Zellic

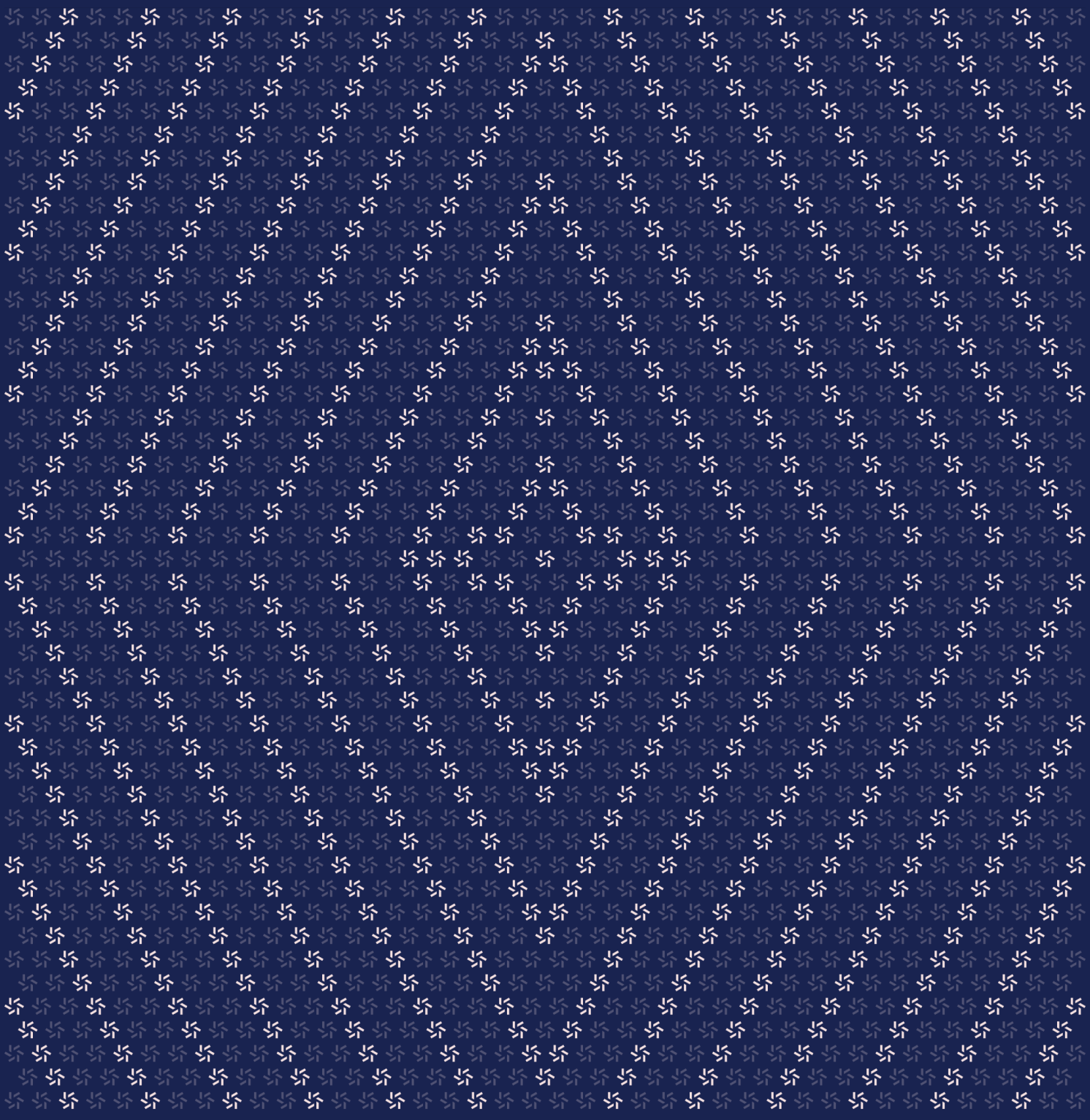
Prepared for
Philip Zudemberg
StakeKit

Prepared by
Jisub Kim
Jaeeu Kim
Zellic

July 12, 2024

StakeKit FeeWrapper

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 StakeKit from July 11th to July 12th, 2024. During this engagement, Zellic reviewed StakeKit FeeWrapper's code for security vulnerabilities, design issues, and general weaknesses in security posture.

