ZUNAMI SMART CONTRACTS SECURITY AUDIT REPORT

OXORIO

FEBRUARY 06, 2024

CONTENTS

1 INTRO

1.1 DISCLAIMER

The audit makes no assertions or warranties about the utility of the code, its security, the suitability of the business model, investment advice, endorsement of the platform or its products, the regulatory regime for the business model, or any other statements about the fitness of the contracts for their intended purposes, or their bug-free status. The audit documentation is for discussion purposes only.

1.2 ABOUT OXORIO

Oxorio is a prominent audit and consulting firm in the blockchain industry, offering top-tier security audits and consulting to organizations worldwide. The company's expertise stems from its active involvement in designing and deploying multiple blockchain projects, wherein it developed and analyzed smart contracts.

With a team of more than six dedicated blockchain specialists, Oxorio maintains a strong commitment to excellence and client satisfaction. Its contributions to several blockchain projects reflect the company's innovation and influence in the industry. Oxorio's comprehensive approach and deep blockchain understanding make it a trusted partner for organizations in the sector.

Contact details:

- oxor.io
- ping@oxor.io
- Github
- Linkedin
- Twitter

1.3 SECURITY ASSESSMENT METHODOLOGY

Several auditors work on this audit, each independently checking the provided source code according to the security assessment methodology described below:

1. Project architecture review

The source code is manually reviewed to find errors and bugs.

2. Code check against known vulnerabilities list

The code is verified against a constantly updated list of known vulnerabilities maintained by the company.

3. Security model architecture and structure check

The project documentation is reviewed and compared with the code, including examining the comments and other technical papers.

4. Cross-check of results by different auditors

The project is typically reviewed by more than two auditors. This is followed by a mutual cross-check process of the audit results.

5. Report consolidation

The audited report is consolidated from multiple auditors.

6. Re-audit of new editions

After the client has reviewed and fixed the issues, these are double-checked. The results are included in a new version of the audit.

7. Final audit report publication

The final audit version is provided to the client and also published on the company's official website.

1.4 FINDINGS CLASSIFICATION

1.4.1 Severity Level Reference

The following severity levels were assigned to the issues described in the report:

- **CRITICAL**: A bug that could lead to asset theft, inaccessible locked funds, or any other fund loss due to unauthorized party transfers.
- **MAJOR**: A bug that could cause a contract failure, with recovery possible only through manual modification of the contract state or replacement.
- **WARNING**: A bug that could break the intended contract logic or expose it to DDoS attacks.
- **INFO**: A minor issue or recommendation reported to or acknowledged by the client's team.

1.4.2 Status Level Reference

Based on the client team's feedback regarding the list of findings discovered by the contractor, the following statuses were assigned to the findings:

- **NEW**: Awaiting feedback from the project team.
- **FIXED**: The recommended fixes have been applied to the project code, and the identified issue no longer affects the project's security.
- **ACKNOWLEDGED**: The project team is aware of this finding. Fixes for this finding are planned. This finding does not affect the overall security of the project.
- **NO ISSUE**: The finding does not affect the overall security of the project and does not violate its operational logic.

1.5 PROJECT OVERVIEW

Zunami is a decentralized protocol that issues aggregated stablecoins, whose collateral is utilized in omnipools and differentiated among various profit-generating strategies. The protocol creates Omni pools and issue zunStables on top of them. Protocol launches two aggregated stablecoins - zunUSD and zunETH.

The Omni pool operates as a Yield Aggregator by providing liquidity to the multiple strategies and reinvesting profits. Each zunStable is backed by its own Omni pool, managed through DAO governance. The DAO manages the addition of new strategies and the rebalancing of funds between strategies.

1.5.1 Documentation

For this audit, the following sources of truth about how the smart contracts should work were used:

- main GitHub repository of the project.

The sources were considered to be the specification. In the case of discrepancies with the actual code behavior, consultations were held directly with the client team.

1.6 AUDIT SCOPE

The scope of the audit includes smart contracts at `contracts` folder except files at `distributor` and `staking` subfolders of the project repository.

