SMART CONTRACT AUDIT REPORT

for

Zunami

Prepared By: Xiaomi Huang

PeckShield

December 12, 2023

Document Properties

Client	Zunami
Title	Smart Contract Audit Report
Target	Zunami
Version	1.0
Author	Xuxian Jiang
Auditors	Jonathan Zhao, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	December 12, 2023	Xuxian Jiang	Final Release
1.0-rc	November 24, 2023	Xuxian Jiang	Release Candidate

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of `Zunami` protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Zunami

`Zunami` is a decentralized protocol that issues aggregated stablecoins, whose collateral is utilized in `omnipools` and differentiated among various profit-generating strategies. The protocol utilizes decentralized revenue aggregator to select the most profitable stablecoin pools and optimally balance funds between them, eliminating the need for constant market research and manual transfers. This allows for users to generate passive income with minimal effort. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Zunami

Item	Description
Issuer	Zunami
Website	https://www.zunami.io/
Type	Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	December 12, 2023

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/ZunamiProtocol/ZunamiProtocolV2.git (626c709)

And here is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/ZunamiProtocol/ZunamiProtocolV2.git (8b7774b)

1.2 About PeckShield

PeckShield Inc. [11] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [10]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

Table 1.3: The Full Audit Checklist

Category	Checklist Items
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	DeltaPrimeLabs DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

To evaluate the risk, we go through a checklist of items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [9], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logic	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `Zunami` protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	4	■■■■
Medium	2	■■
Low	2	■■
Informational	0	
Total	8	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 4 high-severity vulnerabilities, 2 medium-severity vulnerabilities, and 2 low-severity vulnerabilities.

Table 2.1: Key Zunami Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Confused Conversion Between Asset and Share in ERC4626StratBase	Business Logic	Resolved
PVE-002	High	Incorrect LiqToken Price Calculation in EthERC4626StratBase	Business Logic	Resolved
PVE-003	High	Incorrect Reward Distribution in StakingRewardDistributor	Business Logic	Resolved
PVE-004	High	Improper Staking Logic in StakingRewardDistributor	Business Logic	Resolved
PVE-005	Low	Improper Unstaking Logic in StakingRewardDistributor	Business Logic	Resolved
PVE-006	Medium	Revisited Slippage Control in SellingCurveRewardManagerFrxEth And Strategy Withdrawal	Time and State	Resolved
PVE-007	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-008	Low	Improved Fee Rate Activation in ZunamiPoolCompoundController	Coding Practices	Resolved

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Confused Conversion Between Asset and Share in ERC4626StratBase

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: High
- Target: ERC4626StratBase
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

Zunami has a number of built-in strategies to earn yields for staking users. In the process of examining the staking logic to compute the share amount, we notice its implementation needs to be revisited.

To elaborate, we show below the implementation of the related calcTokenAmount() routine. As the name indicates, this routine is used to compute the share amount based on the given token amounts. While the logic is rather straightforward, we notice it explicitly makes use of a wrong helper convertToAssets() to compute the share. The correct helper should be the convertToShare() routine.

```
    function calcTokenAmount(
        uint256[POOL_ASSETS] memory tokenAmounts,
        bool
    ) public view override returns (uint256 sharesAmount) {
        return vault.convertToAssets(convertVaultAssetAmounts(tokenAmounts));
    }
```

Listing 3.1: ERC4626StratBase::calcTokenAmount()

Recommendation Revise the above staking logic to compute the correct share amount.

Status The issue has been fixed by this commit: bdcf1a8.

3.2 Incorrect LiqToken Price Calculation in EthERC4626StratBase

- ID: PVE-002
- Severity: High
- Likelihood: High
- Impact: High
- Target: EthERC4626StratBase
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

As mentioned earlier, Zunami has a number of built-in strategies. In this section, we examine the common EthERC4626StratBase contract and notice an issue that does not correctly compute the liquidity token price.

Specifically, we show below the implementation of the related getLiquidityTokenPrice() routine. It has a rather straightforward logic in computing the liquidity token price. However, our analysis shows that it still needs to factor in the vault.pricePerShare(). In other words, we need to multiply the current result with the following factor: vault.pricePerShare()/ 10**18.

```
    function getLiquidityTokenPrice() internal view override returns (uint256) {
        return
            (oracle.getUSDPrice(address(vaultAsset)) * 1e18) /
            oracle.getUSDPrice(Constants.CHAINLINK_FEED_REGISTRY_ETH_ADDRESS);
    }
```

Listing 3.2: EthERC4626StratBase::getLiquidityTokenPrice()

The same issue is also present in the following ERC4626StratBase::getLiquidityBalance() routine.

```
    function getLiquidityTokenPrice() internal view virtual override returns (uint256) {
        return oracle.getUSDPrice(address(vaultAsset));
    }
```

Listing 3.3: ERC4626StratBase::getLiquidityBalance()

Recommendation Revise the above logic to properly compute the liquidity token price.

