Zellic

Prepared for
whitehat@inferencelabs.com
Inference Labs

Prepared by
Jaeeu Kim
Zellic

April 10, 2024

Omron
Smart Contract Security Assessment

Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Inference Labs on April 10th, 2024. During this engagement, Zellic reviewed Omron's code for security vulnerabilities, design issues, and general weaknesses in security posture.

We conducted an earlier audit of this codebase on March 26th, 2024, focusing on commit 1d8aeb9f ↗. This audit reviewed modifications made to the OmronDeposit contract following our previous assessment.

Changes made to the contract after the March 26th audit include the following:

- Addition of the claim method
- Addition of the withdrawal method
- Addition of a timestamp where points will cease accruing, withdrawals will become enabled, and deposits will become disabled
- Delegation of claims and withdrawals to a claim manager contract, to be set by the owner in the future

1.2. 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 risk of draining more tokens than originally deposited?
- Is there a risk of withdrawing funds belonging to another user?
- Are there any potential risks associated with point accounting?

1.3. 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

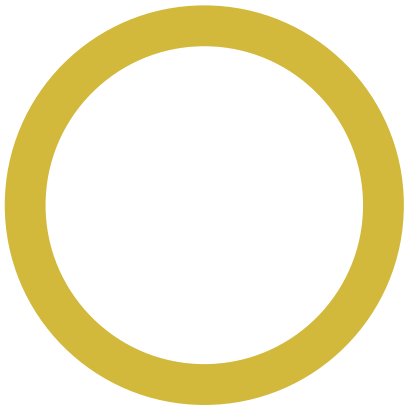
Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.4. Results

During our assessment on the scoped Omron contracts, we discovered two findings, both of which were medium impact.

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	0
🟧 High	0
🟨 Medium	2
🟩 Low	0
⬜ Informational	0



2. Introduction

2.1. About Omron

Inference Labs contributed the following description of Omron:

> Omron provides verified inferences of restaking optimizations to improve efficiency and yields in existing liquid restaking protocols. In its first stage, Omron deploys an LRT and WETH deposit, which allows users to accrue omron points in exchange for their deposits into the contract. During this stage, Omron is using user deposits to track on-chain restaking information for training purposes.

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 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 codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability 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:

Omron Contracts

Repository	https://github.com/inference-labs-inc/omron-contracts ↗
Version	omron-contracts: `61ad8980fde12007d2e7ef5c03319031bd4ac1f6`
Program	contracts/OmronDeposit.sol
Type	Solidity
Platform	EVM-compatible

2.4. Project Overview

Zellic was contracted to perform a security assessment with one consultant for a total of one person-day. The assessment was conducted over the course of one calendar day.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald
Engagement Manager
chad@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Jaeeu Kim
Engineer
jaeeu@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

April 10, 2024	Start of primary review period
April 10, 2024	End of primary review period

3. Detailed Findings

3.1. Centralization risk in withdrawal

Target	OmronDeposit		
Category	Business Logic	**Severity**	High
Likelihood	Low	**Impact**	Medium

Description

The `withdrawTokens` function is used to withdraw all of a user's tokens to `claimManager` after deposits have stopped. The address of the `claimManager` can be set by the owner using the `setClaimManager` function.

There is a possibility that withdrawals may not be delivered to the user because the `claimManager` can be changed at any time, even if the user has verified that it is set correctly before depositing.

```
function withdrawTokens(
    address _userAddress
)
    external
    nonReentrant
    whenNotPaused
    onlyClaimManager
    onlyAfterDepositStop
    returns (uint256[] memory withdrawnAmounts)
{
    // ...

    for (uint256 i; i < allWhitelistedTokens.length; ) {
        // ...

        IERC20 token = IERC20(allWhitelistedTokens[i]);
        token.safeTransfer(claimManager, userBalance);

        unchecked {
            ++i;
        }
    }
    // ...
}
```

```
function setClaimManager(address _newClaimManager) external onlyOwner {
    if (_newClaimManager == address(0)) {
        revert ZeroAddress();
    }
    // Set the new claim manager
    claimManager = _newClaimManager;

    emit ClaimManagerSet(_newClaimManager);
}
```

Impact

A malicious or compromised owner could potentially drain the contract.

Recommendations

Ensure that the ownership of the contract and the reliability of the `claimManager` are prominently documented, so users are aware of and accept the associated risks.

Remediation

This issue has been acknowledged by Inference Labs.

According to Inference Labs's response, they plan to implement a withdraw method in a future claim-manager contract where users will be able to withdraw their funds. For now, the expected functionality for the deposit contract is to pass user funds up to the claim-manager contract, which will handle their tokens as per functionality implemented in the future.