The audited commit identifiers:

- initial commit `8bc108201bef8c4d341ecd3a29a3b1d975019cec`
- audit fixes `9ffa8e1b6128d1ade8459a4e492cee669ed241a1`
- reaudit fixes `79892fe12bec407d3d9706c19cd421d458263c0c`

2 FINDINGS REPORT

OXORIO

2.1 CRITICAL

C-01	No access control for `deposit` function call in `VaultStrat`
Severity	CRITICAL
Status	• FIXED

Location

File	Location	Line
VaultStrat.sol	contract `VaultStrat` > function `deposit`	30

Description

The `deposit` function of the `VaultStrat` contract has an `external` visibility modifier and can be called by any address without permissions. An attacker can make a fake deposit to the `VaultStrat` contract by directly calling the `deposit` function without any token value. This leads to incorrect computation of the withdrawal value during withdrawal from the strategy. The process can be exploited as follows:

- Honest users deposit to `VaultStrat` through the controller.
- The attacker makes a low deposit to `VaultStrat` through the controller.
- The attacker makes a fake deposit to `VaultStrat` directly.
- The attacker withdraws all funds from `VaultStrat` through the controller.

```
it('open deposit method in VaultStrat', async () => {
  const {
      alice,
      bob,
      zunamiPool,
      zunamiPoolController,
      strategies,
      usdt,
  } = await loadFixture(deployFixture);

  // add VaultStrat to zunamiPoolController
  const strategy = strategies[0];
```

```
    await zunamiPool.addStrategy(strategy.address);

    await zunamiPoolController.setDefaultDepositSid(0);
    await zunamiPoolController.setDefaultWithdrawSid(0);

    // Alice deposits 100 to VaultStrat
    await expect(
        zunamiPoolController
            .connect(alice)
            .deposit(getMinAmountZunUSD('100'), alice.getAddress())
    ).to.emit(zunamiPool, 'Deposited');

    // Bob deposits 100 to VaultStrat
    await expect(
        zunamiPoolController
            .connect(bob)
            .deposit(getMinAmountZunUSD('100'), bob.getAddress())
    ).to.emit(zunamiPool, 'Deposited');

    // Bob makes fake deposit to VaultStrat
    await strategy.connect(bob).deposit(getMinAmountZunUSD('200'))

    // Bob withdraws 200 from VaultStrat
    let balanceBefore = BigNumber.from(await usdt.balanceOf(bob.getAddress()));
    let sharesAmount = BigNumber.from(
        await zunamiPool.balanceOf(bob.getAddress())
    );
    await zunamiPool.connect(bob).approve(zunamiPoolController.address, sharesAmount);

    await expect(
        zunamiPoolController.connect(bob).withdraw(sharesAmount, [0, 0, 0, 0, 0],
bob.getAddress())
    ).to.emit(zunamiPool, 'Withdrawn');

    expect(
        BigNumber.from(await usdt.balanceOf(bob.getAddress())).sub(balanceBefore)
    ).to.eq(ethers.utils.parseUnits('200', 'mwei'));
  });
```

Recommendation

We recommend adding access control for the `deposit` function.

Update

Fixed in commit `9ffa8e1b6128d1ade8459a4e492cee669ed241a1`.

C-02	Blocking execution of `inflate` and `deflate` functions in `ConvexCurveStratBase`
Severity	CRITICAL
Status	• FIXED

Location

File	Location	Line
[ConvexCurveStratBase.sol]()	contract `ConvexCurveStratBase` > function `depositBooster`	32

Description

In the `depositBooster` function of the `ConvexCurveStratBase` contract, the `allowance` is increased by an amount that may be insufficient for the subsequent call to `depositAll`. This issue arises during the call to [`depositAll`]() in Convex, where a deposit is made for the entire balance of the strategy:

```
uint256 balance = IERC20(lptoken).balanceOf(msg.sender);
deposit(_pid, balance, _stake);
```

This leads to a problem where, if there are LP tokens on the strategy contract, calling the `inflate` and `deflate` functions can result in an error due to insufficient `allowance` in the `depositBooster` function.
Additionally, it is possible to frontrun transactions calling the `inflate` and `deflate` functions, blocking their execution by adding a small amount of LP tokens to the strategy contract.

