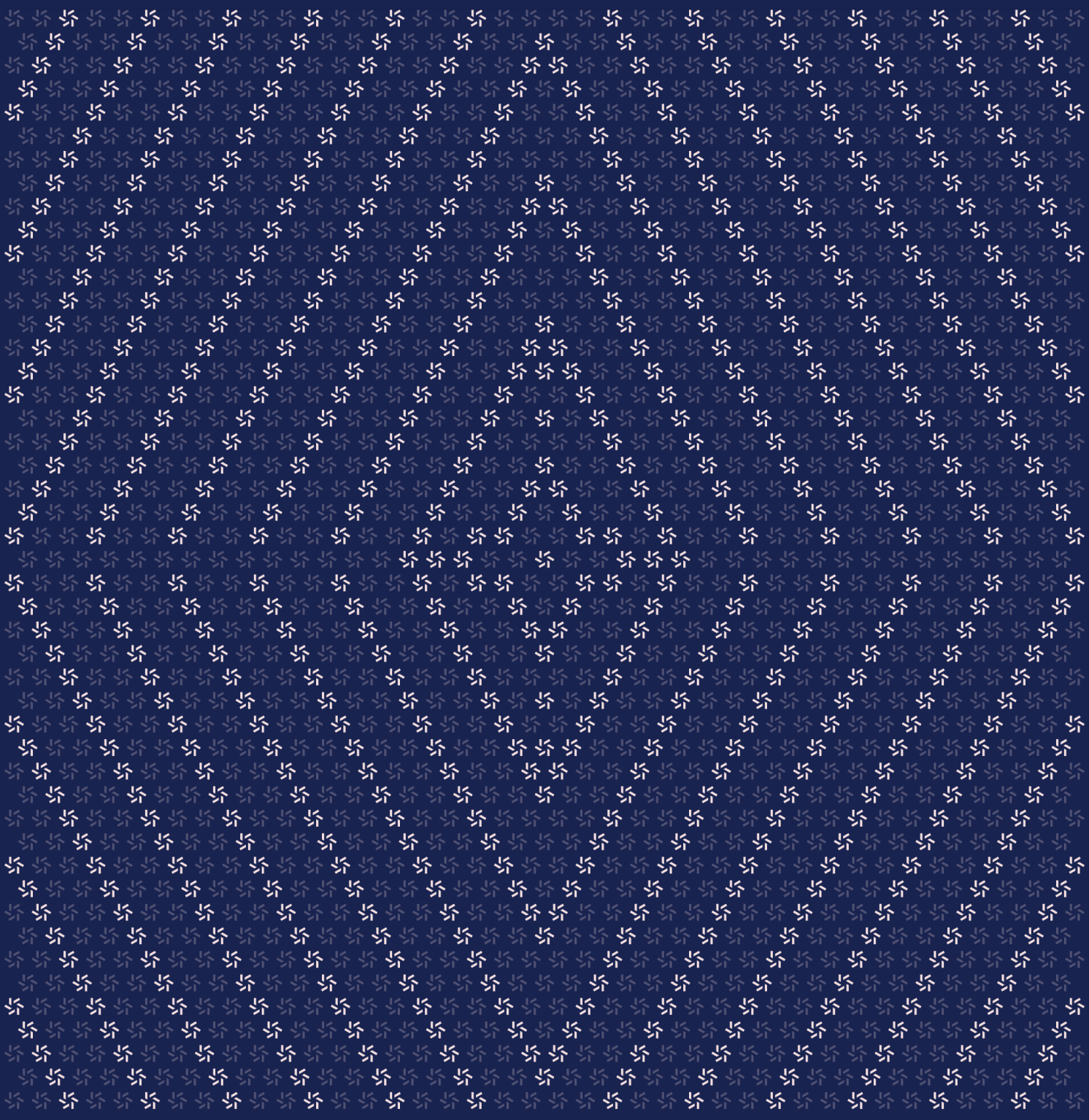
Zellic

Prepared for
StakeKit

Prepared by
Sunwoo Hwang
Doyeon Park
Zellic

March 17, 2025

StakeKit
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 March 5th to March 11th, 2025. During this engagement, Zellic reviewed StakeKit'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:

- Can an attacker steal funds from the vault?
- Is the fee-calculation logic implemented correctly?
- Does the vault have any inflation bugs?
- Are the assets of the vault managed correctly?