3.2. Partial token malfunction could disrupt the entire withdrawal process

Target	OmronDeposit		
Category	Coding Mistakes	**Severity**	High
Likelihood	Medium	**Impact**	Medium

Description

The `withdrawTokens` function is used to withdraw all of a user's tokens to `claimManager` after deposits have stopped. Each token in the `allWhitelistedTokens` array is transferred in an iterating loop.

In this case, all token transfers must be successful to withdraw tokens. If any token transfer fails during the withdrawal process, none of the tokens can be withdrawn.

Additionally, `withdrawTokens` is the only method to withdraw tokens from the contract. This means that the entire fund could be locked in the contract if `withdrawTokens` does not work correctly.

```
function withdrawTokens(
    address _userAddress
)
    external
    nonReentrant
    whenNotPaused
    onlyClaimManager
    onlyAfterDepositStop
    returns (uint256[] memory withdrawnAmounts)
{
    // ...

    for (uint256 i; i < allWhitelistedTokens.length; ) {
        // ...

        IERC20 token = IERC20(allWhitelistedTokens[i]);
        token.safeTransfer(claimManager, userBalance);

        unchecked {
            ++i;
        }
    }
    // ...
}
```

Impact

If the transfer of one of the `allWhitelistedTokens` is reverted — for example, if the token is paused by its owner, the `withdrawTokens` function will not work correctly. In this case, all tokens could be locked in the contract.

Recommendations

Consider implementing a function that allows users to withdraw each token individually from the contract. This approach could prevent a partial malfunction in a single token from locking up a user's entire funds.

Remediation

This issue has been acknowledged by Inference Labs, and a fix was implemented in commit `380f977a` ↗.

4. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the 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: OmronDeposit.sol

Function: `addWhitelistedToken(address _tokenAddress)`

This function is used to add a new deposit token to the contract.

Inputs

- `_tokenAddress`
    - **Control**: Arbitrary.
    - **Constraints**: Not zero address.
    - **Impact**: Address of the token to be added.

Branches and code coverage

Intended branches

- Updates the `whitelistedTokens` mapping.
    - ☑ Test coverage

Negative behavior

- Reverts if the token address is zero.
    - ☑ Negative test
- Reverts if the caller is not the owner.
    - ☑ Negative test

Function: `calculatePoints(address _userAddress)`

This function is used to calculate the points earned by a user.

Inputs

- `_userAddress`
    - **Control**: Arbitrary.

    - **Constraints**: None.
    - **Impact**: Address of the user to calculate the points for.

Branches and code coverage

Intended branches

- Returns the total points earned by the user.
  - ☑ Test coverage

Function: `claim(address _userAddress)`

This function is used to claim all points for the user — returns the number of points claimable by the user.

Inputs

- `_userAddress`
  - **Control**: Arbitrary.
  - **Constraints**: Not zero address.
  - **Impact**: Address of the user to claim for.

Branches and code coverage

Intended branches

- Updates the user's points information by invoking `_updatePoints`.
  - ☑ Test coverage
- Returns the number of points claimable by the user and sets the user's point balance to zero.
  - ☑ Test coverage

Negative behavior

- Reverts if reentrancy is detected.
  - ☑ Negative test
- Reverts if the contract is paused.
  - ☑ Negative test
- Reverts if the caller is not the claim manager.
  - ☑ Negative test
- Reverts if the deposit stop time has not been set.
  - ☑ Negative test

Function: `deposit(address _tokenAddress, uint256 _amount)`

This function is used to deposit a token into the contract.

Inputs

- `_tokenAddress`
    - **Control**: Arbitrary.
    - **Constraints**: Only whitelisted token.
    - **Impact**: Address of the token to be deposited.
- `_amount`
    - **Control**: Arbitrary.
    - **Constraints**: Nonzero amount.
    - **Impact**: Amount of the token to be deposited.

Branches and code coverage

Intended branches

- Updates the user's points information by invoking `_updatePoints`.
    - ☑ Test coverage
- Updates the user's token balance and points per hour.
    - ☑ Test coverage
- Transfers the token from the caller to the contract.
    - ☑ Test coverage

Negative behavior

- Reverts if reentrancy is detected.
    - ☑ Negative test
- Reverts if the contract is paused.
    - ☑ Negative test
- Reverts if the deposit stop time has been set.
    - ☑ Negative test
- Reverts if the amount is zero.
    - ☑ Negative test
- Reverts if the token is not whitelisted.
    - ☑ Negative test
- Reverts if the token transfer fails.
    - ☑ Negative test

Function: `getAllWhitelistedTokens()`

This function is used to get the list of all whitelisted tokens.