Recommendation

We recommend considering the replacement of the `depositAll` function call with a call to the `deposit` function, explicitly specifying the `amount`.

Update

Fixed in commit [`9ffa8e1b6128d1ade8459a4e492cee669ed241a1`]().

C-03	Elevated price in USD in the `getLiquidityTokenPrice` function leads to money theft from the pool in `ZunamiStratBase`
Severity	CRITICAL
Status	• FIXED

Location

File	Location	Line
ZunamiStratBase.sol	contract `ZunamiStratBase` > function `deposit`	71

Description

In the `deposit` function of the `ZunamiStratBase` contract, the cost of Curve pool LP tokens in USD is determined based on the current price received from the oracle. This value is then used in the `processSuccessfulDeposit` function for minting shares. The issue arises because a higher current price of the LP token results in a larger share of the total LP tokens for a user, while the price change does not impact the shares of previous users.

In the strategy, within the `deposit` function, liquidity is deposited into the Curve pool, which returns LP tokens. The amount of these LP tokens is set in the `depositedLiquidity` variable. Finally, the USD value of the LP tokens is returned to `zunamiPool`, calculated based on the current price provided by the oracle:

```
function deposit(uint256[POOL_ASSETS] memory amounts) external returns (uint256) {
    // ...
    uint256 liquidity = depositLiquidity(amounts);
    depositedLiquidity += liquidity;
    return calcLiquidityValue(liquidity);
}
```

In `zunamiPool`, shares are minted based on the USD value of this LP tokens. However, these new shares are allocated without considering the USD value of previously minted shares in the pool according to the new price:

```
    // ...
    minted =
        ((totalSupply() + 10 ** _decimalsOffset()) * depositedValue) /
```

```
        (totalDeposited + 1);
}
_mint(receiver, minted - locked);
_strategyInfo[sid].minted += minted;

totalDeposited += depositedValue;
```

For example:
1) The first user deposits `10000` `zunUSD`, the strategy is receiving `10000` LP tokens from the Curve pool. These LP tokens are recorded in the strategy in the `depositedLiquidity` variable. The cost of these LP tokens in USD, as returned from the oracle, is assumed to be 1 USD per token:

```
getLiquidityTokenPrice = 1 (USD)
strategy.depositedLiquidity = 10000 (LP)
zunamiPool.minted = 10000 (shares)
zunamiPool.totalDeposited = 10000 (USD)
```

2) A second user deposits `1000` `zunUSD`, the strategy is receiving `1000` LP tokens. If the price per LP token has risen to `1.2` USD, the `deposit` function returns `1200` USD. The shares in `zunamiPool` remain unchanged due to the LP token price change:

```
getLiquidityTokenPrice = 1.2 (USD)
strategy.depositedLiquidity = 11000 (LP)
zunamiPool.minted = 11200 (shares)
zunamiPool.totalDeposited = 11200 (USD)
```

3) The second user withdraws their `1200` shares. The `calcRatioSafe` function determines the user's claim to be `1200/11200 = 0.107` of `strategy.depositedLiquidity`, equating to `0.107 * 11000 = 1177` LP tokens. Exchanging these tokens in the Curve pool yields `1177` `zunUSD`:

```
strategy.depositedLiquidity = 9823 (LP)
zunamiPool.minted = 10000 (shares)
zunamiPool.totalDeposited = 10000 (USD)
```

As a result, the second user profits `177` `zunUSD` from the `deposit` and `withdraw` functions, causing a loss for the first user.

Recommendation

We recommend adjusting the allocation of new shares during a deposit in `zunamiPool` to consider the current value of shares in USD.

Update

Zunami's response

Fixed in commit [9ffa8e1b6128d1ade8459a4e492cee669ed241a1](9ffa8e1b6128d1ade8459a4e492cee669ed241a1).

