Zellic

NativeOFTWithFee

Smart Contract Security Assessment

June 30, 2023

Prepared for:

LayerZero

Prepared by:

Syed Faraz Abrar and Nipun Gupta

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 LayerZero from June 27th to June 30th, 2023. During this engagement, Zellic reviewed NativeOFTWithFee'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:

- How does the addition of fees affect the security of the protocol?
- Are fee calculations done correctly?
- Were any vulnerabilities introduced since our previous audit?

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.

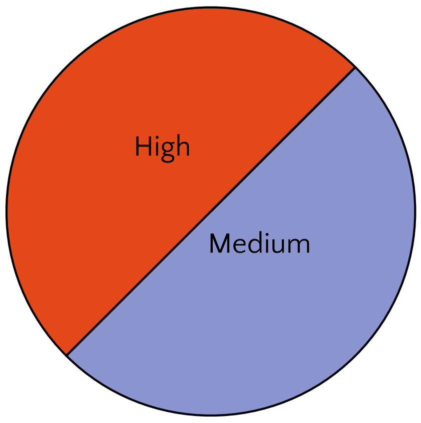
1.3 Results

During our assessment on the scoped NativeOFTWithFee contracts, we discovered two findings. No critical issues were found. One was of high impact and one was of medium impact.

Additionally, Zellic recorded its notes and observations from the assessment for LayerZero's benefit in the Discussion section (4) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	1
Medium	1
Low	0
Informational	0



2 Introduction

2.1 About NativeOFTWithFee

NativeOFTWithFee is LayerZero's example implementation of a cross-chain wrapped native token. On the source chain, where NativeOFTWithFee is deployed, users can deposit the native tokens and receive NativeOFTWithFee at a 1:1 ratio, withdraw the native tokens back for NativeOFTWithFee at a 1:1 ratio, and send NativeOFTWithFee tokens to a different LayerZero-supported chain. This version allows for fees.

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 standards. 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:

NativeOFTWithFee Contracts

Repository	https://github.com/LayerZero-Labs/solidity-examples
Version	solidity-examples: `16f8e645325bfedf6270367f59b709e46782c30a`
Programs	• NativeOFTWithFee • OFTWithFee • BaseOFTWithFee • Fee
Type	Solidity
Platform	EVM-compatible

2.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of six person-days. The assessment was conducted over the course of three calendar days.

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:

Syed Faraz Abrar, Engineer
faith@zellic.io

Nipun Gupta, Engineer
nipun@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

June 27, 2023	Kick-off call
June 27, 2023	Start of primary review period
June 30, 2023	End of primary review period

3 Detailed Findings

3.1 Fee can be set to 100%

- **Target**: Fee
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: High
- **Impact**: High

Description

The setDefaultFeeBp and setFeeBp functions allow the owner to adjust the fee to a maximum value of 100%. This means that the entire amount of the transaction will be transferred to the designated feeOwner. However, if the owner's account is compromised, this could result in the loss of user funds when attempting to transfer funds from the contract.

```
function setDefaultFeeBp(uint16 _feeBp) public virtual onlyOwner {
    require(_feeBp ≤ BP_DENOMINATOR, "Fee: fee bp must be ≤
    BP_DENOMINATOR");
    defaultFeeBp = _feeBp;
    emit SetDefaultFeeBp(defaultFeeBp);
}

function setFeeBp(uint16 _dstChainId, bool _enabled, uint16 _feeBp)
    public virtual onlyOwner {
    require(_feeBp ≤ BP_DENOMINATOR, "Fee: fee bp must be ≤
    BP_DENOMINATOR");
    chainIdToFeeBps[_dstChainId] = FeeConfig(_feeBp, _enabled);
    emit SetFeeBp(_dstChainId, _enabled, _feeBp);
}
```

Impact

If the owner's account is compromised, it allows an attacker to set the fee to 100%, resulting in the full amount of user funds being transferred to the designated feeOwner whenever a transaction is made. This effectively bypasses the intended fee distribution mechanism and leads to a loss of user funds.

Recommendations

Consider setting a reasonable limit on the maximum fee percentage that can be set by the owner.

Remediation

This issue has been acknowledged by LayerZero.

LayerZero's response is included below:

> Since its an admin feature, its up to the multisig to set whatever fee they want to collect. To be as generic and least-opinionated as possible, we simply allow any value.

