



Euler Claims Audit

Presented by:

OtterSec

Robert Chen

Vishvesh Rao

contact@osec.io

r@osec.io

vishvesh@osec.io



Contents

01 Executive Summary	2
Overview	2
Key Findings	2
Scope	2
02 Findings	3
03 General Findings	4
OS-EUL-SUG-00 Validate Proof Length	5
OS-EUL-SUG-01 Minor Optimizations	6
 Appendices	
A Vulnerability Rating Scale	7
B Procedure	8

01 | Executive Summary

Overview

Euler Finance engaged OtterSec to perform an assessment of the `euler-claims-contract` program. This assessment was conducted between April 9th and April 10th, 2023. For more information on our auditing methodology, see [Appendix B](#).

Key Findings

Over the course of this audit engagement, we produced 2 findings total.

In particular, we noted minor security improvements and gas optimizations around the usage of the `MerkleProof` library. We also provided additional recommendations to mitigate against accidental misuse of the `transferOwnership` function.

Scope

The source code was delivered to us in a git repository at github.com/euler-xyz/euler-claims-contract. This audit was performed against commit [1c45b22](#).

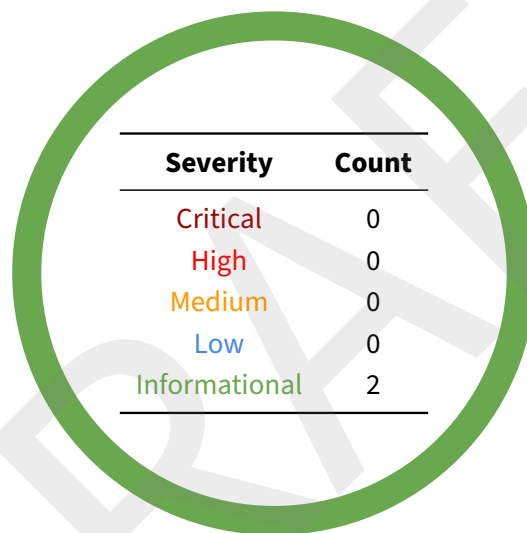
A brief description of the programs is as follows.

Name	Description
euler-claims-contract	Euler Claims contract is a merkle-tree based distributor contract that allows users to redeem recovered funds from the 2023 Euler protocol hack.

02 | Findings

Overall, we reported 2 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings don't have an immediate impact but will help mitigate future vulnerabilities.



Severity	Count
Critical	0
High	0
Medium	0
Low	0
Informational	2

03 | 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 antipatterns and could lead to security issues in the future.

ID	Description
OS-EUL-SUG-00	Given changes in leaf node format, missing proof length validation could allow an attacker to theoretically construct malicious Merkle proofs for intermediate nodes.
OS-EUL-SUG-01	Miscellaneous suggestions to improve gas efficiency and remove potential footguns.

OS-EUL-SUG-00 | Validate Proof Length

Description

The Euler Claims contract uses OpenZeppelin's `MerkleProof` library to validate proofs. If the leaf value happens to be exactly 64 bytes, an attacker could collide an intermediate Merkle tree node with the encoded data.

@openzeppelin/contracts/utils/cryptography/MerkleProof.sol

SOLIDITY

```
* WARNING: You should avoid using leaf values that are 64 bytes long prior to
* hashing, or use a hash function other than keccak256 for hashing leaves.
* This is because the concatenation of a sorted pair of internal nodes in
* the merkle tree could be reinterpreted as a leaf value.
* OpenZeppelin's JavaScript library generates merkle trees that are safe
* against this attack out of the box.
```

Note that in the current implementation, this is not a security concern because the leaf size is at least 128 bytes before hashing. However, seemingly innocuous changes such as removing the token amounts from the leaf node could result in an exploitable issue.

EulerClaims.sol

SOLIDITY

```
keccak256(abi.encode(index, msg.sender, tokenAmounts))
```

Remediation

Consider explicit validation on the proof length to avoid any risk of reinterpreting leaf nodes.

OS-EUL-SUG-01 | Minor Optimizations

Description

1. Because proof is specified as `calldata`, consider using `MerkleProof::verifyCalldata` instead of `verify`.
2. In `transferOwnership`, consider using a two-step process to transfer ownership to avoid transferring ownership to an inaccessible address.

Remediation

Implement the suggestions as described.

A | Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings can be found in the [General Findings](#) section.

Critical	<p>Vulnerabilities that immediately lead to loss of user funds with minimal preconditions</p> <p>Examples:</p> <ul style="list-style-type: none">• Misconfigured authority or access control validation• Improperly designed economic incentives leading to loss of funds
High	<p>Vulnerabilities that could lead to loss of user funds but are potentially difficult to exploit.</p> <p>Examples:</p> <ul style="list-style-type: none">• Loss of funds requiring specific victim interactions• Exploitation involving high capital requirement with respect to payout
Medium	<p>Vulnerabilities that could lead to denial of service scenarios or degraded usability.</p> <p>Examples:</p> <ul style="list-style-type: none">• Malicious input that causes computational limit exhaustion• Forced exceptions in normal user flow
Low	<p>Low probability vulnerabilities which could still be exploitable but require extenuating circumstances or undue risk.</p> <p>Examples:</p> <ul style="list-style-type: none">• Oracle manipulation with large capital requirements and multiple transactions
Informational	<p>Best practices to mitigate future security risks. These are classified as general findings.</p> <p>Examples:</p> <ul style="list-style-type: none">• 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 implementation of the program 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 get a comprehensive understanding of 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 the other 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.