QuillAudits

Audit Report
August, 2023

For

ritestream

Table of Content

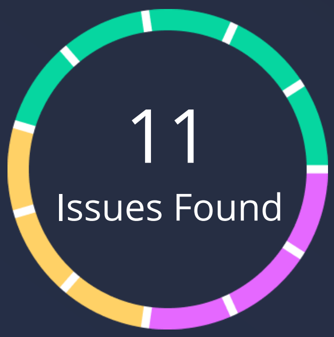
Table of Content

Executive Summary

Project Name	Ritestream
Project URL	*https://www.ritestream.io/*
Overview	A staking contract is the the contract to allow subscribers to claim the rewards. The operator would be calling the stake() function with the amount user used for subscription, the user can then claim the rewards allowed. The reward is 50% of the staked amount.
Audit Scope	*https://github.com/ritestream/ritestream-contract/blob/master/contracts/Staking.sol*
Contracts in Scope	Staking.sol
Commit Hash	627fda5586ed474ac917be9b8da2c8cd62fcea18
Language	Solidity
Blockchain	BSC
Method	Manual Analysis, Functional testing, Automated Testing
Review 1	22 August 2023 - 24 August 2023
Updated Code Received	28 August 2023
Review 2	28 August 2023 - 29 August 2023
Fixed In	*https://github.com/ritestream/ritestream-contract/commit/b19822f6bf074468bd64e837ee0f37941b29749c*

Executive Summary

Number of Security Issues per Severity



🟥 High 🟨 Medium

🟩 Low 🟪 Informational

	High	Medium	Low	Informational
Open Issues	0	0	0	0
Acknowledged Issues	0	1	4	1
Partially Resolved Issues	0	0	0	0
Resolved Issues	0	2	1	2

Types of Severity

Every issue in this report has been assigned to a severity level. There are four levels of severity, and each of them has been explained below.

High Severity Issues

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.

Medium Severity Issues

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

Low Severity Issues

Low-level severity issues can cause minor impact and are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

Informational

These are four severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Types of Issues

Open
Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved
These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged
Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved
Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

Checked Vulnerabilities

- ✓ Access Management
- ✓ Arbitrary write to storage
- ✓ Centralization of control
- ✓ Ether theft
- ✓ Improper or missing events
- ✓ Logical issues and flaws
- ✓ Arithmetic Computations
- ✓ Race conditions/front running
- ✓ SWC Registry
- ✓ Re-entrancy
- ✓ Timestamp Dependence
- ✓ Tautology or contradiction
- ✓ Return values of low-level calls
- ✓ Missing Zero Address Validation
- ✓ Private modifier
- ✓ Revert/require functions
- ✓ Gas Limit and Loops
- ✓ Exception Disorder
- ✓ Gasless Send
- ✓ Use of tx.origin
- ✓ Malicious libraries
- ✓ Compiler version not fixed
- ✓ Address hardcoded
- ✓ Divide before multiply
- ✓ Integer overflow/underflow
- ✓ ERC's conformance
- ✓ Dangerous strict equalities
- ✓ Multiple Sends
- ✓ Using suicide
- ✓ Using delegatecall
- ✓ Upgradeable safety
- ✓ Using throw

Checked Vulnerabilities

- ✓ Re-entrancy
- ✓ Timestamp Dependence
- ✓ Tautology or contradiction
- ✓ Return values of low-level calls

Techniques and Methods

Throughout the audit of smart contracts, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behaviour.
- Token distribution and calculations are as per the intended behaviour mentioned in the whitepaper.
- Implementation of ERC standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods and tools were used to review all the smart contracts.

Structural Analysis

In this step, we have analysed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

A static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms used for Audit

Remix IDE, Truffle,Solhint, Mythril, Slither, Solidity statistic analysis.

A. Contract - Staking.sol

High Severity Issues

No issues found

Medium Severity Issues

A.1 Incorrect logic in stake function

Description

It is required that the stake() function should check that the staked _amount when added in totalStaked should not exceed the MAX_CAP. The stake() contains if else block to ensure this condition.

the issue lies in the way the function handles the calculation of a user's stake amount with a conditional statement. It checks if totalStaked + _amount <= MAX_CAP and then assigned it to the original amount else it deducts the totalStaked from the MAX_CAP.