Status The issue has been fixed by this commit: bdcf1a8.

3.3 Incorrect Reward Distribution in StakingRewardDistributor

- ID: PVE-003
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `StakingRewardDistributor`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

To incentivize protocol participation, `Zunami` has the `StakingRewardDistributor` contract to manage and distribute protocol rewards. While examining the logic to keep track of user reward balances, we notice several issues in current implementation that may not properly calculate and distribute rewards.

To elaborate, we use the `distribute()` routine as an example and show below its implementation. Given the new reward amount, the new `rewardPerBlock` needs to be properly updated. However, it comes to our attention that it was incorrectly updated as `reward.rewardPerBlock = block.number` (line 174).

In addition, if new reward arrives before old reward is fully distributed, the remaining reward is computed based on the remaining blocks, i.e., `remainDistributionBlocks = block.number - reward.distributionBlock` (line 168), which should be revised as `reward.distributionBlock + BLOCKS_IN_2_WEEKS - block.number`. Moreover, the reward rate `rewardPerBlock` does not have the `1*18` multiplier (line 171). Similarly, there is a need to ensure the `lastRewardBlock` stays in the range of `distributionBlock` and `distributionBlock + BLOCKS_IN_2_WEEKS`.

```
152     function distribute(uint256 tid, uint256 amount) external onlyRole(DISTRIBUTOR_ROLE)
         {
153         RewardTokenInfo storage reward = rewardTokenInfo[tid];
154 
155         reward.token.safeTransferFrom(msg.sender, address(this), amount);
156 
157         if (reward.rewardPerBlock > 0) {
158             updateAllPools();
159         }
160 
161         if (
162             reward.distributionBlock == 0
163             block.number > reward.distributionBlock + BLOCKS_IN_2_WEEKS
164         ) {
165             reward.distributionBlock = block.number;
166             reward.rewardPerBlock = amount / BLOCKS_IN_2_WEEKS;
167         } else {
168             uint256 remainDistributionBlocks = block.number - reward.distributionBlock;
169 
```

```
            reward.rewardPerBlock =
                (amount + (remainDistributionBlocks * reward.rewardPerBlock) / 1e18) /
                BLOCKS_IN_2_WEEKS;

            reward.rewardPerBlock = block.number;
        }

        emit RewardPerBlockSet(tid, reward.rewardPerBlock);
    }
```

Listing 3.4: `StakingRewardDistributor::distribute()`

Recommendation Revise the above-mentioned routines to properly keep track of the reward distribution and user reward balances.

Status The issue has been fixed by this commit: `bdcf1a8`.

3.4 Improper Staking Logic in StakingRewardDistributor

- ID: PVE-004
- Severity: High
- Likelihood: High
- Impact: High
- Target: `StakingRewardDistributor`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

In Section 3.3, we examine the `StakingRewardDistributor` contract and report related issues in reward calculation and distribution. In this section, we examine the same contract and report an issue in its staking logic.

To elaborate, we show below the related `deposit()` routine that is designed to stake user assets. In this routine, user rewards from earlier stakes will need to be computed. However, current reward is calculated based on the total stake amount that includes new stake amount (line 241). To fix, there is a need to follow the call order as follows: `updatePool()--> accrueReward()-> transferFrom() -> calcReward()`.

```
223     function deposit(uint256 _pid, uint256 _amount, bool _infiniteLock) external
            nonReentrant {
224         updatePool(_pid);
225 
226         UserPoolInfo storage userPool = userPoolInfo[_pid][msg.sender];
227 
228         if (_amount > 0) {
229             poolInfo[_pid].token.safeTransferFrom(address(msg.sender), address(this),
                _amount);
```

```
            userPool.amount = userPool.amount + _amount;

            IERC20Supplied stakingToken = poolInfo[_pid].stakingToken;
            if (address(stakingToken) != address(0)) {
                stakingToken.mint(msg.sender, _amount);
            }
        }

        uint256 length = rewardTokenInfo.length;
        for (uint256 tid = 0; tid < length; ++tid) {
            accrueReward(tid, _pid);
            userPool.accruedRewards[tid] = calcReward(
                tid,
                poolInfo[_pid],
                userPoolInfo[_pid][msg.sender]
            );
        }

        userPool.depositedBlock = block.number;
        userPool.infiniteLock = userPool.infiniteLock  _infiniteLock; // stay true if
            was true
        emit Deposited(msg.sender, _pid, _amount);
    }
```

Listing 3.5: `StakingRewardDistributor::deposit()`

Recommendation Correct the above routine to properly compute user reward amount.