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:

- Could a malicious user steal funds from the contracts?
- Could a malicious user lock up users' funds?
- Could a malicious user brick the contracts?

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.

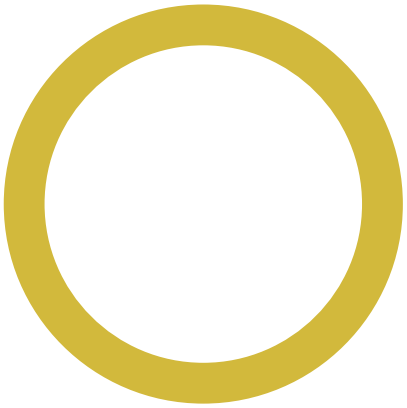
1.4. Results

During our assessment on the scoped StakeKit FeeWrapper contracts, we discovered three findings, all of which were medium impact.

Additionally, Zellic recorded its notes and observations from the assessment for StakeKit's benefit in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	3
Low	0
Informational	0



2. Introduction

2.1. About StakeKit FeeWrapper

StakeKit contributed the following description of StakeKit FeeWrapper:

> StakeKit is a powerful API & widget for self-custodial staking. It supports the most popular tokens and chains out of the box.

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.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

StakeKit FeeWrapper Contracts

Type	Solidity
Platform	EVM-compatible
Target	contracts
Repository	https://github.com/stakekit/contracts ↗
Version	`506e0b338ef15c5994c32e711f07bcbd04b8288d`
Programs	`FeeWrapper4626.sol` `FeeManager`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of two person-days. The assessment was conducted by two consultants over the course of two 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:

Jisub Kim
Engineer
jisub@zellic.io

Jaeeu Kim
Engineer
jaeeu@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

July 11, 2024	Start of primary review period
July 12, 2024	End of primary review period

3. Detailed Findings

3.1. The `transferFrom` function could fail

Target	FeeWrapper4626.sol		
Category	Business Logic	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The FeeWrapper4626 vault currently uses the `transferFrom` function to transfer ERC-20 tokens. However, some nonstandard ERC-20 tokens, such as USDT, do not return a success status from their `transferFrom` function. This deviation from the standard EIP-20 implementation can lead to unexpected behavior in the protocol.

Impact

Incompatibility with some nonstandard tokens.

Recommendations

We recommend using OpenZeppelin's SafeERC20 ↗ versions with the `safeTransfer` and `safeTransferFrom` functions that handle the return-value check, as well as non–standard-compliant tokens.

Remediation

This issue has been acknowledged by StakeKit. The issue was fixed with commit `c14f8456` ↗. They now handle nonstandard ERC20 tokens using OpenZeppelin's `SafeERC20` Library.

3.2. The `approve` function could fail

Target	FeeWrapper4626.sol		
Category	Business Logic	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

In the `deposit` function of FeeWrapper4626, the result of `approve` is checked before proceeding with the deposit. For nonstandard ERC-20 tokens, such as USDT, the `approve` function may not return a success status, which could cause the protocol to malfunction.

Additionally, some nonstandard ERC-20 tokens, including USDT, require a two-step approval process to prevent race conditions with allowances as below:

1. nonzero -> 0
2. 0 -> amount

This means a single approve call may not change the allowance.

Impact

This means incompatibility with some nonstandard tokens.

Recommendations

We recommend using OpenZeppelin's SafeERC20 versions with the `safeIncreaseAllowance` and `safeDecreaseAllowance` functions that handle the return-value check, as well as non–standard-compliant tokens.