Oxorio's response

Made solution not fully fixes bug scenario in case of `ThroughController` usage:
1) The first user deposits `10000` `DAI`, the strategy is receiving `10000` LP tokens from the Curve pool. These LP tokens are recorded in the strategy in the `depositedLiquidity` variable. The cost of these LP tokens in USD, as returned from the oracle, is assumed to be 1 USD per token:

```
    getLiquidityTokenPrice = 1 (USD)
    strategy.depositedLiquidity = 10000 (LP)
    zunamiPool.minted = 10000 (shares)
    zunamiPool.totalDeposited = 10000 (USD)
```

2) A second user deposits `1000` `DAI`, the strategy is receiving `1000` LP tokens. If the price per LP token has risen to `1.2` USD, the `deposit` function returns `1200` USD. Also, the new `extraGains` logic of `zunamiPool` mints `2000` new shares to the pool address:

```
    getLiquidityTokenPrice = 1.2 (USD)
    strategy.depositedLiquidity = 11000 (LP)
    zunamiPool.minted = 13200 = 10000 + 2000 (new shares for pool) + 1200 (new shares for
depositer)
    zunamiPool.totalDeposited = 13200 (USD)
```

3) Let's assume that the price spike of `DAI` was short-lived and after a while, the price returned to `1` dollar. The first user withdraws their `10000` shares. The `calcRatioSafe` function determines the user's claim to be `10000/13200 = 0.758` of `strategy.depositedLiquidity`, equating to `0.758 * 11000 = 8338` LP tokens. Exchanging these tokens in the Curve pool yields `8338` `DAI`:

```
strategy.depositedLiquidity = 2662 (LP)
zunamiPool.minted = 3200 (shares)
zunamiPool.totalDeposited = 3200 (USD)
```

As a result, the first user loses `1662` `DAI` from the `deposit` and `withdraw` function calls.

Also, the new `extraGains` logic mints shares at a [1 to 1 rate](#):

```
uint256 gains = currentTotalHoldings - totalDeposited;
extraGains += gains;
totalDeposited += gains;
extraGainsMintedBlock = block.number;
_mint(address(this), gains);
```

But as the pool contract implements defense from inflation attack, it mints shares in a shifted rate [during deposit](#):

```
minted = ((totalSupply() + 10 ** _decimalsOffset()) * depositedValue) / (totalDeposited + 1);
...
_mint(receiver, minted);
```

We recommend using the same shifted rate for minting in the `extraGains` logic.

Zunami's second response

Fixed in commit `79892fe12bec407d3d9706c19cd421d458263c0c`.

In the current architecture, a protocol has capital stored in strategies. Essentially, investing capital through the pool, the protocol mint own zun stablecoins in return. The capital the protocol held earns the rewards and the yield in the base scenario. Currently, the DAO explicitly withdraws the rewards and converts capital growth into `zun` stablecoins for withdrawal as well. In other words, it's the normal operation mode of the protocol where it constantly gains capital growth. However, in the event of a force majeure and if the protocol has an unsuccessful strategy where the token in which the capital is stored in an external protocol drops in price (for example, Curve LP token), the `DAO` initiate a recapitalization procedure to restore 100% backing selling stacked `zun` stablecoin and collected rewards. In the protocol, the period between losing full backing of own stablecoin with capital and its restoration is a standard procedure that cannot be fixed algorithmically because the problem lies in the external protocol, which has become imbalanced. And yes, users take on the risk that in the event of exiting the zunami pool (omni or APS), they may lose funds if capital is lost in an external project before the recapitalization happens.

C-04	Price-feed returns ETH price in `FrxETHOracle`
Severity	CRITICAL
Status	• FIXED

Location

File	Location	Line
FrxETHOracle.sol	contract `FrxETHOracle`	27

Description

In the contract `FrxETHOracle` the oracle requests the price for `ETH` instead of the price of `frxETH`. `frxETH` peg is defined as 1% on each side of `1.00` exchange rate meaning the `frxETH` exchange rate rests between `1.01-0.99` `ETH` per 1 `frxETH`. In case of depeg, oracle will return incorrect value.

Recommendation

We recommend changing the code to return the correct price of `frxETH`.

Update

Fixed in commit `9ffa8e1b6128d1ade8459a4e492cee669ed241a1`.