Status The issue has been fixed by this commit: `bdcf1a8`.

3.5 Improper Unstaking Logic in StakingRewardDistributor

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `StakingRewardDistributor`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

In last Section 3.4, we examine the `StakingRewardDistributor` contract and report issues in current staking logic. Next, we examine the unstaking logic in the same contract.

In the following, we show the implementation of the `withdraw()` routine. When the user withdraws their funds, there is a need to compute the reward and transfer back to user. However, the reward is computed based on the remaining amount after the withdrawal. The correct reward should be computed based on the staked amount before the withdrawal.

```
    function withdraw(uint256 _pid, uint256 _amount) external nonReentrant {
        require(userPoolInfo[_pid][msg.sender].amount >= _amount, 'withdraw: not enough
            amount');
        updatePool(_pid);
        UserPoolInfo storage userPool = userPoolInfo[_pid][msg.sender];
        if (_amount > 0) {
            userPool.amount -= _amount;
            uint256 transferAmount = _amount;
            if (
                userPool.infiniteLock  userPool.depositedBlock > block.number -
                    BLOCKS_IN_4_MONTHS
            ) {
                transferAmount =
                    (_amount * (PERCENT_DENOMINATOR - EXIT_PERCENT)) /
                    PERCENT_DENOMINATOR;
                poolInfo[_pid].token.safeTransfer(earlyExitReceiver, _amount -
                    transferAmount);
            }
            poolInfo[_pid].token.safeTransfer(address(msg.sender), transferAmount);

            IERC20Supplied stakingToken = poolInfo[_pid].stakingToken;
            if (address(stakingToken) != address(0)) {
                stakingToken.burn(_amount);
            }
        }

        uint256 length = rewardTokenInfo.length;
        for (uint256 tid = 0; tid < length; ++tid) {
            accrueReward(tid, _pid);
            userPool.accruedRewards[tid] = calcReward(
                tid,
                poolInfo[_pid],
                userPoolInfo[_pid][msg.sender]
            );
        }
        emit Withdrawn(msg.sender, _pid, _amount);
    }
```

Listing 3.6: `StakingRewardDistributor::withdraw()`

Moreover, in the related `withdrawEmergency()` routine, there is a need to add the `nonReentrant` modifier to block possible reentrancy risk. And to be consistent with current normal withdrawal logic, we need to burn respective `stakingToken` as well.

Recommendation Revise the above unstaking logic to compute the proper amount and burn `stakingToken`, if any.

Status The issue has been fixed by this commit: `bdcf1a8`.

3.6 Revisited Slippage Control in SellingCurveRewardManagerFrxEth And Strategy Withdrawal

- ID: PVE-006
- Severity: Medium
- Likelihood: Low
- Impact: Medium
- Target: `SellingCurveRewardManagerFrxEth`
- Category: Time and State [8]
- CWE subcategory: CWE-682 [3]

Description

As mentioned earlier, the `Zunami` protocol utilizes decentralized revenue aggregators to select the most profitable strategies. The profit will be sent to reward manager for dissemination. While examining the specific reward manager `SellingCurveRewardManagerFrxEth`, we notice the current slippage control can be improved.

```
    function checkSlippage(address reward, uint256 amount, uint256 wethAmount) internal view {
        address rewardEthOracle = rewardEthChainlinkOracles[reward];
        uint256 wethAmountByOracle;
        if (rewardEthOracle != address(0)) {
            AggregatorV2V3Interface oracle = AggregatorV2V3Interface(rewardEthOracle);
            (, int256 answer, , uint256 updatedAt, ) = oracle.latestRoundData();
            require(block.timestamp - updatedAt <= STALE_DELAY, 'Oracle stale');

            wethAmountByOracle = (uint256(answer) * amount);
        } else {
            AggregatorV2V3Interface rewardOracle = AggregatorV2V3Interface(
                rewardUsdChainlinkOracles[reward]
            );
            (, int256 rewardAnswer, , uint256 updatedAt, ) = rewardOracle.latestRoundData();

            require(block.timestamp - updatedAt <= STALE_DELAY, 'Oracle usd stale');

            AggregatorV2V3Interface ethOracle = AggregatorV2V3Interface(ethUsdChainlinkOracle);
            (, int256 ethAnswer, , uint256 ethUpdatedAt, ) = ethOracle.latestRoundData();
            require(block.timestamp - ethUpdatedAt <= STALE_DELAY, 'Oracle eth stale');

            wethAmountByOracle =
                (uint256(rewardAnswer) * amount * TOKEN_PRICE_MULTIPLIER) /
                uint256(ethAnswer);
        }
```

```
139         uint256 wethAmountByOracleWithSlippage = (wethAmountByOracle *
140             (SLIPPAGE_DENOMINATOR - defaultSlippage)) / SLIPPAGE_DENOMINATOR;
141         require(
142             wethAmount >= wethAmountByOracleWithSlippage / TOKEN_PRICE_MULTIPLIER,
143             'Wrong slippage'
144         );
145     }
```

