Zellic

Euler

Smart Contract Security Assessment

April 11, 2023

Prepared for:

Brandon Neal

Euler Labs Ltd.

Prepared by:

Syed Faraz Abrar and William Bowling

Zellic Inc.

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please contact us at hello@zellic.io.

1 Executive Summary

Zellic conducted a security assessment for Euler Labs Ltd. from April 10th to April 11th, 2023. During this engagement, Zellic reviewed Euler's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1 Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Is there a way for users to claim redemption tokens more than once?
- Can the owner accidentally lock funds into the contract?
- Can the owner accidentally lose ownership of the contract?

1.2 Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

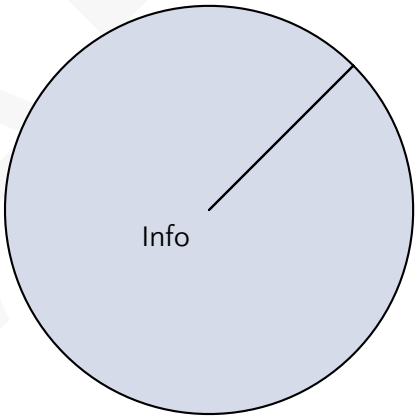
Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.3 Results

During our assessment on the scoped Euler contracts, we discovered two findings. No critical issues were found. Both findings were informational in nature.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	0
Low	0
Informational	2



2 Introduction

2.1 About Euler

Euler is a noncustodial protocol on Ethereum that allows users to lend and borrow almost any crypto asset.

The EulerClaims contract implements a mechanism for users to redeem funds that were stolen during the March 13th, 2023 hack of the Euler protocol and subsequently recovered.

2.2 Methodology

During a security assessment, Zellic works through standard phases of security auditing including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review the contracts' external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the code base in general. We look for violations of industry best practices and guidelines and code quality stan-

dards. We also provide suggestions for possible optimizations, such as gas optimization, upgradeability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

2.3 Scope

The engagement involved a review of the following targets:

Euler Contracts

Repository	https://github.com/euler-xyz/euler-claims-contract
Version	euler-claims-contract: `1c45b22112e0b04c4a6051fd38677e5cd824496b`
Program	• EulerClaims
Type	Solidity
Platform	EVM-compatible

2.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of one engineer day. The assessment was conducted over the course of two days.

Contact Information

The following project managers were associated with the engagement:

Chad McDonald, Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Syed Faraz Abrar, Engineer
faith@zellic.io

William Bowling, Engineer
vakzz@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

April 10, 2023	Start of primary review period
April 11, 2023	End of primary review period
TBD	Closing call

3 Detailed Findings

3.1 Indexes in `recoverTokens()` might not match `TokenAmounts()`

- **Target**: EulerClaims.sol
- **Category**: Coding Mistakes
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

The `recoverTokens()` function is intended to be used by the owner to recover tokens from the contract. This might be required if the tokens have been in the contract for a long time without being claimed. The owner passes in a list of indexes that will be marked as "claimed" and a list of tokens and their corresponding amounts to be recovered.

The current implementation of `recoverTokens()` lacks a check to ensure that the indexes provided by the owner correspond to the token amounts being recovered.

```
function recoverTokens(uint[] calldata indexList, TokenAmount[]
    calldata tokenAmounts) external onlyOwner nonReentrant {
    for (uint i = 0; i < indexList.length; ++i) {
        uint index = indexList[i];
        require(!alreadyClaimed[index], "already claimed");
        alreadyClaimed[index] = true;
        emit ClaimedByOwner(index);
    }
    for (uint i = 0; i < tokenAmounts.length; ++i) {
        SafeERC20.safeTransfer(IERC20(tokenAmounts[i].token), owner,
    tokenAmounts[i].amount);
    }
}
```

Impact

The state tracking of the token balances in the contract may be left in an inconsistent state.

Recommendations

Merkle proofs should be used to ensure that the indexes passed in match the token amounts being recovered. The number of items in both the `indexList` and the `recoverTokensList` should be checked to be the same.