2.2 MAJOR

M-01	Latency of APS logic in `CrvUsdApsConvexCurveStratBase`, `FraxApsConvexCurveStratBase`
Severity	MAJOR
Status	• FIXED

Location

File	Location	Line
[CrvUsdApsConvexCurveStratBase.sol]()	contract `CrvUsdApsConvexCurveStratBase` > function `inflate`	90
[CrvUsdApsConvexCurveStratBase.sol]()	contract `CrvUsdApsConvexCurveStratBase` > function `deflate`	125
[FraxApsConvexCurveStratBase.sol]()	contract `FraxApsConvexCurveStratBase` > function `inflate`	91
[FraxApsConvexCurveStratBase.sol]()	contract `FraxApsConvexCurveStratBase` > function `deflate`	121

Description

The functions `inflate` and `deflate` in the contracts `CrvUsdApsConvexCurveStratBase` and `FraxApsConvexCurveStratBase` can only be called by the DAO. The DAO mechanism involves significant latency between the start of voting and the execution of proposals. For instance, if the governance voting period is 7 days, then all APS strategy functions (`inflate` and `deflate`) are executed with a 7-day delay. This could lead to the temporary depegging of the `zunUSD` token.

Recommendation

We recommend implementing an emergency APS mechanism that can be activated without any latency.

Update

Fixed in commit `9ffa8e1b6128d1ade8459a4e492cee669ed241a1`.

M-02	Array out of bounds in `_setTokens` when deleting tokens in `ZunamiPool`
Severity	MAJOR
Status	• FIXED

Location

File	Location	Line
ZunamiPool.sol	contract `ZunamiPool` > function `_setTokens`	88

Description

In the `_setTokens` function of the `ZunamiPool` contract, there is a potential for an array out-of-bounds error when attempting to delete more tokens than were initially set.

The function operates by setting or removing tokens from the array across `POOL_ASSETS` iterations. Consider the following sequence:

- Initially, the `_setTokens` function sets the token count equal to `POOL_ASSETS`.
- Subsequently, a number of tokens equal to `POOL_ASSETS-3` is passed to `_setTokens`, resulting in the removal of three tokens from the `_tokens` array.
- If `_setTokens` is then called to set a token count of `POOL_ASSETS-2`, an array out-of-bounds error will occur in the `_tokens` array.

Recommendation

We recommend revising the token deletion logic in `_setTokens` to ensure it does not attempt to delete more elements than are present in the array.

Update

Fixed in commit `9ffa8e1b6128d1ade8459a4e492cee669ed241a1`.

2.3 WARNING

W-01	High slippage in `CrvUsdApsConvexCurveStratBase`, `FraxApsConvexCurveStratBase`
Severity	WARNING
Status	• ACKNOWLEDGED

Location

File	Location	Line
CrvUsdApsConvexCurveStratBase.sol	contract `CrvUsdApsConvexCurveStratBase` > function `inflate`	90
CrvUsdApsConvexCurveStratBase.sol	contract `CrvUsdApsConvexCurveStratBase` > function `deflate`	125
FraxApsConvexCurveStratBase.sol	contract `FraxApsConvexCurveStratBase` > function `inflate`	91
FraxApsConvexCurveStratBase.sol	contract `FraxApsConvexCurveStratBase` > function `deflate`	121

Description

The functions `inflate` and `deflate` in the contracts `CrvUsdApsConvexCurveStratBase` and `FraxApsConvexCurveStratBase` use the `minDeflateAmount` parameter, which limits slippage and is valued in USD, set in advance as a function parameter. This could lead to transaction reversion if the limit is too low or result in high slippage.

Recommendation

We recommend refactoring the slippage limitation mechanism of the APS strategies.

Update

Oxorio's response

We recommend implementing a percent-based slippage mechanism instead of a fixed value slippage in USD, to ensure that the slippage logic does not depend on fluctuations in the price of the asset used.

For example:
The `zunUSD` price is `0.9` USD, and it is necessary to exchange 10,000 `zunUSD` for `crvUsd` using the `deflate` method to equalize the exchange rate. Setting the slippage:

- In the current implementation: 90 USD (= 1%)
- In a percent-based implementation: 1% (= 90 USD)

Let's say the `zunUSD` price is `0.7` USD at the moment of transaction execution. So, the acceptable slippage is:

- In the current implementation: 90 USD (= 1.3%)
- In a percent-based implementation: 1% (= 70 USD)

As a result, the acceptable slippage in the current implementation is `1.3%`, which is more than the initial `1%`.

