
✓ Should have correct intial values
getFee()
✓ Should return fee and base
transferTokenIn()
✓ Should not transfer in ERC1155 notes (68ms)
✓ Should transfer in ERC20 notes (71ms)
✓ It should not unsheild ERC1155
transferTokenOUT()
✓ Should sheild ERC20 notes (102ms)
✓ Should unsheild ERC721 and ERC20 notes (66ms)
validateCommitmentPreimage()
✓ Should not validate an invalid NFT note value (57ms)
✓ Shold return Invalid Note Value for 0 value tokens
✓ Should return Invalid Note NPK (38ms)
✓ Should not validate Unsupported Token (120ms)
getTokenId()
✓ getTokenId
checkSafetyVectors()
✓ Should revert
✓ Should revert
addVector()
✓ adds a Vector
✓ removes a vector
hashCommitment()
✓ It should hash commitments (130ms)
accumulateAndNullifyTransaction()
✓ Should accumulate and nullify transactions (1783ms)

```
    sumCommitments()
    ✓ Should sum commitments in a transaction (911ms)
    nonTransfering
    ✓ Should not transfer non transfering note (156ms)
    changeFee()
    ✓ Should change fees and only by owner (67ms)
    changeTreasury()
    ✓ Should change treasury only by owner
    validateTransaction()
    ✓ Should validate transactions (5661ms)
  contract: RailgunSmartWallet
    Sheild()
    ✓ Sheild (586ms)
    Transact()
    ✓ transact (1858ms)
    ✓ Should reject invalid transactions (1846ms)
    ✓ Should revert if note value is invalid (58ms)

    27 passing (39s)
```

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

FILE	% STMTS	% BRANCH	% FUNCS	% LINES	% UNCOVERED LINES
RailgunLogic.sol	98.82	85.71	100	99.02	
RailgunSmartWallet.sol	95.45	83.33	100	96.43	
FILE	**96.43**	**86.54**	**100**	**99.98**	

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