The totalStaked was already updated with the _amount (totalStaked += _amount; on L91) and still adds _amount to the already updated totalStaked in the if (totalStaked + _amount <= MAX_CAP). Another issue can happen when else executes while subtracting the totalStaked from MAX_CAP.

Example

The operator stakes _amount as 100000e18
And totalSupply is 4900000e18
So before if else block, the _amount would be added to totalStaked.
now totalStaked would be 5000000e18 (equal to MAX_CAP)

So in if condition it will check (5000000e18 + 100000e18)<= MAX_CAP
so for checking this the _amount (100000e18) is getting added again to the totalStaked (5000000e18) in which _amount was already added before the if statement.

Here if condition will fail , In else condition it tries to subtract totalStaked from MAX_CAP but here the the output of subtraction would be 0 because MAX_CAP is 5000000e18 and totalStaked is also 5000000e18 (because 100000e18 _amount was already added to the totalStaked before if block)

There can be multiple conditions depending on the values of totalStaked and _amount but the required logic can be fulfilled by using require statement as the required logic is to check that the total amount staked should be always less than or equal to the MAX_CAP.

A.1 Incorrect logic in stake function

Remediation

require(totalStaked + _amount <= MAX_CAP,"Max cap exceeded"); can be added in stake() and the if else can be removed.

The updated code would look like this:

```
function stake(
        uint256 _amount,
        address _address,
        string memory _month
    ) external onlyOperator {
        require(_amount > 0, "Staking: amount is zero");
        require(totalStaked + _amount <= MAX_CAP,"Max cap exceeded");
        Stake[] memory userStakes = stakes[_address];
        for (uint i = 0; i < userStakes.length; i++) {
            require(
                keccak256(bytes(userStakes[i].month)) !=
                    keccak256(bytes(_month)),
                "Staking: already staked for this month"
            );
        }
        totalStaked += _amount;
        stakes[_address].push(
            Stake(
                _amount,
                block.timestamp,
                 block.timestamp + duration,
                _month
            )
        );

        emit Staked(_address, _amount, block.timestamp, _month);
    }
```