Zunami's response

In the `deflate` and `inflate` methods, two parameters are used: a percentage of the managed capital strategy in the external protocol and a minimum number of tokens. In the case of inflation, the second parameter determines the minimum number of stables that were obtained when withdrawing the tokens from the external pool and depositting the `Zunami Pool` to mint `zun` stables and return them back to the external protocol, thereby expanding the emission of zun stables. In the case of deflation, it determines the minimum number of stables that were obtained when converting `zun` stables before being deposited into the external protocol. Since the first parameter is initially specified in percentages, the minimum expected number of tokens after all conversions is specified in units, not percentages, to minimize the attack vector at the time of withdrawal and conversion. Therefore, specifying the second parameter as a percentage of slippage is considered a riskier scenario than specifying an explicit minimum number of tokens withdrawn.

W-02	Underflow in case of `depositedValue` is lower than `MINIMUM_LIQUIDITY` on the first deposit to the strategy in `ZunamiPool`
Severity	WARNING
Status	• FIXED

Location

File	Location	Line
ZunamiPool.sol	contract `ZunamiPool` > function `processSuccessfulDeposit`	199

Description

In the `processSuccessfulDeposit` function of the `ZunamiPool` contract, there is a risk of underflow if `depositedValue` is less than `MINIMUM_LIQUIDITY` during the initial deposit to the strategy. This situation arises because the value of `minted` would be lower than `locked`, leading to an underflow error:

```
if (totalSupply() == 0) {
    minted = depositedValue;
    locked = MINIMUM_LIQUIDITY;
    _mint(MINIMUM_LIQUIDITY_LOCKER, MINIMUM_LIQUIDITY);
} else {
    // ...
}
_mint(receiver, minted - locked);
```

Recommendation

We recommend implementing a validation check for the deposit size to ensure that the amount of tokens minted in the pool is not less than `MINIMUM_LIQUIDITY`.

Update

Fixed in commit `9ffa8e1b6128d1ade8459a4e492cee669ed241a1`.

2.4 INFO

I-01	Inflation attack in `ZunamiPool`
Severity	INFO
Status	• FIXED

Location

File	Location	Line
ZunamiPool.sol	contract `ZunamiPool` > function `processSuccessfulDeposit`	196

Description

In the function `processSuccessfulDeposit` of the `ZunamiPool` contract, the balance of the contract can be inflated by directly sending funds. This can result in an incorrect amount of shares issued.

```
minted =
  ((totalSupply() + 10 ** _decimalsOffset()) * depositedValue) /
  (totalDeposited + 1);
```

The attacker can front-run the first deposit and inflate the `totalDeposited` variable, resulting in zero shares being minted. While this attack results in loss for the attacker, the user still can loose their deposit.

Recommendation

We recommend increasing the `_decimalsOffset` value (for example `3`).

Update

Fixed in commit `79892fe12bec407d3d9706c19cd421d458263c0c`.

I-02	Redundant extension of the `AccessControl` contract to check two roles at once in `AccessControl2RolesValuation`
Severity	INFO
Status	• FIXED

Location

File	Location	Line
AccessControl2RolesValuation.sol	-	6

Description

In the `AccessControl2RolesValuation` contract, the `only2Roles` modifier is introduced to check the permissions of two roles simultaneously, specifically for the pair `DEFAULT_ADMIN_ROLE` and `EMERGENCY_ROLE`.

However, the `DEFAULT_ADMIN_ROLE` is the primary administrative role with authority to assign other roles, including the `EMERGENCY_ROLE`. Thus, an admin with the `DEFAULT_ADMIN_ROLE` can assign the `EMERGENCY_ROLE` to themselves.

Consequently, using `only2Roles([DEFAULT_ADMIN_ROLE, EMERGENCY_ROLE])` becomes redundant and can be replaced with the simpler modifier `onlyRole(EMERGENCY_ROLE)`.

Recommendation

We recommend revisiting the use of the `only2Roles` modifier and considering the use of `onlyRole` for code simplification.

Update

Fixed in commit `9ffa8e1b6128d1ade8459a4e492cee669ed241a1`.