Remediation

This issue has been acknowledged by StakeKit. The issue was fixed with commit `c14f8456`. They now handle nonstandard ERC20 tokens using OpenZeppelin's `SafeERC20` Library.

3.3. Rounding errors in computing fee

Target	FeeManager.sol		
Category	Business Logic	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

Due to Solidity's lack of native support for floating-point arithmetic, the results of computing fees may be inconsistent.

```
function computeFee(uint256 amount, FeeConfig memory feeConfig)
    public pure returns (uint256) {
    uint16 fee = feeConfig.fee;
    uint256 feeAmount = (amount * fee) / 10_000;
    return feeAmount;
}
```

For instance, when `amount` <= 9999 and `fee` is 1, the `feeAmount` would be 0.

```
uint256 amount = 9999;
uint16 fee = 1;
uint256 feeAmount = (amount*fee) / 10_000; // would be 0
```

Impact

Inconsistency in fee calculations can lead to incorrect calculation.

Recommendations

Add a minimum deposit limit, or consider having fees accumulate in the contract and having a `payout` function.

Remediation

This issue has been acknowledged by StakeKit. The issue was fixed with commit `c14f8456`↗. They revert if the `feeAmount` is less than 1 wei to enforce a minimum deposit.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Use of ReentrancyGuard is recommended

Currently, reentrancy seems hard in the reviewed codebase. However, to prevent potential future threats, it is recommended to use OpenZeppelin's ReentrancyGuard ↗ to lock the calling functions.

The StakeKit implemented `ReentrancyGuard` in this commit `c14f8456` ↗ to prevent potential reentrancy attack in deposit function.

4.2. No proxy pattern for upgrades

The contract does not implement any proxy patterns for upgrades, which is a deliberate decision by the team. The team has responded stating that they have no plans to upgrade the contract, explaining the absence of upgrade mechanisms.

4.3. No test coverage for nonstandard ERC-20 tokens

The contract lacks test cases for nonstandard ERC20 tokens such as USDT. This oversight could lead to unexpected behavior when interacting with these tokens. Implementing compatibility tests for various token types, including nonstandard ones, is recommended to ensure the contract's reliability and functionality across different token implementations. Such testing would enhance the contract's robustness and user confidence by verifying its ability to handle a wide range of tokens securely.

5. 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.

5.1. Module: FeeManager.sol

Function: `addFeeConfig(address depositContract, address depositToken, uint16 fee, bool enabled)`

This function is used to add a new fee configuration without a fee recipient. This function can only be called by the contract owner.

Inputs

- `depositContract`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the deposit contract.
- `depositToken`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the deposit token.
- `fee`
  - **Control**: Arbitrary.
  - **Constraints**: Value between zero and 10,000.
  - **Impact**: Fee in basis points.
- `enabled`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Boolean value to determine if the fee configuration is enabled.

Branches and code coverage

Intended branches

- Update the fee configuration.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the contract owner.
  - ☑ Negative test
- Revert if the fee is greater than 10,000.
  - ☑ Negative test

Function: `addFeeConfig(address depositContract, address depositToken, uint16 fee, bool enabled, address feeRecipient, uint16 providerFee)`

This function is used to add a new fee configuration with a fee recipient. This function can only be called by the contract owner.

Inputs

- `depositContract`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the deposit contract.
- `depositToken`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the deposit token.
- `fee`
  - **Control**: Arbitrary.
  - **Constraints**: Value between zero and 10,000.
  - **Impact**: Fee in basis points.
- `enabled`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Boolean value to determine if the fee configuration is enabled.
- `feeRecipient`
  - **Control**: Arbitrary.
  - **Constraints**: Nonzero.
  - **Impact**: Address of the fee recipient.
- `providerFee`
  - **Control**: Arbitrary.
  - **Constraints**: Value between zero and 10,000.
  - **Impact**: Provider's share of the fee in basis points.