Status

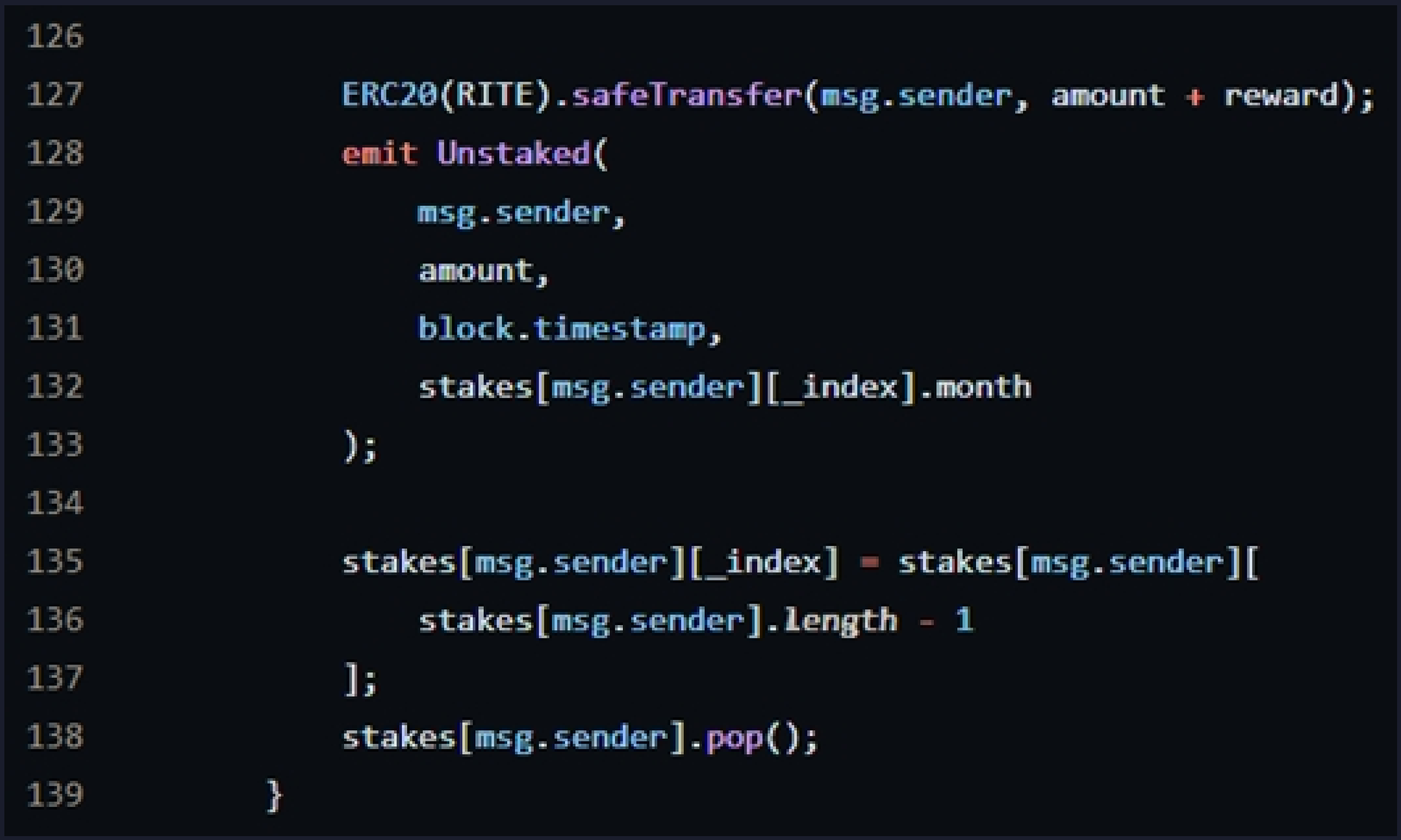
Resolved

A.2 Possible reentrancy for ERC20 token with hooks

Line

127 - 128

Function



```
126
127             ERC20(RITE).safeTransfer(msg.sender, amount + reward);
128             emit Unstaked(
129                 msg.sender,
130                 amount,
131                 block.timestamp,
132                 stakes[msg.sender][_index].month
133             );
134
135             stakes[msg.sender][_index] = stakes[msg.sender][
136                 stakes[msg.sender].length - 1
137             ];
138             stakes[msg.sender].pop();
139         }
```

Description

If the RITE token uses hooks/callback functions which calls specific method on the token receiver to check that the receiver supports the type of token that is getting sent to it e.g ERC777 compatible token then in case the receiver is the smart contract address the attack can be perfomed by having function on the contract to call the unstake() in staking contract again where the Stake element still would be there and the attacker can get more reward amount than what he is eligible for by calling function more than one times for a same month/index.

Here, the problem happens because the code to pop the element on the index from which user is unstaking is getting executed after the transfer. so that while transferring the token the reentering is possible.

A.2 Possible reentrancy for ERC20 token with hooks

Remediation

Follow check effects interaction pattern in unstake.
use the code to pop the element from stakes before the transferFrom call. Before poping element the stakes[msg.sender][_index].month can be stored in temporary variable so that it can be used later while emitting Unstaked event for month parameter.

Additionally it is a good idea to not use tokens with hooks/callbacks incase that functionality is not needed.

