QuillAudits

Audit Report
September, 2023

For

METIS

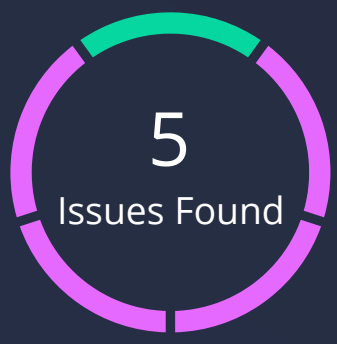
Table of Content

Table of Content

Executive Summary

Project Name	Metis
Project URL	*https://metis.io*
Overview	RMetis is a standard ERC20 token contract with mint and burn function - the Redemption token. The Vesting Vault serves as the contract owner for the RMetis token. When the contract owner calls the deposit from the Vesting Vault with Metis, an equal proportion of the rMetis token is minted into the Vesting Vault. The burn function can be called by any user to burn the rMetis token. Vesting Vault holds the redemption tokens for users to claim as airdrops. And with the use of Openzeppelin merkle tree cryptography library to verify users inclusion, this also indicates the amount a user is allocated. When these tokens have been claimed within the claimable time period for airdrop, users can therefore redeem rMetis token for Metis.
Audit Scope	*https://github.com/t0mcr8se/rMetis-contracts/blob/dev/*
Contracts in Scope	RMetis and VestingVault
Commit Hash	9dd55708895e682f7f2313fd89dfe75333c98ed0
Language	Solidity
Blockchain	Metis
Method	Manual Analysis, Functional Testing, Automated Testing
Review 1	28th August 2023 - 1st September 2023
Updated Code Received	4th September 2023
Review 2	5th September 2023
Fixed In	3fbb208533c06f3bede6704ba7ae0c1976a6fd51

The Number of Security Issues per Severity



High
Medium
Low
Informational

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

Checked Vulnerabilities

- Access Management
- Arbitrary write to storage
- Centralization of control
- Ether theft
- Improper or missing events
- Logical issues and flaws
- Arithmetic Correctness
- Race conditions/front running
- SWC Registry
- Re-entrancy
- Timestamp Dependence
- 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
- Tautology or contradiction
- Return values of low-level calls
- Missing Zero Address Validation
- Private modifier
- Revert/require functions
- Multiple Sends
- Using suicide
- Using delegatecall
- Upgradeable safety
- Using throw

Checked Vulnerabilities

- Using inline assembly
- Style guide violation
- Unsafe type inference
- Implicit visibility level

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 analyzed 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 static analysis.

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.

A. Common Issues

High Severity Issues

No issues found

Medium Severity Issues

No issues found

Low Severity Issues

No issues found

Informational Issues

A.1: Unlocked pragma (pragma solidity ^0.8.0)

Description

Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.

Remediation

Here all the in-scope contracts have an unlocked pragma, it is recommended to lock the same. Moreover, we strongly suggest not to use experimental Solidity features (e.g., pragma experimental ABIEncoderV2) or third-party unaudited libraries. If necessary, refactor the current code base to only use stable features.

Status

Acknowledged

B. Contract - RMetis

High Severity Issues

No issues found

Medium Severity Issues

No issues found

Low Severity Issues

No issues found

Informational Issues

No issues found

C. Contract - VestingVault

High Severity Issues

No issues found

Medium Severity Issues

No issues found

Low Severity Issues

C.1: Ownership Transfer should be a Two-way Process

Description

The contract owner plays a critical role to every of the activity happening on the vault; from regulating the claiming of airdrops with the pause and unpause attribute, to the claiming of rMetis remnant in contract, to redeeming Metis on the price ratio of 1:1, to also redeeming slashed fees. It is important to state the safety of ownership transfer to ensure that a mistaken transfer does not cost losing ownership properties to an unintended address or the null address. The transferOwnership in Openzeppelin Ownable contract automatically sets the passed address as the new owner.

C.1: Ownership Transfer should be a Two-way Process

Remediation

Use the openzeppelin Ownable2Step contract as this checks for the pending owner before it claims ownership.

Status

Resolved

Reference

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/access/Ownable2Step.sol

Informational Issues

C.2: Prevent Deposit Function Call when Msg.value is Zero

```
function deposit() external payable onlyOwner {
    rMetis.mint(msg.value); // Mint an equal amount of msg.value to `this`
}
```

Description

The deposit function can only be called by the owner of the Vesting Vault contract so that the redemption tokens gets minted the exact amount of rMetis token. However, when there is no msg.value sent with the deposit function, no rMetis token is minted but the function is successful. This will cause a waste of gas when called without a native token.

Remediation

Add a check that msg.value is not zero. The inclusion of a check preventing the successful call of the deposit function unless called with an amount of the native token greater than 0 will remedy this issue.

Status

Resolved

C.3: Variables without a Setter Function can be set as Immutable

Description

There are some variables in the contract that only get set at the constructor level but have no other setter functions to change them again in the contract. Making these variables immutable will prevent adding them to the storage slot and instead make them inline with the contract bytecode that gets accessed without overhead gas when needed to perform a functionality. The variables are as follows:
- merkleRoot
- claimDeadline
- startDate
- endDate
- minPrice
- maxPrice

Remediation

Make these variables immutable to help reduce gas consumption.

Status

Resolved

C.4: Incorrect Input Value Sanitization

```
require(minPrice <= maxPrice && maxPrice <= PRICE_PRECISION, "VestingVault: Invalid price
range.");
```

Description

At the constructor level where some variables are to be set for the contract, there was a check to ensure that the passed parameters for the minPrice and maxPrice stays between a minPrice lower than maxPrice and a maxPrice below and/or equal to the PRICE_PRECISION. This check will always pass because it checks the state variable instead with the input values passed into the constructor function.

Remediation

The correct check should be between “_minPrice” and “_maxPrice” so that these values are sanitized before they get set to the state variables.

Status

Resolved

D.1 General Recommendation

The Metis' RMetis token and vesting vault provides a platform for users to claim redemption tokens from the vault supposing they are whitelisted through the merkle tree cryptography. The activity of the vesting vaults rests on the contract owner as he could regulate the claim or redeem activity anytime. There is also the emergencyRecoverToken present in contract for the contract owner to withdraw tokens from the contract in the emergency period. Ensure that the contract owner is trustworthy not to pause the contract unless in an emergency period.

Status

Acknowledged

Functional Tests

Some of the tests performed are mentioned below:

- Should get the name of the token
- Should get the symbol/tinker of the token
- Should get the decimal of the token
- Should deposit Metis into the vault to mint equivalent amount of rMetis tokens
- Should allow contract owner to pause and unpause the claim period
- Should revert if claim function is called after the claimable period
- Should revert when invalid merkle proof is provided to the claim function
- Should revert when airdop is already claimed by an address
- Should allow the contract owner claim all remaining tokens yet unclaimed when it reaches the claim deadline
- Should allow users to redeem Metis and burn the rMetis token
- Should allow the contract owner to claim slashed amounts derived from redeeming Metis and reset currentSlashed to zero
- Should allow the contract owner to suddenly withdraw tokens from the contract with emergencyRecoverToken function
- Should allow any user to burn the rMetis token from the token contract

Automated Tests

No major issues were found. 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 Metis. We performed our audit according to the procedure described above.

Some issues of 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 Metis 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 Metis 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 Metis 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
September, 2023

For

METIS

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com

audits@quillhash.com