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. During this assessment, gaps in response times as well as a lack of developer-oriented documentation and thorough end-to-end testing impacted the momentum of our auditors.

1.4. Results

During our assessment on the scoped StakeKit contracts, we discovered five findings. One critical issue was found. One was of high impact, one was of medium impact, and two were of low impact.

Breakdown of Finding Impacts

Impact Level	Count
Critical	1
High	1
Medium	1
Low	2
Informational	0

2. Introduction

2.1. About StakeKit

StakeKit contributed the following description of StakeKit:

> StakeKit is a self-custodial staking and DeFi API that enables wallets, custodians, and dApps to integrate staking and yield-bearing opportunities seamlessly. It standardizes interactions across multiple protocols, providing a unified interface for actions like stake, unstake, and claim rewards, along with real-time metadata such as APYs, fees, and disclaimers.

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:

StakeKit Contracts

Type	Solidity
Platform	EVM-compatible
Target	contracts
Repository	https://github.com/stakekit/contracts ↗
Version	`ff3423ceb63eeb3216d79744d2c4a63e6c0da8e3`
Programs	`src/allocator-vaults/AllocatorVaultV1.sol`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 1.5 person-weeks. The assessment was conducted by two consultants over the course of one calendar week.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io

Chad McDonald
Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Sunwoo Hwang
Engineer
sunwoo@zellic.io

Doyeon Park
Engineer
doyeon@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

March 5, 2025	Kick-off call
March 5, 2025	Start of primary review period
March 11, 2025	End of primary review period

3. Detailed Findings

3.1. Users' funds can be stolen

Target	AllocatorVaultV1.sol		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

According to ERC-4626, the `redeem` function should deduct an allowance when `msg.sender` is not the `owner`, ensuring that only authorized users can redeem shares on behalf of others. However, in this contract, the redeem function does not enforce any such checks.

As a result:

- The function directly burns shares from the `owner`, regardless of whether `msg.sender` is authorized.
- This allows an attacker to call `redeem` with another user's `owner` address, burning their `shares` and redirecting assets to an arbitrary `receiver`.

```
function redeem(uint256 shares, address receiver, address owner)
    public override nonReentrant returns (uint256) {
    uint256 maxShares = maxRedeem(owner);
    require(shares > 0, "Cannot redeem 0");
    require(shares <= maxShares, "Exceeded max redeem");

    harvest();
    uint256 assets = convertToAssets(shares);
    uint256 _underlying = strategy.convertToAssets(assets);

    _burn(owner, shares); // @audit does not burn caller's shares, can steal
    other users' shares
```

Impact

An attacker can redeem other users' shares.

Recommendations

Add a check to ensure the `msg.sender` is the `owner` or spends allowance from the `owner`.

Remediation

This issue has been acknowledged by StakeKit, and a fix was implemented in commit 22be2310 ↗.

3.2. Fee can be minted multiple times

Target	AllocatorVaultV1.sol		
Category	Business Logic	**Severity**	High
Likelihood	High	**Impact**	High

Description

The `harvest` function is called in `deposit`, `withdraw`, `mint`, and `redeem` functions. In these functions, the `totalUnderlyingStamp` and `lastStamp` are updated at the end of the function.

The `harvest` function can also be called externally for minting a fee to the `feeRecipient`. The `computeHarvestFee` function returns the fee amount based on the time passed since the `lastStamp` and the amount of `totalUnderlyingStamp`. However, the `lastStamp` and `totalUnderlyingStamp` are not updated in the `harvest` function. So the `harvest` function can be called multiple times to mint the fee multiple times.

```
function harvest() public returns (uint256) { // @audit harvest does not update
    the lastStamp and totalUnderlyingStamp.
                                                // so it can mint the newShares
    multiple times.
    uint256 fee = computeHarvestFee();
    uint256 newShares = (totalSupply() + VIRTUAL_SHARES) * fee / (MAX_BPS
    - fee);
    console.log("newShares", newShares);
    if (newShares != 0) {

        _mint(config.feeRecipient, newShares);
    }
    return newShares;
}
```

Impact

An attacker can mint the fee multiple times, inflating the total supply of shares. This abnormal increase affects the exchange rate of shares and assets.

