KnownOrigin Merkle Royalties Distributor Audit

October 2021

By CoinFabrik

Introduction

CoinFabrik was asked to audit the contracts for the KnownOrigin's Merkle Royalties Distributor project. First we will provide a summary of our discoveries, which include a minor security vulnerability and one recommendation, and then we will show the details of our findings.

Summary

The contracts audited are from the GitHab repository at https://github.com/knownorigin/merkle-royalties-distributor/. The audit is based on the commit `0469dea0911ef35f1b1f2d387b7587564eb8a410`.

Contracts

The audited contracts are:

- contracts/MerkleVault.sol
- contracts/IMerkleVault.sol

Analyses

The following analyses were performed:

- Misuse of the different call methods
- Integer overflow errors
- Division by zero errors
- Outdated version of Solidity compiler
- Front running attacks
- Reentrancy attacks
- Misuse of block timestamps
- Softlock denial of service attacks
- Functions with excessive gas cost
- Missing or misused function qualifiers
- Needlessly complex code and contract interactions
- Poor or nonexistent error handling

- Failure to use a withdrawal pattern
- Insufficient validation of the input parameters
- Incorrect handling of cryptographic signatures

We further analyzed the logic of the contract and offchain interactions to establish whether contract users could claim funds illicitly.

Findings and Fixes

ID	Title	Severity	Status
MI-1	Merkle Tree Unclaimed Funds Resulting in Excessive Gas Cost	Minor	Not fixed

We further issue a recommendation based on a situation in which the contracts could be misused depending on what happens offchain. Development team explained the case is addressed securely as described below.

Severity Classification

Security risks are classified as follows:

- **Critical:** These are issues that we manage to exploit. They compromise the system seriously. They must be fixed **immediately**.
- **Medium:** These are potentially exploitable issues. Even though we did not manage to exploit them or their impact is not clear, they might represent a security risk in the near future. We suggest fixing them **as soon as possible**.
- **Minor:** These issues represent problems that are relatively small or difficult to take advantage of but can be exploited in combination with other issues. These kinds of issues do not block deployments in production environments. They should be taken into account and be fixed **when possible**.
- **Enhancement:** These kinds of findings do not represent a security risk. They are best practices that we suggest to implement.

This classification is summarized in the following table:

SEVERITY	EXPLOITABLE	ROADBLOCK	TO BE FIXED

Critical	Yes	Yes	Immediately
Medium	In the near future	Yes	As soon as possible
Minor	Unlikely	No	Eventually
Enhancement	No	No	Eventually

Issues Found by Severity

Critical severity

No issues found.

Medium severity

No issues found.

Minor severity

MI-01 Merkle Tree Unclaimed Funds Resulting in Excessive Gas Cost

At each time there is a Merkle tree whose nodes are used by claimants to claim their tokens. Basically, in order to claim his funds a claimant must present the node which includes the amount he is owed and the associated Merkle proof, and the contract will transfer this amount and mark these funds as claimed if the proof is valid. Merkle trees are indexed by a counter (merkleVersion) and only one tree is active at a time.

According to developers, unclaimed funds are transferred from Merkle tree to Merkle tree offchain. If a big number of claimants failed to claim their funds, then this would imply that Merkle proofs become bigger and more expensive to verify.

Solution

Add an ownerOnly `claimUnclaimed()` function that takes an array of claimants and forcibly claims funds on their behalf.

Enhancements

No issues found.

Recommendations

Merkle Tree Updating and misuse

The struct `fundsClaimed[claimant][merkleVersion]` records whether a user (claimant) has claimed the funds associated with a Merkle tree. From time to time, a new Merkle tree needs to be created. This update happens offchain and outside the contracts. At any time the contract owner may call, `updateMerkleTree()` which, in particular, increases `merkleVersion` by one.

Since `claim()`, as well as `fundsClaimed[claimant][merkleVersion]`, relies on `merkleVersion` the information in the old Merkle tree is no longer accessible after an update. At least the following threats are possible.

New Merkle trees must be carefully constructed incorporating only the unclaimed funds from their predecessor tree (and new claim additions), and the new merkle proofs must be communicated to users through a secure channel. During the update process, the contract should be paused. If not done carefully, then a user could be able to claim twice or may lose the ability to claim his funds.

Response

Developers answered that a Merkle tree version records cumulative totals so that users claim from one version alone. In particular, if a claimant has not claimed his funds, then upon updating his balance is added to the latest version of the Merkle tree. The Merkle tree creator reconciles what has been claimed and what has not been claimed into fundsClaimed[`claimant][merkleVersion`].

The flow is:

1 - pause contract
2 - generate new tree and include previous ones
3 - set new data
4 - unpause

This flow and mechanism ensures no problems occur.

Conclusion

We found the contracts to be simple and straightforward and have an adequate amount of documentation. The contract owner controls the claimant's balances, and a big part of this happens offchain and outside of the contracts scope. The operation should follow the flow documented above in order to avoid inconsistencies with these balances.

MORE DETAILS

Disclaimer: This audit report is not a security warranty, investment advice, or an approval of the KnownOrigin Merkle Royalties Distributor project since CoinFabrik has not reviewed its platform. Moreover, it does not provide a smart contract code faultlessness guarantee.