An example implementation is shown below:

```
struct RecoverTokens {
    uint index;
    address from;
    TokenAmount[] tokenAmounts;
    bytes32[] proof;
}

function recoverTokens(RecoverTokens[] calldata recoverTokensList)
    external onlyOwner nonReentrant {
    for (uint i = 0; i < recoverTokensList.length; ++i) {
        uint index = recoverTokensList[i].index;
        bytes32[] calldata proof = recoverTokensList[i].proof;
        address from = recoverTokensList[i].from;
        TokenAmount[] calldata tokenAmounts
    = recoverTokensList[i].tokenAmounts;

        require(!alreadyClaimed[index], "already claimed");
        alreadyClaimed[index] = true;
        emit ClaimedByOwner(index);


        require(MerkleProof.verify(proof, merkleRoot,
    keccak256(abi.encode(index, from, tokenAmounts))), "proof invalid");
        for (uint i = 0; i < tokenAmounts.length; ++i) {
            SafeERC20.safeTransfer(IERC20(tokenAmounts[i].token), owner,
    tokenAmounts[i].amount);
        }
    }
}
```

Remediation

TBD

3.2 Single-step ownership transfer may cause loss of contract ownership

- **Target**: EulerClaims.sol
- **Category**: Coding Mistakes
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

The `transferOwnership()` function is used to transfer ownership of the contract to a different address. This is done in a single step, meaning that the ownership is fully transferred after this function is called.

```
function transferOwnership(address newOwner) external onlyOwner {
    require(newOwner ≠ address(0), "owner is zero");
    owner = newOwner;
    emit OwnerChanged(newOwner);
}
```

Impact

The function checks that the new owner is not set to `address(0)` to prevent an erroneous transfer of ownership. However, there is still a risk that the owner may input an incorrect address for the new owner, either due to a typo or other mistakes. If this happens, it can result in a loss of ownership of the contract, potentially leading to unclaimed funds being permanently locked into the contract.

Recommendations

Consider using a two-step ownership transfer mechanism. See OpenZeppelin's implementation of Ownable2Step here.

Remediation

TBD

4 Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the smart contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

4.1 Module: EulerClaims.sol

Function: `claimAndAgreeToTerms(byte[32] acceptanceToken, uint256 index, TokenAmount tokenAmounts, byte[32] proof)`

Used by users to claim their redemption tokens.

Inputs

- `acceptanceToken`
  - **Control**: Fully controlled.
  - **Constraints**: Must be a hash of the user's address concatenated with a pre-set terms and conditions hash.
  - **Impact**: Reverts if this is not correct.
- `index`
  - **Control**: Fully controlled.
  - **Constraints**: Used to verify the Merkle proof, so it cannot be forged.
  - **Impact**: Reverts if forged.
- `tokenAmounts`
  - **Control**: Fully controlled.
  - **Constraints**: Used to verify the Merkle proof, so it cannot be forged.
  - **Impact**: Reverts if forged.
- `proof`
  - **Control**: Fully controlled.
  - **Constraints**: The proof that the other inputs are verified against. Cannot be forged as it is used to get back to the Merkle root.
  - **Impact**: Reverts if forged.

Branches and code coverage (including function calls)

Intended branches

- Simple Merkle tree works correctly.
  - ☑ Test coverage
- Large Merkle tree works correctly.
  - ☑ Test coverage

Negative behavior

- Reverts if terms and conditions were not accepted.
  - ☑ Negative test
- Reverts if an invalid proof is passed in.
  - ☑ Negative test
- Reverts if an invalid index is passed in.
  - ☑ Negative test
- Reverts if the user claiming the tokens is not eligible to them.
  - ☑ Negative test
- Reverts if `tokenAmounts` is forged or tampered with.
  - ☑ Negative test

5 Audit Results

At the time of our audit, the code was not deployed to mainnet Ethereum.

During our audit, we discovered two findings. Both were suggestions (informational). Euler Labs Ltd. acknowledged all findings and implemented fixes.

5.1 Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the audit version of our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.