Branches and code coverage

Intended branches

- Returns the list of all whitelisted tokens.
  - ☑ Test coverage

Function: `getUserInfo(address _userAddress)`

This function is used to get point information about the provided address.

Inputs

- `_userAddress`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the user to check the point information for.

Branches and code coverage

Intended branches

- Returns the point information of the user.
  - ☑ Test coverage

Function: `pause()`

This function is used to pause the contract.

Branches and code coverage

Intended branches

- Sets the contract to a paused state.
  - ☑ Test coverage

Negative behavior

- Reverts if the caller is not the owner.
  - ☑ Negative test
- Reverts if the contract is already paused.
  - ☑ Negative test

Function: `setClaimManager(address _newClaimManager)`

This function is used to set the address of the contract, which is allowed to claim points or withdraw tokens on behalf of users.

Inputs

- `_newClaimManager`
    - **Control**: Arbitrary.
    - **Constraints**: Not zero address.
    - **Impact**: Address of the `claimManager`.

Branches and code coverage

Intended branches

- Updates the `claimManager` variable.
    - ☑ Test coverage

Negative behavior

- Reverts if the new claim-manager address is zero.
    - ☑ Negative test
- Reverts if the caller is not the owner.
    - ☑ Negative test

Function: `stopDeposits()`

This function is used to end the deposit period. If this function is called, the deposit stop time will be set to the current block time.

Branches and code coverage

Intended branches

- Updates the `depositStopTime` variable as the current block time.
    - ☑ Test coverage

Negative behavior

- Reverts if the deposit stop time has already been set.
    - ☑ Negative test
- Reverts if the caller is not the owner.
    - ☑ Negative test

Function: `tokenBalance(address _userAddress, address _tokenAddress)`

This function is used to get the token balance for a user.

Inputs

- `_userAddress`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the user to check the token balance for.
- `_tokenAddress`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the token to check the balance for.

Branches and code coverage

Intended branches

- Returns the token balance of the user for the specified token.
  - ☑ Test coverage

Function: `unpause()`

This function is used to unpause the contract.

Branches and code coverage

Intended branches

- Sets the contract to an unpaused state.
  - ☑ Test coverage

Negative behavior

- Reverts if the caller is not the owner.
  - ☑ Negative test
- Reverts if the contract is already unpaused.
  - ☑ Negative test

Function: `withdrawTokens(address _userAddress)`

This function is used to withdraw all tokens of a user from the contract.

Inputs

- `_userAddress`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Address of the user to withdraw the tokens from.

Branches and code coverage

Intended branches

- Updates the user's points information by invoking `_updatePoints`.
    - ☑ Test coverage
- Iterates over all whitelisted tokens, updates the user's token balances to zero, and transfers the tokens to the `claimManager`.
    - ☑ Test coverage
- Updates the user's points per hour to zero.
    - ☑ Test coverage

Negative behavior

- Reverts if reentrancy is detected.
    - ☑ Negative test
- Reverts if the contract is paused.
    - ☑ Negative test
- Reverts if the caller is not the claim manager.
    - ☑ Negative test
- Reverts if the deposit stop time has not been set.
    - ☑ Negative test
- Reverts if the user address is zero.
    - ☑ Negative test

Function: `_calculatePointsDiff(UserInfo _user)`

This function is used to calculate the points earned by a user between their last updated timestamp and the current block timestamp or the deposit stop time, whichever comes first.

Inputs

- `_user`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Address of the user to calculate the points for.

Branches and code coverage

Intended branches

- Returns zero if the user has not deposited, the user is not earning any points per hour, or the last updated timestamp is later than the deposit stop time.
  - ☑ Test coverage
- Returns the number of points earned by the user.
  - ☑ Test coverage

Function: `_updatePoints(UserInfo _user)`

This function is used to update points information for a user. It calculates the points earned by the user based on the amount of time the tokens are held in the contract.

Inputs

- `_user`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: User to update the points for.

Branches and code coverage

Intended branches

- Updates the user's point balance by invoking `_calculatePointsDiff`.
  - ☑ Test coverage
- Updates the user's last updated timestamp.
  - ☑ Test coverage

5. Assessment Results

At the time of our assessment, the reviewed code was not deployed to the Ethereum Mainnet.

During our assessment on the scoped Omron contracts, we discovered two findings, both of which were medium impact. Inference Labs 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 version reviewed during 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. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

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.