Branches and code coverage

Intended branches

- Update the fee configuration.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the contract owner.
  - ☑ Negative test
- Revert if the fee is greater than 10,000.
  - ☑ Negative test
- Revert if the provider fee is greater than 10,000.
  - ☑ Negative test
- Revert if the fee-recipient address is invalid.
  - ☑ Negative test

Function: `computeFeeSplit(uint256 feeAmount, FeeConfig feeConfig)`

This function is used to compute the fee split between the fee recipient and the provider.

Inputs

- `feeAmount`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Total fee amount, which is split between the fee recipient and the provider.
- `feeConfig`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Fee configuration.

Branches and code coverage

Intended branches

- Calculate the fee split.
  - ☑ Test coverage

Function: `computeFee(uint256 amount, FeeConfig feeConfig)`

This function is used to compute the fee amount based on the deposit amount and fee configuration.

Inputs

- `amount`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Value of the amount that the fee is computed on.
- `feeConfig`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Fee configuration.

Branches and code coverage

Intended branches

- Calculate the fee amount.
  - ☑ Test coverage

Function: `removeFeeConfig(address depositContract, address depositToken)`

This function is used to remove a fee configuration. This function can only be called by the contract owner.

Inputs

- `depositContract`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the deposit contract.
- `depositToken`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the deposit token.

Branches and code coverage

Intended branches

- Remove the fee configuration.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the contract owner.
  - ☑ Negative test
- Revert if the fee configuration does not exist.
  - ☑ Negative test

Function: `setEnabled(address depositContract, address depositToken, bool enabled)`

This function is used to set the enabled status of a fee configuration. This function can only be called by the contract owner.

Inputs

- `depositContract`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the deposit contract.
- `depositToken`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the deposit token.
- `enabled`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: New enabled status.

Branches and code coverage

Intended branches

- Update the enabled status.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the contract owner.
  - ☑ Negative test
- Revert if the fee configuration does not exist.
  - ☑ Negative test

Function: `setFeeRecipient(address depositContract, address depositToken, address feeRecipient)`

This function is used to set the fee recipient of a fee configuration. This function can only be called by the contract owner.

Inputs

- `depositContract`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the deposit contract.
- `depositToken`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the deposit token.
- `feeRecipient`
  - **Control**: Arbitrary.
  - **Constraints**: Nonzero.
  - **Impact**: Address of the fee recipient.

Branches and code coverage

Intended branches

- Update the fee recipient.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the contract owner.
  - ☑ Negative test
- Revert if the fee configuration does not exist.
  - ☑ Negative test

Function: `setFee(address depositContract, address depositToken, uint16 fee)`

This function is used to set the fee for a specific fee configuration. This function can only be called by the contract owner.

Inputs

- `depositContract`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the deposit contract.
- `depositToken`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the deposit token.
- `fee`
  - **Control**: Arbitrary.
  - **Constraints**: Value between zero and 10,000.
  - **Impact**: New fee in basis points.

Branches and code coverage

Intended branches

- Update the fee.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the contract owner.
  - ☑ Negative test
- Revert if the fee configuration does not exist.
  - ☑ Negative test
- Revert if the fee is greater than 10,000.
  - ☑ Negative test

Function: `setProviderFeeRecipient(address newProviderFeeRecipient)`

This function is used to set a new provider-fee recipient. This function can only be called by the contract owner.

Inputs

- `newProviderFeeRecipient`
  - **Control**: Arbitrary.
  - **Constraints**: Nonzero.
  - **Impact**: Address of the new provider-fee recipient.

Branches and code coverage

Intended branches

- Update the provider-fee recipient.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the contract owner.
  - ☑ Negative test

Function: `setProviderFee(address depositContract, address depositToken, uint16 providerFee)`

This function is used to set the provider's share of the fee in a fee configuration. This function can only be called by the contract owner.