3.2 Ownership transfer should be two-step

- **Target**: Fee
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: Medium
- **Impact**: Medium

Description

The `setFeeOwner` function allows the modification of the `feeOwner` by specifying a new address. However, it is possible that an erroneous input of an incorrect address can lead to the `feeOwner` being updated to an unintended address.

```
function setFeeOwner(address _feeOwner) public virtual onlyOwner {
    require(_feeOwner ≠ address(0x0), "Fee: feeOwner cannot be 0x");
    feeOwner = _feeOwner;
    emit SetFeeOwner(_feeOwner);
}
```

Impact

The fee amount will be transferred to an undesired destination, and subsequent changes to the ownership may become impossible.

Recommendations

It is better to use a two-step method to change the fee owner, similar to how it is implemented in OpenZeppelin's Ownable2Step.

Remediation

This issue has been acknowledged by LayerZero.

LayerZero's response is included below:

> The multisig can set it to whatever it wants. It's up to them to make sure its set properly. in the case they set it wrongly, they can set it again.

4 Patch Review

The patch added a new contract named NativeOFTWithFee that adds a new fee mechanism to NativeOFTV2. The contract NativeOFTWithFee extends the following contracts: OFTWithFee, BaseOFTWithFee, OFTCoreV2, and Fee.

NativeOFTWithFee contract changes

1. Contract name and inheritance:
    - The contract name has been updated from **NativeOFTV2** to **NativeOFTWithFee**.
    - The contract now inherits **OFTWithFee** instead of **OFTV2**.
2. Constructor:
    - The constructor now calls the constructor of **OFTWithFee** instead of **OFTV2**.
3. Functions `_send` and `_sendAndCall`:
    - These functions are removed from the **NativeOFTWithFee** contract and are moved to the **BaseOFTWithFee** contract.
4. Require messages:
    - The require messages are updated to reflect the name of the new contract.

OFTWithFee contract changes

1. Contract name and inheritance:
    - The contract name has been updated from **OFTV2** to **OFTWithFee**.
    - The contract now inherits **BaseOFTWithFee** instead of **BaseOFTV2**.
2. Require messages:
    - The require messages are updated to reflect the name of the new contract.

BaseOFTWithFee contract changes

The new contract **BaseOFTWithFee** is derived from the original contract **BaseOFTV2** and introduces additional functionality related to fee calculation and handling.

1. Contract name and inheritance:

- The contract name has been updated from **BaseOFTV2** to **BaseOFTWithFee**.
- The new contract inherits from **OFTCoreV2**, **Fee**, **ERC-165**, and **IOFTWithFee**, whereas the original contract inherits from **OFTCoreV2**, **ERC-165**, and **IOFTV2**.

2. Functions `_send` and `_sendAndCall`:
   - These functions now include an additional call to **`_payOFTFee`** to calculate and deduct the fee from the **`_amount`** parameter.
3. Interface support:
   - The **`supportsInterface`** function in the new contract includes a check for the **IOFTWithFee** interface.
4. Internal function override:
   - The new contract overrides the **`_transferFrom`** function, which is defined in both **Fee** and **OFTCoreV2** contracts.

Fee contract

The **Fee** contract is an abstract contract that handles fee calculations and management. Here is a brief review of the contract:

1. Constant and variables:
   - The contract defines a constant **`BP_DENOMINATOR`** with a value of 10,000. It represents the basis points (BP) denominator, which is 10,000 for 100%.
   - The contract has several state variables:
     - **`chainIdToFeeBps`**: A mapping associating destination chain IDs with fee configurations.
     - **`defaultFeeBp`**: The default fee BP when no specific fee is set for a chain.
     - **`feeOwner`**: The address that receives the fees. It defaults to the contract owner.
2. Events:
   - The contract emits three events:
     - **`SetFeeBp`**: Triggered when the fee BP and enabled status are set for a destination chain.
     - **`SetDefaultFeeBp`**: Triggered when the default fee BP are set.
     - **`SetFeeOwner`**: Triggered when the fee owner address is set.
3. Public functions:

- `setDefaultFeeBp`: Allows the contract owner to set the default fee BP.
- `setFeeBp`: Allows the contract owner to set the fee BP and enabled status for a chain.
- `setFeeOwner`: Allows the contract owner to set the address that receives the fees.
- `quoteOFTFee`: Calculates the fee amount for a given destination chain ID and transfer amount.
- `_payOFTFee`: Calculates the fee amount and deducts it from a given transfer amount.

5 Audit Results

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

During our assessment on the scoped NativeOFTWithFee contracts, we discovered two findings. No critical issues were found. One was of high impact and one was of medium impact. LayerZero 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.