Status

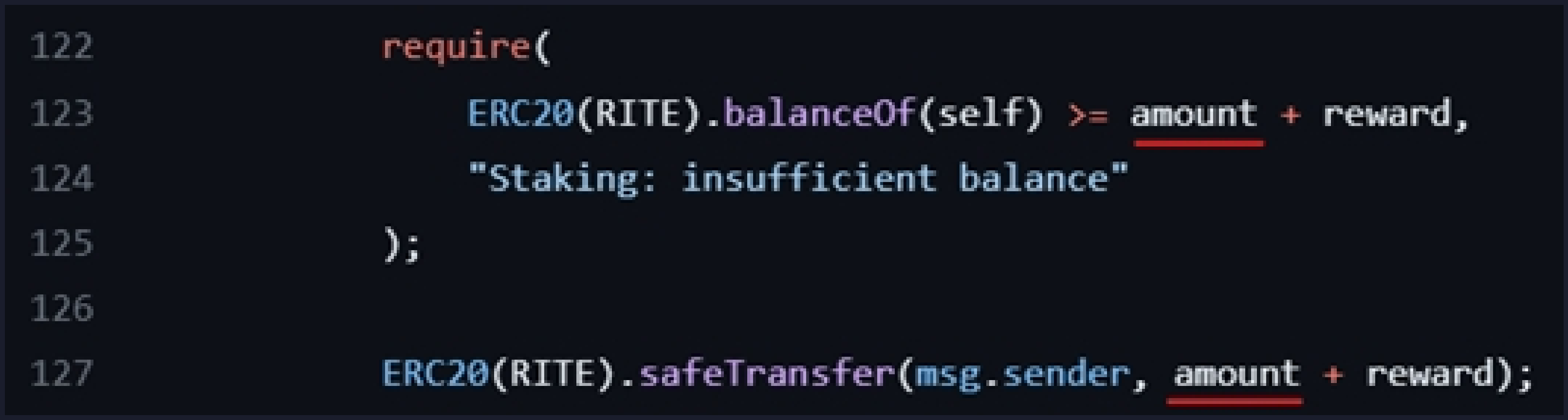
Resolved

A.3 Unstake() transfers amount + reward while transferring reward

Line

123 - 127

Function



```
122            require(
123                ERC20(RITE).balanceOf(self) >= amount + reward,
124                "Staking: insufficient balance"
125            );
126
127            ERC20(RITE).safeTransfer(msg.sender, amount + reward);
```

Description

In unstake() the intentional logic for unstaking is to calculate 50% of the amount staked amount and transfer it to the user. But while transferring it transfers the amount + reward. Here amount wasn't actually staked by the operator (i.e it wasn't transferred to the staking contract while staking). But while sending it is getting sent. which will create problem as the users will get more amount. i.e reward and the amount also (which was never transferred while staking).

A.3 Unstake() transfers amount + reward while transferring reward

Remediation

Verify the logic and remove the amount from L123 and L127.

Status

Resolved

Low Severity Issues

A.4 copying and poping elements can create a confusion while unstaking with index

Description

E.g Operator stakes 5 times for 5 months for specific user for some stakes the end date is reached so for them users can now unstake.

From stakes of 5 months (08/2023, 09/2023, 10/2023, 11/2023, 12/2023) User decides to unstake for the first one i.e 08/2023 So the code will copy element of the month 12/2023 (last element) to the first (i.e 0th element) and will pop the last one which would be duplicate of 12/2023

so after unstaking the array element's sequence will be changed because of copy and pop operation so the index of the elements for month is also changed.

So before unstaking it was in this sequence:
(08/2023, 09/2023, 10/2023, 11/2023, 12/2023) and
after unstaking it is in this sequence:
(12/2023, 09/2023, 10/2023, 11/2023) where the index of element is changed.

In this case, the user can get confused if he assumed that the stake of any month is going to be at the specific index because that got staked (pushed to array) in the same order of increasing month.

A.4 copying and poping elements can create a confusion while unstaking with index

Remediation

This can be handled by fetching the array of stakes using getStakes() so user would be able to know which index needs to be enterd for which month. For backend, managing the unstake function calls would also need to do this while deciding which index to enter.

Status

Acknowledged

A.5 Centralization

Description

Operator can call stake() here in this case if the user fully trusts the operator that after subscribing the operator will going to stake then it may happen that a malicious operator may not stake with that user address even after subscribing.

Remediation

Make sure the operator is the trusted address and validates the things before staking behalf of the user.