Inputs

- `depositContract`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the deposit contract.
- `depositToken`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the deposit token.
- `providerFee`
  - **Control**: Arbitrary.
  - **Constraints**: Value between zero and 10,000.
  - **Impact**: Provider's share of the fee in basis points.

Branches and code coverage

Intended branches

- Update the provider fee.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the contract owner.
  - ☑ Negative test
- Revert if the provider fee is greater than 10,000.

- ☑ Negative test
- Revert if the fee configuration does not exist.
  - ☑ Negative test

5.2. Module: FeeWrapper4626.sol

Function: `deposit(address depositContract, address depositToken, uint256 amount)`

This function is used to deposit tokens into the target ERC-4626 vault and collect fees.

Inputs

- `depositContract`
  - **Control**: Arbitrary.
  - **Constraints**: Mapped in `feeConfigs`.
  - **Impact**: Address of the ERC-4626 vault.
- `depositToken`
  - **Control**: Arbitrary.
  - **Constraints**: Mapped in `feeConfigs`.
  - **Impact**: Address of the ERC-20 token to deposit.
- `amount`
  - **Control**: Arbitrary.
  - **Constraints**: Nonzero.
  - **Impact**: Amount of tokens to deposit.

Branches and code coverage

Intended branches

- Invoke `computeFee` to calculate the fee with amount.
  - ☑ Test coverage
- Transfer the fee to the `feeRecipient` if `feeRecipient` exists.
  - ☑ Test coverage
- Transfer the fee to the `providerFeeRecipient`.
  - ☑ Test coverage
- Receive the deposit amount from `msg.sender`.
  - ☑ Test coverage
- Call approve to `depositContract` with `depositAmount`.
  - ☑ Test coverage
- Call `depositContractInstance.deposit` with `depositAmount` and `msg.sender`.
  - ☑ Test coverage

Negative behavior

- Revert if `amount` is zero.
  - ☑ Negative test
- Revert if `depositContract` and `depositToken` are not enabled in `feeConfigs`.
  - ☑ Negative test
- Revert if the fee transfer to `feeRecipient` fails.
  - ☑ Negative test
- Revert if the provider-fee transfer fails.
  - ☑ Negative test
- Revert if the transfer deposit amount fails.
  - ☑ Negative test
- Revert if the deposit token approval fails.
  - ☑ Negative test

Function call analysis

- `depositTokenInstance.transferFrom(msg.sender, feeConfig.feeRecipient, feeRecipientAmount)`
  - **What is controllable?** `feeRecipientAmount` based on the user's deposit amount.
  - **If the return value is controllable, how is it used and how can it go wrong?** It could be executed if the transfer fails.
- `depositTokenInstance.transferFrom(msg.sender, this.providerFeeRecipient, feeAmount)`
  - **What is controllable?** `feeAmount` based on the user's deposit amount.
  - **If the return value is controllable, how is it used and how can it go wrong?** It could be executed if the transfer fails.
- `depositTokenInstance.transferFrom(msg.sender, address(this), depositAmount)`
  - **What is controllable?** `depositAmount` based on the user's deposit amount.
  - **If the return value is controllable, how is it used and how can it go wrong?** It could be executed if the transfer fails.
- `depositTokenInstance.approve(depositContract, depositAmount)`
  - **What is controllable?** `depositAmount` based on the user's deposit amount.
  - **If the return value is controllable, how is it used and how can it go wrong?** Transfer will fail in `depositContractInstance.deposit`.
- `depositContractInstance.deposit(depositAmount, msg.sender)`
  - **What is controllable?** `depositAmount` based on the user's deposit amount.
  - **If the return value is controllable, how is it used and how can it go wrong?** The share could be manipulated to return a different value.

6. Assessment Results

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

During our assessment on the scoped StakeKit FeeWrapper contracts, we discovered three findings, all of which were medium impact.

6.1. 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.