Recommendations

Update both `lastStamp` and `totalUnderlyingStamp` within the `harvest` function to prevent repeated minting of the same fee.

Remediation

This issue has been acknowledged by StakeKit, and a fix was implemented in commit `e01ab9b9`↗.

3.3. Drain and financial loss resulting from rounding issue

Target	AllocatorVaultV1.sol		
Category	Code Maturity	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

In AllocatorVaultV1, the `convertToShares` and `convertToAssets` functions determine the ratio between assets and shares, influencing how deposits and withdrawals are executed:

```
function convertToShares(uint256 assets)
    public view override returns (uint256) {
    return (assets * (totalSupply() + VIRTUAL_SHARES)) / (totalAssets()
    + VIRTUAL_ASSETS);
}

/**
 * @dev Converts an amount of shares to an equivalent amount of assets
 * @param shares Amount of shares to convert
 * @return Amount of assets
 */
function convertToAssets(uint256 shares)
    public view override returns (uint256) {
    return (shares * (totalAssets() + VIRTUAL_ASSETS)) / (totalSupply()
    + VIRTUAL_SHARES);
}
```

However, if the balance between these values becomes skewed, rounding errors can occur, potentially allowing an attacker to exploit these calculations. This could lead to an attacker receiving more shares than intended for a given deposit or withdrawing more assets than they should.

If the value of `totalAssets()` is disproportionally larger than that of `totalSupply()`, an attacker can manipulate the withdrawal process by passing an `_underlying` value that causes the return value to become zero due to an issue in rounding. Consequently, while the submitted `_underlying` value remains unchanged, the return value of `convertToShares` becomes zero. Thus, the attacker can drain assets without providing any shares in return. Such conditions can manifest when the operator supplies initial liquidity.

```
function withdraw(
        uint256 _underlying,
        address receiver,
        address owner
    )
        public
        override
        nonReentrant
        returns (uint256)
    {
        uint256 maxUnderlying = maxWithdraw(owner);
        require(_underlying > 0, "Cannot withdraw 0");
        require(_underlying <= maxUnderlying, "Exceeded max withdraw");

        harvest();

        uint256 assets = strategy.convertToShares(_underlying);
        uint256 shares = convertToShares(assets);

        _burn(msg.sender, shares);
        // [...]
```

If `totalSupply()` is significantly larger than `totalAssets()`, calling the `redeem` function may result in `convertToAssets` returning zero for a given `shares` value. In this case, users would have their shares burned without receiving any assets in return. This issue can occur if excessive fees create a severe imbalance between total shares and assets.

```
function redeem(uint256 shares, address receiver, address owner)
    public override nonReentrant returns (uint256) {
        uint256 maxShares = maxRedeem(owner);
        require(shares > 0, "Cannot redeem 0");
        require(shares <= maxShares, "Exceeded max redeem");

        harvest();
        uint256 assets = convertToAssets(shares);
        uint256 _underlying = strategy.convertToAssets(assets);

        _burn(owner, shares);
        // [...]
```

Impact

This vulnerability poses risks such as the draining of assets on a small scale, financial losses for users, and the potential for funds to be frozen. While the drain may be minor, if the share values of a

nonstandard ERC-4626 represent substantial worth, it could result in severe asset damage. To enhance the security of StakeKit in light of its scalability, it is crucial to consider these security edge cases.

Recommendations

To address the rounding bug, we recommend adopting the rounding rules utilized in ERC-4626 ↗.