Status

Acknowledged

A.6 Transfer the reward amount in the contructor

Description

The total reward that the contract is going to distribute is fixed because the totalStaked plus the _amount that user will add, will be kept below MAX_CAP and the amount of reward that user receives is 50% of what is getting staked for it's address.

That means if the overall amount that is going to be staked is less than or equal to the MAX_CAP then 50% can be calculated of MAX_CAP and it can be transferred in the constructor to improve transparency. After that if required then additional amount can be transferred.

A.6 Transfer the reward amount in the contructor

Remediation

ERC20’s transferFrom() can be used in the constructor to transfer the reward token amount from deployer to the contract.

Status

Acknowledged

A.7 Lack of support for fee on transfer tokens

Description

In case the RITE token that is getting used in this contract deducts fee on transfer then it can create some accounting problems. Eg. While transferring the reward amount to this smart contract it can happen that the reward amount got transferred after deducting the tax/fee so in that case the contract will receive the less amount.

Remediation

Make sure to not use fee on transfer tokens. If the RITE token is fee on transfer token then care should be taken while transferring the reward amount to the contract so that it can be distributed to eligible subscribers and the contract will have enough amount.

Status

Acknowledged

A.8: The usage of loop can be replaced with counter

Description

The stake() has the loop to ensure the the month entered is not already used for staking. After a certain limit the transaction for stake() can go out of gas while iterating through userStakes.
It needs to be checked how much times the loop can iterate without going out of gas.
Or instead of using the loop to check if the month is used before for staking or not, the counter can be used, so for every address there would be a different counter. So while staking the counter number should be entered and that counter for the address should be more than what it was before. It will ensure that the staking is not happening for the already used month for that user.

A.8: The usage of loop can be replaced with counter

Remediation

Ensure and test that the loop is going to execute the predicted amount of time i.e the staking will happen limited amount of time for a specific user so that there won't be risk of transaction reverting with out of gas error.
On the other hand, this issue can be fixed by using the counter as mentioned in the above description.

Status

Resolved

Informational Issues

A.9 MAX_CAP can be declared as constant

Description

MAX_CAP is not getting reinitialized after first initialization. So it can be declared as constant variable.

Remediation

Make MAX_CAP a constant variable.

Status

Resolved

A.10 Unused import

Description

import "./Token.sol" is not getting used and should be removed.

Remediation

Remove the unused import.

