yAudit Royco CCDM Merkle Vault Review

Review Resources:

- docs

Auditors:

- Fedebianu
- Puxyz

Table of Contents

Review Summary

Royco CCDM Merkle Vault

The Royco CCDM Merkle Vault upgrade introduces an enhanced deposit model, aggregating multiple deposits within a Merkle tree structure.

The contracts of the Royco CCDM repository were reviewed over two days. The code review was performed by two auditors between 9th January and 10th January 2024. The repository was under active development during the review, but the review was limited to commit 99523629a6ff546de60aa8138405ef898cb0d959 for the Royco CCDM repo.

This review follows up on the initial Royco CCDM audit. Several mitigations were implemented between the first and second reviews. Our objectives were to verify the effectiveness of these mitigations and identify any potential issues introduced by the changes and new developments.

Scope

The scope of the review consisted of the following contracts at the specific commits:

Royco CCDM

```
src
├── core
│   ├── DepositExecutor.sol
│   └── DepositLocker.sol
├── interfaces
│   ├── ILayerZeroComposer.sol
│   ├── IOFT.sol
│   └── IWETH.sol
└── libraries
    ├── CCDMFeeLib.sol
    └── CCDMPayloadLib.sol
```

After the findings were presented to the Royco team, fixes were made and included in several PRs.

This review is a code review to identify potential vulnerabilities in the code. The reviewers did not investigate security practices or operational security and assumed that privileged accounts could be trusted. The reviewers did not evaluate the security of the code relative to a standard or specification. The review may not have identified all potential attack vectors or areas of vulnerability.

yAudit and the auditors make no warranties regarding the security of the code and do not warrant that the code is free from defects. yAudit and the auditors do not represent nor imply to third parties that the code has been audited or is free from defects. By deploying or using the code, Royco and users of the contracts agree to use the code at their own risk.

Code Evaluation Matrix

Category	Mark	Description
Access Control	Good	Proper access control mechanisms are implemented, ensuring only authorized users or contracts can interact with critical functions. Clear separation of privileges.
Mathematics	Good	Precise handling of token decimals and LP calculations. No overflow/underflow risks.
Complexity	Good	Modular design with clear separation of concerns. Functions follow the single responsibility principle. Complexity justified by cross-chain functionality.
Libraries	Good	Relies on standard OpenZeppelin and Solmate libraries. LayerZero is used as a proven bridge solution.
Decentralization	Average	Distributed control through multiple roles. Critical operations require multiple participants. Some of them have to be trusted.
Code stability	Good	Mature and stable codebase. No experimental features.
Documentation	Good	The comprehensive documentation explains the core functions and overall system architecture.
Monitoring	Good	Monitoring mechanisms are in place to track key events and changes within the system.
Testing and verification	Average	Basic functionality and fuzzing covered, but invariant tests are missing.

Findings Explanation

Findings are broken down into sections by their respective impact:

- Critical, High, Medium, Low impact
  - These are findings that range from attacks that may cause loss of funds, impact control/ownership of the contracts, or cause any unintended consequences/actions that are outside the scope of the requirements.
- Gas savings
  - Findings that can improve the gas efficiency of the contracts.
- Informational
  - Findings including recommendations and best practices.

Critical Findings

1. Critical - Incorrect bridge amount normalization leads to token loss in `merkleBridgeLpTokens()`.

`merkleBridgeLpTokens()` incorrectly normalizes token amounts by dividing them by the decimal conversion rate, causing users to bridge a significantly reduced amount of tokens than intended.

Technical Details

`merkleBridgeLpTokens()` mistakenly divides the token amounts without restoring them to their original value, effectively reducing the amount of tokens being bridged.

Example scenario when bridging one token with 18 decimals and shared decimals of 8:

1. Initial amount: 1000000000000000000 (1e18)
2. After current normalization: 100000000 (1e8)
3. Amount bridged: 0.000000001 tokens instead of 1 token

The intent of the normalization should be to only remove dust amounts by performing both division and multiplication with the conversion rate.

Impact

Critical. Users' funds will be locked after bridging LP tokens through `merkleBridgeLpTokens()`.

Recommendation

Modify the normalization to only remove dust while preserving the original amount:

```
        if (token0_DecimalConversionRate != 1) {
-           token0_AmountToBridge = token0_AmountToBridge /
token0_DecimalConversionRate;
+           token0_AmountToBridge = (token0_AmountToBridge /
token0_DecimalConversionRate) * token0_DecimalConversionRate;
        }
        if (token1_DecimalConversionRate != 1) {
-           token1_AmountToBridge = token1_AmountToBridge /
token1_DecimalConversionRate;
+           token1_AmountToBridge = (token1_AmountToBridge /
token1_DecimalConversionRate) * token1_DecimalConversionRate;
        }
```

Developer Response

Fixed in this commit.

Informational Findings

1. Informational - Avoid redundant logic

Technical Details

Several functions (`merkleBridgeSingleTokens()`, `merkleBridgeLpTokens()`, `bridgeSingleTokens()`, `bridgeLpTokens()`, etc.) perform very similar operations when bridging depositors tokens. Common tasks such as fee management, amount calculation, dust handling, bridging payload construction, and event emissions are repeated. This repetitive logic increases contract size, makes the code harder to maintain, and risks inconsistencies if updates are made in one place but not propagated everywhere.

While this poses no security risk, it complicates future maintenance and potentially opens the door for bugs through logic drift. Consistency issues can arise if one bridging function is patched or improved while others are overlooked.

Impact

Informational.

Recommendation

Refactor the flows into shared internal methods that handle common tasks in one place.

Developer Response

Acknowledged.

Final remarks

The Merkle-based deposit upgrade significantly improves scalability for large cross-chain deposit campaigns. By storing all depositors in a Merkle tree, bridging requires only a single root and aggregate deposit amount rather than an exhaustive list of depositors. This reduces gas costs and simplifies on-chain accounting. Depositors' withdrawals, validated by standard Merkle proofs, further streamline the settlement process.