Ensure that rounding is done in a way that avoids excess minting or burning of shares. For example, rounding down when minting and rounding up when burning is a safer approach. This way, the user never gets more shares than they should (during deposits) or loses fewer shares than they should (during withdrawals).

```
// ERC4626Upgradeable.sol
    function previewDeposit(uint256 assets)
    public view virtual returns (uint256) {
        return _convertToShares(assets, Math.Rounding.Floor);
    }

    /** @dev See {IERC4626-previewMint}. */
    function previewMint(uint256 shares) public view virtual returns (uint256)
    {
        return _convertToAssets(shares, Math.Rounding.Ceil);
    }

    /** @dev See {IERC4626-previewWithdraw}. */
    function previewWithdraw(uint256 assets)
    public view virtual returns (uint256) {
        return _convertToShares(assets, Math.Rounding.Ceil);
    }

    /** @dev See {IERC4626-previewRedeem}. */
    function previewRedeem(uint256 shares)
    public view virtual returns (uint256) {
        return _convertToAssets(shares, Math.Rounding.Floor);
    }
```

Remediation

This issue has been acknowledged by StakeKit, and a fix was implemented in commit e01ab9b9 ↗.

3.4. Freezing of users' funds due to excessive fee settings

Target	AllocatorVaultV1.sol		
Category	Protocol Risks	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

In AllocatorVaultV1, fees are set through the `configureVault` function. Currently, the `MAX_BPS` is configured as `MAX_BPS = 10_000`, which represents 100%.

```
function configureVault(AllocatorVaultConfig memory _config)
    public onlyRole(DEFAULT_ADMIN_ROLE) {
        require(_config.depositFee <= MAX_BPS, "Deposit fee too high");
        require(_config.performanceFee <= MAX_BPS, "Performance fee too high");
        require(_config.managementFee <= MAX_BPS, "Management fee too high");
        require(_config.feeRecipient != address(0), "Invalid fee recipient");
        config = _config;
    }
```

If fees are set excessively, an attacker can invoke the `harvest` function to significantly increase the `totalSupply()` value. Consequently, users may find themselves unable to proceed with withdrawals, as the exchange rate of assets becomes prohibitively high.

Impact

If the fee value is set excessively high, leading to a substantial increase in the `totalSupply()` value, users may risk having their funds permanently frozen.

Recommendations

To mitigate this risk, consider implementing a more reasonable upper limit for fees, such as capping them at a lower percentage (e.g., 5%). Additionally, requiring multi-signature approval for fee changes can provide an extra layer of security and prevent accidental or malicious fee misconfigurations.

Remediation

This issue has been acknowledged by StakeKit, and a fix was implemented in commit `30f915fc` ↗.

3.5. Precision loss prevents harvesting fees

Target	AllocatorVaultV1.sol		
Category	Protocol Risks	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

In the current implementation, the `computeHarvestFee` function returns a percentage of the fee based on the time elapsed since the last harvest. The fee is calculated as a yearly rate using `10000` as the denominator for the fee calculation. For example, if the fee is `100` (1%) per year, approximately `0.0027%` would be charged per day, as the fee is divided by `SECONDS_IN_YEAR` (31536000).

With this implementation, the time must elapse at least 315360 seconds (approximately 3.65 days) to harvest the minimum fee of 1 basis point. This means that if no one interacts with the vault for this period, only then would the minimum fee be harvested.

```
function computeHarvestFee() public view returns (uint256) {
    uint256 timeElapsed = block.timestamp - lastStamp;
    uint256 mgmtFeeNum = config.managementFee * timeElapsed ;
    uint256 perfFeeNum = config.performanceFee * timeElapsed;
// [...]
    uint256 fee =
        (totalUnderlyingStamp * mgmtFeeNum + gain * perfFeeNum)
    / (currentTotalUnderlying * SECONDS_IN_YEAR);
    return fee;
}
```

Impact

This leads to a precision loss for the short-term fee calculation, preventing the vault from harvesting fees as expected.

Recommendations

We recommend several approaches to address this issue:

1. Restrict the `harvest()` function to be callable only by the owner.

2. Implement a minimum time threshold between harvests (e.g., 1 day) to ensure fee accumulation.

Remediation

This issue has been acknowledged by StakeKit, and fixes were implemented in the following commits:

- 30f915fc ↗
- e13e938e ↗

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: AllocatorVaultV1.sol

Function: `deposit(uint256 _underlying, address receiver)`

When a user deposits assets, the function deposits those assets into another ERC-4626 vault and manages the resulting shares, subsequently minting new shares for the users based on the shares obtained.