I-03	Parameter validation
Severity	INFO
Status	• FIXED

Location

File	Location	Line
FraxApsConvexCurveStratBase.sol	contract `FraxApsConvexCurveStratBase` > `constructor`	46-47
CrvUsdApsConvexCurveStratBase.sol	contract `CrvUsdApsConvexCurveStratBase` > `constructor`	45-46
ConvexCurveStratBase.sol	contract `ConvexCurveStratBase` > `constructor`	22-23
CurveStratBase.sol	contract `CurveStratBase` > `constructor`	19-20
StakeDaoCurveStratBase.sol	contract `StakeDaoCurveStratBase` > `constructor`	15
RecapitalizationManager.sol	contract `RecapitalizationManager` > `constructor`	31
StakingRewardDistributor.sol	contract `StakingRewardDistributor` > function `withdrawEmergency`	430
StakingRewardDistributor.sol	contract `StakingRewardDistributor` > function `claim`	381
StakingRewardDistributor.sol	contract `StakingRewardDistributor` > function `updatePool`	300
StakingRewardDistributor.sol	contract `StakingRewardDistributor` > function `reallocatePool`	471
StakingRewardDistributor.sol	contract `StakingRewardDistributor` > function `addPool`	152
StakingRewardDistributor.sol	contract `StakingRewardDistributor` > function `addRewardToken`	131
ZunDistributor.sol	contract `ZunDistributor` > function `constructor`	78
GenericOracle.sol	contract `GenericOracle` > function `setCustomOracle`	43
ZunamiStratBase.sol	contract `ZunamiStratBase` > `constructor`	32
ZunamiStratBase.sol	contract `ZunamiStratBase` > `constructor`	33

Description

In the locations mentioned above, function parameters are not validated. This lack of validation can lead to unpredictable behavior or the occurrence of panic errors.

Recommendation

We recommend implementing validation for function parameters to ensure stable and predictable behavior.

Update

Fixed in commit `9ffa8e1b6128d1ade8459a4e492cee669ed241a1`.

I-04	Using constant in `CurveStratBase`
Severity	INFO
Status	• FIXED

Location

File	Location	Line
CurveStratBase.sol	contract `CurveStratBase` > function `checkDepositSuccessful`	36
ERC4626StratBase.sol	contract `ERC4626StratBase` > function `checkDepositSuccessful`	40

Description

In this locations, a hardcoded number `5` is used:

```
for (uint256 i = 0; i < 5; i++) {
```

Recommendation

We recommend using the `POOL_ASSETS` constant instead of a hardcoded number.

Update

Fixed in commit `9ffa8e1b6128d1ade8459a4e492cee669ed241a1`.

I-05	High decimals tokens support in `ZunamiStratBase`
Severity	INFO
Status	• ACKNOWLEDGED

Location

File	Location	Line
ZunamiStratBase.sol	contract `ZunamiStratBase`	23

Description

In the `ZunamiStratBase` contract, the `tokenDecimalsMultipliers` variable is used to support tokens with fewer than 18 decimals. However, there is no provision to support tokens with more than 18 decimals.

Recommendation

We recommend implementing support for tokens with high decimal counts.

I-06	Floating pragma
Severity	INFO
Status	• ACKNOWLEDGED

Description

All contracts across the codebase use the following pragma statement:

```
pragma solidity ^0.8.22;
```

Contracts should be deployed with the same compiler version and flags used during development and testing. An outdated pragma version might introduce bugs that affect the contract system negatively or recent compiler versions may have unknown security vulnerabilities.

Recommendation

We recommend locking the pragma to a specific version of the compiler.

3 CONCLUSION

The following table contains all the findings identified during the audit, grouped by statuses and severity levels:

Severity	FIXED	ACKNOWLEDGED	Total
CRITICAL	4	0	4
MAJOR	2	0	2
WARNING	1	1	2
INFO	4	2	6
Total	11	3	14

The found Critical and Major vulnerabilities have been fixed. However, further testing of the protocol and achieving full test coverage to ensure that the protocol meets the highest standards of security is recommended.

THANK YOU FOR CHOOSING

OXORIO