QuillAudits

Audit Report, March, 2024

For

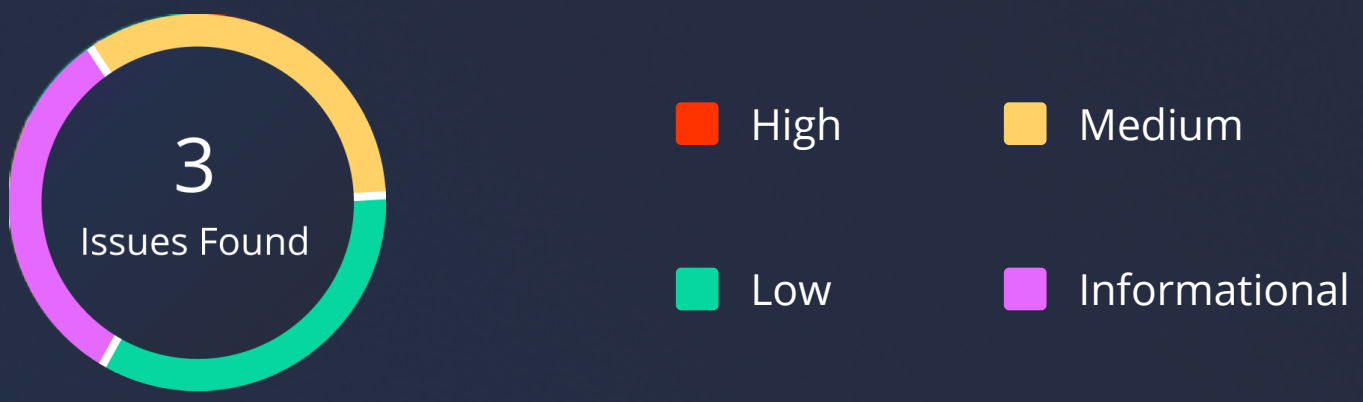
METIS

Table of Content

Executive Summary

Project Name	Metis
Overview	The VestingVault contract incorporates the use of the Merkle tree mechanism that apportions an amount that a user can claim. This contract is an upgradable contract that inherits Ownable2StepUpgradable and PausableUpgradable contracts from the Openzeppelin standard library. These libraries helped to modify some permissioned functions and regulate when some other functions can be called. When users invoke the claim function, a unique token ID of rMetis ERC1155 tokens are minted to them, and the vesting period runs for the period of time provided by the user. Afterwards, users can redeem when the redeem date has reached. Users have the ability to reset the redeem date.
Timeline	7th March 2024 - 14th March 2024
Update code Received	15th March 2024
Second Review	18th March 2024
Method	Manual Review, Functional Testing, Automated Testing, etc. All the raised flags were manually reviewed and re-tested to identify any false positives.
Audit Scope	This audit aimed to analyze the Metis **VestingVault Contract** for quality, security, and correctness.
Source code	*https://github.com/MetisProtocol/rMetis-contracts-v2*
Branch	Main
Fixed in	https://github.com/MetisProtocol/rMetis-contracts-v2/commit/79c7899510c518f806cd61db8d0c02bcfecdf606
Blockchain	Metis Contract will get Deployed on **“Metis L2 chain”.**

Number of Issues per Severity



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

Checked Vulnerabilities

We scanned the application for commonly known and more specific vulnerabilities. Here are some of the commonly known vulnerabilities that we considered:

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- DoS with Block Gas Limit
- Use of tx.origin
- Exception disorder
- Gasless send
- Balance equality
- Byte array
- Transfer forwards all gas
- Transaction-Ordering Dependence
- Redundant fallback function
- ERC20 API violation
- ERC721 API violation
- Malicious libraries
- Compiler version not fixed
- Send instead of transfer
- Style guide violation
- Unchecked external call
- Unchecked math
- 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 behavior.
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper.
- Implementation of ERC-20 token 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

Hardhat, Foundry.

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.

Issues Found

Medium Severity Issues

1. Native tokens sent into the contract is not irredeemable

Path

VestingVault.sol

Function

receive()

Description

```
receive() external payable {}
```

The vault is designed with the receive payable function which makes the contract capable of receiving native tokens. However, it fails to implement a corresponding function to to withdraw native tokens sent into it. This implies that if the contract receives native tokens, they will be stuck in the contract as there is no function to withdraw.

Proof of Concept :

```
function test_stuck_of_funds() external {
    vm.startPrank(claimer);

    uint256 redeemEnd = block.timestamp + 259200;
    vault.claim(claimAmount, proof, claimAmount, redeemEnd);

    assertEq(vault.claimed(claimer), claimAmount);

    uint256 redeemAmount = rMetis.balanceOf(claimer, vault.redeemsCnt());

    // deal the claimer 50 ether to send 10 ether into the vault contract
    vm.deal(address(claimer), 50 ether);
    // send 10 ether to vault
    (bool success,) = address(vault).call{value: 10 ether}("");
    require(success, "failed to transfer");

    console.log("Vault recieved: %e", address(vault).balance);
    console.log("claimer: %e", address(claimer).balance);

    vm.stopPrank();
    // these native tokens that goes into the vault contract is stuck
    // due to no withdraw function for the purpose of withdrawing native tokens
}
```

Recommendation

Create a withdrawal function for native tokens if the vesting contract is created on purpose to receive native tokens.

Metis Team's Comment

This can be dismissed because the contract will be deployed on Metis L2 blockchain, which has the native token in an ERC20 wrapper at predeploy address: 0xDeadDeAddeAddEAddeadDEaDDEAdDeaDDeAD0000

Status

Resolved

Low Severity Issues

2. Centralization Issue

Files

VestingVault.sol

Modifier

whenNotPaused

Description

There are some critical functions in the contract with the whenNotPaused modifier which restricts when it can be invoked by anyone or not. Since the contract owner can pause and unpause the contract, activities on the contract could be halted by a malicious contract owner.

Recommendation

Users should have a high level of trust in the contract owner to regulate activities as at when due or advised that a multisig is used as the owner.

Status

Acknowledged

Informational Issues

3. Floating solidity version

Path

VestingVault

Version

```
pragma solidity ^0.8.20;
```

Description

Contract has a floating solidity pragma version, ^0.8.20. This is present also in inherited contracts. Locking the pragma helps to ensure that the contract does not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively. The recent solidity pragma version also possesses its own unique bugs.

Recommendation

Making the contract use a stable solidity pragma version prevents bugs occurrence that could be ushered in by prospective versions. It is recommended, therefore, to use a fixed solidity pragma version while deploying to avoid deployment with versions that could expose the contract to attack.

Status

Resolved

Functional Tests

Some of the tests performed are mentioned below:

Token Contract

- ✓ Should confirm successful claiming of all apportioned amount in the merkle proof
- ✓ Should test claiming with an redeemEnd value lesser than cliff period
- ✓ Should observe claiming in small amount until it reaches amount in the merkle
- ✓ Should revert when users invoke the claim function after deadline has passed
- ✓ Should estimate the implication for minting new rMetis unique id for every small portion claimed
- ✓ Should ascertain the amount of metis tokens redeemed after the vesting period
- ✓ Should burn all rMetis tokens minted during the claim period
- ✓ Should verify the time interval expected for users to reset their end of redeem date
- ✓ Should confirm if the contract can receive native tokens and retrieve it

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 the **Metis** codebase. 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 **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.

1000+
Audits Completed

$30B
Secured

+1M
Lines of Code Audited

Follow Our Journey

Audit Report
March, 2024

For

METIS

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com

audits@quillhash.com