Inputs

- `_underlying`
  - **Control**: Arbitrary.
  - **Constraints**: It must be greater than zero.
  - **Impact**: It specifies the amount of assets to be paid.
- `receiver`
  - **Control**: Arbitrary.
  - **Constraints**: It must not be the `address(0)`.
  - **Impact**: It is the address of the wallet receiving the shares.

Branches and code coverage

Intended branches

- Normal minting occurs when an `_underlying` value greater than zero is provided.
  - ☑ Test coverage

Negative behavior

- The transaction reverts when the `_underlying` value is zero.
  - ☑ Negative test

Function call analysis

- `this.strategy.previewDeposit(_underlying)`
    - **What is controllable?** The `_underlying` value.
    - **If the return value is controllable, how is it used and how can it go wrong?** It can still be bypassed even when a value of zero is passed to the `_underlying` parameter.
    - **What happens if it reverts, reenters or does other unusual control flow?** No impact.
- `SafeERC20.safeTransferFrom(IERC20(this.underlying), msg.sender, address(this), _underlying)`
    - **What is controllable?** The `_underlying` value.
    - **If the return value is controllable, how is it used and how can it go wrong?** No impact.
    - **What happens if it reverts, reenters or does other unusual control flow?** If reentrancy is possible, it may lead to an increase in the `totalSupply()` value through repetitive calls to the `harvest` function, potentially resulting in the permanent freezing of funds.
- `IERC20(this.underlying).approve(address(this.strategy), _underlying)`
    - **What is controllable?** The `_underlying` value.
    - **If the return value is controllable, how is it used and how can it go wrong?** No impact.
    - **What happens if it reverts, reenters or does other unusual control flow?** While a reentrancy scenario is possible, it may not provide sufficient incentive for an attack, as funds are paid in advance.
- `this.strategy.deposit(_underlying, address(this))`
    - **What is controllable?** The `_underlying` value.
    - **If the return value is controllable, how is it used and how can it go wrong?** By maliciously manipulating the `receivedAssets` value, a substantial number of shares can be minted.
    - **What happens if it reverts, reenters or does other unusual control flow?** While a reentrancy scenario is possible, it may not provide sufficient incentive for an attack, as funds are paid in advance.
- `this.strategy.convertToAssets(this.totalAssets())`
    - **What is controllable?** It is uncontrollable.
    - **If the return value is controllable, how is it used and how can it go wrong?** The fee process can be maliciously manipulated.
    - **What happens if it reverts, reenters or does other unusual control flow?** While a reentrancy scenario is possible, it may not provide sufficient incentive for an attack, as funds are paid in advance.

Function: `harvest()`

This function mints to the `feeRecipient` in accordance with the specified fee rate.

Branches and code coverage

Intended branches

- Anyone from the outside can invoke it to mint fees to the `feeRecipient`.
  - ☐ Test coverage

Function call analysis

- `this.computeHarvestFee() -> this.strategy.convertToAssets(this.totalAssets())`
  - **What is controllable?** It is uncontrollable.
  - **If the return value is controllable, how is it used and how can it go wrong?** Maliciously manipulating the minting value can lead to the permanent freezing of users' assets.
  - **What happens if it reverts, reenters or does other unusual control flow?** No impact.

Function: `mint(uint256 shares, address receiver)`

This function allows for the direct specification of the `shares` value to designate the desired amount of shares to be minted, thereby depositing the corresponding assets into the ERC-4626 vault and facilitating the payment for the received shares.

Inputs

- `shares`
  - **Control**: Arbitrary.
  - **Constraints**: It must be greater than zero.
  - **Impact**: It determines the final minting value of the shares.
- `receiver`
  - **Control**: Arbitrary.
  - **Constraints**: It must not be the `address(0)`.
  - **Impact**: It is the address of the wallet receiving the shares.