Status

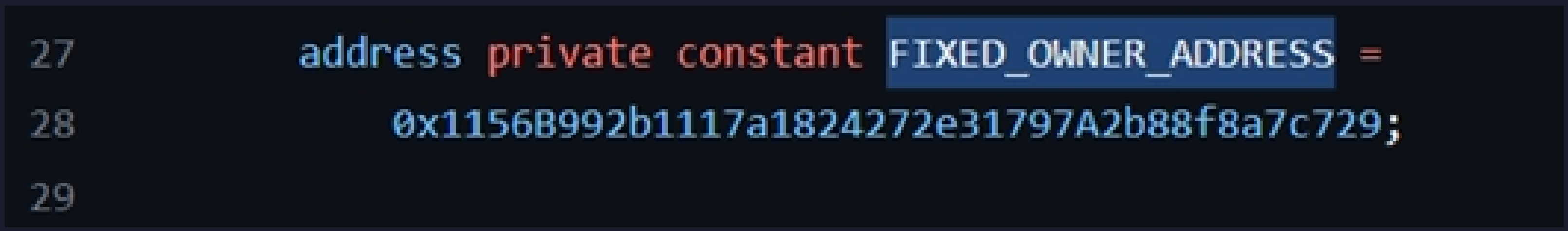
Resolved

A.11 Hardcoded address

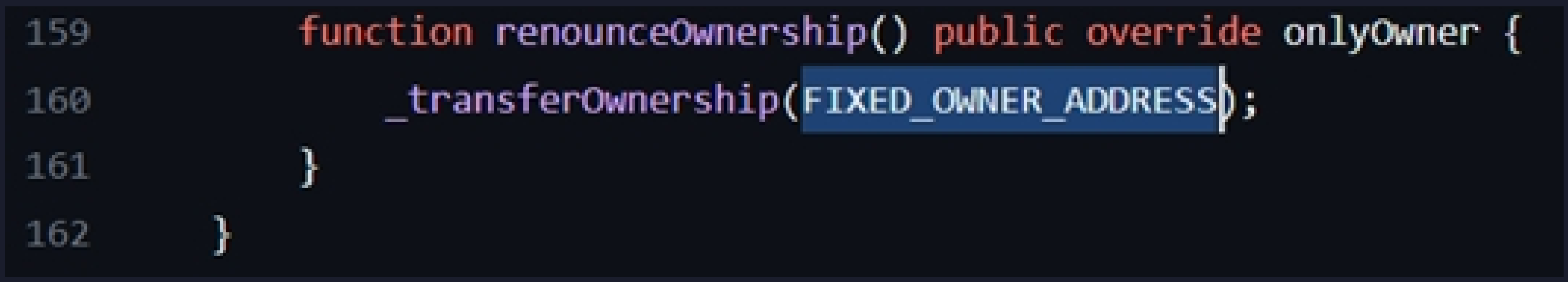
Line

27, 160

Contract - Staking



```
27          address private constant FIXED_OWNER_ADDRESS =
28              0x1156B992b1117a1824272e31797A2b88f8a7c729;
29
```



```
159         function renounceOwnership() public override onlyOwner {
160             _transferOwnership(FIXED_OWNER_ADDRESS);
161         }
162     }
```

Description

The renounceOwnership() transfers the ownership to hardcoded address 0x1156B992b1117a1824272e31797A2b88f8a7c729 . It can happen that the contract gets deployed to the network where this hardcoded address doesn't exist, in that case the project will lose the ownership of the contract.

Recommendation

Make sure to check the address is right and exist on the network/blockchain on which the contract is going to get deployed or take new owner address through pararmeter to set in renounceOwnership().

Status

Acknowledged

Functional Testing

Some of the tests performed are mentioned below:

- ✓ Operator should be able to stake
- ✓ The user address should be able to unstake rewards
- ✓ Owner should be able to renounce the ownership to the fixed address
- ✓ Owner should be able to withdraw tokens
- ✓ Reverts if the unauthorized address calls the stake function
- ✓ Reverts if user tries to unstake without having any stake for his address
- ✓ Reverts if user tries to unstake for invalid month/index
- ✓ Reverts if non owner tries to withdraw tokens with withdraw function

Automated Tests

Static analysis covered some issues like possibility of reentrancy in unstake. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Closing Summary

In this report, we have considered the security of the Ritestream. We performed our audit according to the procedure described above.

Some issues of Medium, Low and informational severity were found, Some suggestions and best practices are also provided in order to improve the code quality and security posture.

Disclaimer

QuillAudits Smart contract security audit provides services to help identify and mitigate potential security risks in Ritestream smart contracts. However, it is important to understand that no security audit can guarantee complete protection against all possible security threats. QuillAudits audit reports are based on the information provided to us at the time of the audit, and we cannot guarantee the accuracy or completeness of this information. Additionally, the security landscape is constantly evolving, and new security threats may emerge after the audit has been completed.

Therefore, it is recommended that multiple audits and bug bounty programs be conducted to ensure the ongoing security of Ritestream smart contracts. One audit is not enough to guarantee complete protection against all possible security threats. It is important to implement proper risk management strategies and stay vigilant in monitoring your smart contracts for potential security risks.

QuillAudits cannot be held liable for any security breaches or losses that may occur subsequent to and despite using our audit services.. It is the responsibility of the Ritestream to implement the recommendations provided in our audit reports and to take appropriate steps to mitigate potential security risks

About QuillAudits

QuillAudits is a secure smart contracts audit platform designed by QuillHash Technologies. We are a team of dedicated blockchain security experts and smart contract auditors determined to ensure that Smart Contract-based Web3 projects can avail the latest and best security solutions to operate in a trustworthy and risk-free ecosystem.

850+
Audits Completed

$30B
Secured

800K
Lines of Code Audited

Follow Our Journey

Audit Report
August, 2023

For

ritestream

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com

audits@quillhash.com