Royco CCDM

Security Assessment

February 4th, 2025 — Prepared by OtterSec

Robert Chen r@osec.io

Renato Eugenio Maria Marziano renato@osec.io

Table of Contents

01 — Executive Summary

Overview

Royco engaged OtterSec to assess the `cross-chain-deposit-module` program. This assessment was conducted between January 20th and January 31st, 2025. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 3 suggestion throughout this audit engagement.

We made suggestions regarding inconsistencies in the codebase and ensuring adherence to coding best practices (OS-RCM-SUG-00) and advised to enhance efficiency through improved gas optimization techniques (OS-RCM-SUG-01), as well as pointing out a minor accounting issue (OS-RCM-SUG-02).

Scope

The source code was delivered to us in a Git repository at https://github.dev/roycoprotocol/cross-chain-deposit-module. This audit was performed against c670ded.

A brief description of the program is as follows:

Name	Description
cross-chain-deposit-module	An extension of the Royco Protocol, designed specifically for cross-chain liquidity acquisition campaigns. It enables Incentive Providers (IPs) to offer incentives for Action Providers (APs) to commit liquidity on a source chain and programmatically execute specific actions on a destination chain (such as supplying, LPing, swapping, etc).

02 — General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Description
OS-RCM-SUG-00	Suggestions regarding inconsistencies in the codebase and ensuring adherence to coding best practices.
OS-RCM-SUG-01	Recommendations for enhancing efficiency through improved gas optimization techniques.
OS-RCM-SUG-02	Incorrect deposit tracking in `DepositLocker` causes potential underflow errors and incorrect withdrawal calculations.

Code Maturity OS-RCM-SUG-00

Description

1. Add a modifier similar to the `onlyWeirollWallet` modifier in `DepositLocker` to `DepositExecutor`. This may be utilized to guard specific functions such as `executeDepositRecipes`, `withdrawMerkleDeposit`, and `withdrawIndividualDeposits`, improving access control.
2. Currently, `setCampaignDepositRecipe` in `DepositExecutor` assigns `_depositRecipe` (a `calldata` argument) directly to `campaign.depositRecipe`. However, if `depositRecipe` were a `storage`, it may have been possible for a reentrancy time-of-Check to time-of-Use vulnerability to occur. Apply the `nonReentrant` modifier as a reentrancy guard to `setCampaignDepositRecipe`.

   >_ src/core/DepositExecutor.sol SOLIDITY

   ```
   function setCampaignDepositRecipe(bytes32 _sourceMarketHash, Recipe calldata
       ↪  _depositRecipe) external onlyCampaignOwner(_sourceMarketHash) {
       // Get the deposit campaign corresponding to this source market hash
       DepositCampaign storage campaign =
           ↪  sourceMarketHashToDepositCampaign[_sourceMarketHash];
       // Check that the campaign has been initialized
       require(address(campaign.receiptToken) != address(0), CampaignIsUninitialized());
       // Set the campaign's deposit recipe and mark the campaign as unverified
       campaign.depositRecipe = _depositRecipe;
       delete campaign.verified;
       emit CampaignDepositRecipeSet(_sourceMarketHash);
   }
   ```

3. Add a check in `DepositLocker::merkleBridgeLpTokens` to ensure the bridged token is a legitimate Uniswap V2 LP token.

Remediation

Implement the above-mentioned suggestions.

Gas Optimizations

OS-RCM-SUG-01

Description

1. Gas profiling utilizing `forge snapshot --diff` revealed that the optimizations in this commit (cad9547), particularly the `decimalConversionRate != 1` checks, increased gas consumption instead of reducing it. It is recommended to remove these checks to improve efficiency.
2. Update `DepositCampaign.unlockTimestamp` type from a `uint256` to `uint64` and reorder the `DepositCampaign` structure as shown below to allow for tighter packing to improve storage efficiency by optimizing the structure layout to minimize storage slots utilized, thereby reducing gas costs.

>_ src/core/DepositExecutor.sol SOLIDITY

```
struct DepositCampaign {
    address owner;
    bool verified;
    uint8 numInputTokens;
    uint64 unlockTimestamp;
    ERC20[] inputTokens;
    ERC20 receiptToken;
    Recipe depositRecipe;
    mapping(uint256 => address) ccdmNonceToWeirollWallet;
    mapping(address => WeirollWalletAccounting) weirollWalletToAccounting;
}
```

Remediation

Incorporate the above gas optimizations.

Deposit Accounting Issue OS-RCM-SUG-02

Description

The field `walletInfo.amountDeposited` is not reset after bridging, leading to discrepancies in withdrawal amounts. When a user performs a sequence of `deposit` → `bridge` → `deposit` → `withdraw`, the system fails with an underflow error, preventing correct fund retrieval.

The code would undeflow here:

>_ *src/core/DepositLocker.sol* SOLIDITY

```
function withdraw() external nonReentrant {
    ...

    uint256 amountToWithdraw = walletInfo.amountDeposited;

    require(walletInfo.ccdmNonceOnDeposit > depositorInfo.lastCcdmNonceBridged &&
        ↪  amountToWithdraw > 0, NothingToWithdraw());

    depositorInfo.totalAmountDeposited -= amountToWithdraw;
    delete walletInfo.amountDeposited;
    ...
}
```

Note that this would not disrupt the flow of funds for rational user.

Remediation

Ensure `walletInfo.amountDeposited` is reset after bridging to maintain accurate balances. Additional test cases should be implemented to validate interactions between deposit, bridge, and withdrawal processes.

A — Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings.

CRITICAL

Vulnerabilities that immediately result in a loss of user funds with minimal preconditions.

Examples:

- Misconfigured authority or access control validation.
- Improperly designed economic incentives leading to loss of funds.

HIGH

Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit.

Examples:

- Loss of funds requiring specific victim interactions.
- Exploitation involving high capital requirement with respect to payout.

MEDIUM

Vulnerabilities that may result in denial of service scenarios or degraded usability.

Examples:

- Computational limit exhaustion through malicious input.
- Forced exceptions in the normal user flow.

LOW

Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.

INFO

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants.
- Improved input validation.

B — Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that others may have missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.