Branches and code coverage

Intended branches

- Normal minting occurs when a `shares` value greater than zero is provided.
  - ☑ Test coverage

Negative behavior

- The transaction reverts when the `shares` value is zero.
  - ☑ Negative test

Function call analysis

- `this.previewMint(shares) -> this.previewHarvest() -> this.computeHarvestFee() -> this.strategy.convertToAssets(this.totalAssets())`
  - **What is controllable?** The `shares` value.
  - **If the return value is controllable, how is it used and how can it go wrong?** The value of the assets to be deposited can be maliciously altered.
  - **What happens if it reverts, reenters or does other unusual control flow?** No impact.
- `this.previewMint(shares) -> this.strategy.previewMint(assets)`
  - **What is controllable?** The `shares` value.
  - **If the return value is controllable, how is it used and how can it go wrong?** The value of the assets to be deposited can be maliciously altered.
  - **What happens if it reverts, reenters or does other unusual control flow?** No impact.
- `SafeERC20.safeTransferFrom(IERC20(this.underlying), msg.sender, address(this), _underlying)`
  - **What is controllable?** The `_underlying` value.
  - **If the return value is controllable, how is it used and how can it go wrong?** No impact.
  - **What happens if it reverts, reenters or does other unusual control flow?** If reentrancy is possible, it may lead to an increase in the `totalSupply()` value through repetitive calls to the `harvest` function, potentially resulting in the permanent freezing of funds.
- `IERC20(this.underlying).approve(address(this.strategy), _underlying)`
  - **What is controllable?** The `_underlying` value.
  - **If the return value is controllable, how is it used and how can it go wrong?** No impact.
  - **What happens if it reverts, reenters or does other unusual control flow?**

While a reentrancy scenario is possible, it may not provide sufficient incentive for an attack, as funds are paid in advance.

- `this.strategy.deposit(_underlying, address(this))`
  - **What is controllable?** The `_underlying` value.
  - **If the return value is controllable, how is it used and how can it go wrong?** No impact.
  - **What happens if it reverts, reenters or does other unusual control flow?** While a reentrancy scenario is possible, it may not provide sufficient incentive for an attack, as funds are paid in advance.
- `this.strategy.convertToAssets(this.totalAssets())`
  - **What is controllable?** It is uncontrollable.
  - **If the return value is controllable, how is it used and how can it go wrong?** The fee process can be maliciously manipulated.
  - **What happens if it reverts, reenters or does other unusual control flow?** While a reentrancy scenario is possible, it may not provide sufficient incentive for an attack, as funds are paid in advance.

Function: `redeem(uint256 shares, address receiver, address owner)`

This function allows for the specification of the desired amount of `shares` to be burned directly, followed by the withdrawal of assets from the ERC-4626 vault.

Inputs

- `shares`
  - **Control**: Arbitrary.
  - **Constraints**: It must be greater than zero and less than or equal to the balance of the `owner`.
  - **Impact**: Specifies the amount of shares to be burned.
- `receiver`
  - **Control**: Arbitrary.
  - **Constraints**: A value of type `address`.
  - **Impact**: It is the address of the wallet receiving the assets.
- `owner`
  - **Control**: Arbitrary.
  - **Constraints**: It must not be the `address(0)`.
  - **Impact**: Specifies the address of the wallet that will burn the shares.

Branches and code coverage

Intended branches

- A normal withdrawal occurs when `shares` greater than zero and less than or equal to the owner's balance is provided.
  - ☑ Test coverage

Negative behavior

- The transaction reverts when the `shares` value is zero.
  - ☐ Negative test
- The transaction reverts when the `shares` value is less than the owner's balance.
  - ☐ Negative test

Function call analysis

- `this.strategy.convertToAssets(assets)`
  - **What is controllable?** The `assets` value.
  - **If the return value is controllable, how is it used and how can it go wrong?** No impact.
  - **What happens if it reverts, reenters or does other unusual control flow?** If reentrancy is possible, it may lead to an increase in the `totalSupply()` value through repetitive calls to the `harvest` function, potentially resulting in the permanent freezing of funds.
- `SafeERC20.safeTransfer(IERC20(address(this.strategy)), receiver, assets)`
  - **What is controllable?** The `assets` value and the `receiver` value.
  - **If the return value is controllable, how is it used and how can it go wrong?** No impact.
  - **What happens if it reverts, reenters or does other unusual control flow?** While a reentrancy scenario is possible, it may not provide sufficient incentive for an attack, as shares are burned in advance.
- `this.strategy.redeem(assets, receiver, address(this))`
  - **What is controllable?** The `assets` value and the `receiver` value.
  - **If the return value is controllable, how is it used and how can it go wrong?** No impact.
  - **What happens if it reverts, reenters or does other unusual control flow?** While a reentrancy scenario is possible, it may not provide sufficient incentive for an attack, as shares are burned in advance.
- `this.strategy.convertToAssets(this.totalAssets())`
  - **What is controllable?** It is uncontrollable.

- **If the return value is controllable, how is it used and how can it go wrong?** The fee process can be maliciously manipulated.
- **What happens if it reverts, reenters or does other unusual control flow?** While a reentrancy scenario is possible, it may not provide sufficient incentive for an attack, as shares are burned in advance.

Function: `withdraw(uint256 _underlying, address receiver, address owner)`

This function designates the assets to be withdrawn using the `_underlying` value, facilitating the withdrawal of assets from the ERC-4626 vault.

Inputs

- `_underlying`
  - **Control**: Arbitrary.
  - **Constraints**: It must be greater than zero and less than or equal to the balance of `maxWithdraw(owner)`.
  - **Impact**: Specifies the amount of assets to be received.
- `receiver`
  - **Control**: Arbitrary.
  - **Constraints**: A value of type `address`.
  - **Impact**: It is the address of the wallet receiving the assets.
- `owner`
  - **Control**: Arbitrary.
  - **Constraints**: A value of type `address`.
  - **Impact**: Used only for validating the `_underlying` value.

Branches and code coverage

Intended branches

- A normal withdrawal occurs when `_underlying` greater than zero and less than or equal to the `maxWithdraw(owner)` is provided.
  - ☑ Test coverage

Negative behavior

- The transaction reverts when the `_underlying` value is zero.
  - ☐ Negative test

- The transaction reverts when the `_underlying` value is less than `maxWithdraw(0)`.
    - ☐ Negative test

Function call analysis

- `this.maxWithdraw(owner) -> this.previewRedeem(shares) -> this.previewHarvest() -> this.computeHarvestFee() -> this.strategy.convertToAssets(this.totalAssets())`
    - **What is controllable?** The `owner` value.
    - **If the return value is controllable, how is it used and how can it go wrong?** The range of the `_underlying` value can be extended.
    - **What happens if it reverts, reenters or does other unusual control flow?** No impact.
- `this.strategy.convertToShares(_underlying)`
    - **What is controllable?** The `_underlying` value.
    - **If the return value is controllable, how is it used and how can it go wrong?** The amount of assets received can be maliciously manipulated.
    - **What happens if it reverts, reenters or does other unusual control flow?** If reentrancy is possible, it may lead to an increase in the `totalSupply()` value through repetitive calls to the `harvest` function, potentially resulting in the permanent freezing of funds.
- `SafeERC20.safeTransfer(IERC20(address(this.strategy)), receiver, assets)`
    - **What is controllable?** The `assets` value and `receiver` value.
    - **If the return value is controllable, how is it used and how can it go wrong?** No impact.
    - **What happens if it reverts, reenters or does other unusual control flow?** While a reentrancy scenario is possible, it may not provide sufficient incentive for an attack, as shares are burned in advance.
- `this.strategy.withdraw(_underlying, receiver, address(this))`
    - **What is controllable?** The `_underlying` value and `receiver` value.
    - **If the return value is controllable, how is it used and how can it go wrong?** No impact.
    - **What happens if it reverts, reenters or does other unusual control flow?** While a reentrancy scenario is possible, it may not provide sufficient incentive for an attack, as shares are burned in advance.
- `this.strategy.convertToAssets(this.totalAssets())`
    - **What is controllable?** It is uncontrollable.
    - **If the return value is controllable, how is it used and how can it go wrong?** The fee process can be maliciously manipulated.
    - **What happens if it reverts, reenters or does other unusual control flow?**

While a reentrancy scenario is possible, it may not provide sufficient incentive for an attack, as shares are burned in advance.

5. Assessment Results

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

During our assessment on the scoped StakeKit contracts, we discovered five findings. One critical issue was found. One was of high impact, one was of medium impact, and two were of low impact.

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.