Listing 3.7: `SellingCurveRewardManagerFrxEth::checkSlippage()`

Specifically, if we examine the `SellingCurveRewardManagerFrxEth` contract, it has the `checkSlippage()` routine to check whether the slippage control is acceptable or not. We notice the reward amount will be converted to `WETH` for comparison, i.e., `wethAmountByOracle = (uint256(answer)* amount)` (line 121). However, current result needs to be normalized to have `18` decimals. In other words, we still need to multiply with `TOKEN_PRICE_MULTIPLIER/ oracle.decimals()`.

Moreover, when examining the withdrawal logic in current strategies, there is a support to withdraw all funds in the given strategy. We notice the full withdrawal logic does not have any slippage control. Further, normal withdrawal logic specifies the minimum amounts for the received tokens. However, it is enforced as the minimal LP tokens that is computed by depositing these minimum tokens.

Recommendation Improve the above-mentioned slippage control to avoid unwanted MEV risks.

Status The issue has been fixed by this commit: `eda29b2`.

3.7 Trust Issue of Admin Keys

- ID: PVE-007
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the `Zunami` protocol, there is a special administrative account (with the `DEFAULT_ADMIN_ROLE`). This account plays a critical role in governing and regulating the protocol-wide operations (e.g., parameter configuration and strategy management). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and its related privileged accesses in current contracts.

```
    function setTokens(
        address[] memory tokens_,
        uint256[] memory _tokenDecimalMultipliers
    ) external onlyRole(DEFAULT_ADMIN_ROLE) {
        _setTokens(tokens_, _tokenDecimalMultipliers);
    }

    function replaceToken(
        uint256 _tokenIndex,
        address _token,
        uint256 _tokenDecimalMultiplier
    ) external onlyRole(DEFAULT_ADMIN_ROLE) {
        address oldToken = address(_tokens[_tokenIndex]);
        _tokens[_tokenIndex] = IERC20(_token);
        _decimalsMultipliers[_tokenIndex] = _tokenDecimalMultiplier;
        emit UpdatedToken(_tokenIndex, _token, _tokenDecimalMultiplier, oldToken);
    }

    function pause() external onlyRole(DEFAULT_ADMIN_ROLE) {
        _pause();
    }

    function unpause() external onlyRole(DEFAULT_ADMIN_ROLE) {
        _unpause();
    }
```

Listing 3.8: Example Privileged Operations in `ZunamiPool`

```
    function withdrawStuckToken(IERC20 _token) external onlyRole(DEFAULT_ADMIN_ROLE) {
        uint256 tokenBalance = _token.balanceOf(address(this));
        if (tokenBalance > 0) {
            _token.safeTransfer(_msgSender(), tokenBalance);
        }
    }

    function setEarlyExitReceiver(address _receiver) external onlyRole(
        DEFAULT_ADMIN_ROLE) {
        earlyExitReceiver = _receiver;
        emit EarlyExitReceiverChanged(_receiver);
    }

    function addRewardToken(IERC20 _token) external onlyRole(DEFAULT_ADMIN_ROLE) {
        uint256 tid = rewardTokenInfo.length;
        rewardTokenInfo.push(
            RewardTokenInfo({ token: _token, rewardPerBlock: 0, distributionBlock: 0 })
        );
        rewardTokenTidByAddress[address(_token)] = tid;

        emit RewardTokenAdded(address(_token), tid);
    }
```

Listing 3.9: Example Privileged Operations in `StakingRewardDistributor`

We understand the need of the privileged functions for contract maintenance, but it is worrisome if the privileged account is a plain EOA account. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO.

Moreover, it should be noted that current contracts are to be deployed behind a proxy with the typical `UUPSUpgradeable` implementation. And naturally, there is a need to properly manage the admin privileges as they are capable of upgrading the entire protocol implementation.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been mitigatd as the team clarifies the use of a `DAO` multisig.

3.8 Improved Fee Rate Activation in ZunamiPoolCompoundControllers

- ID: PVE-008
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `ZunamiPoolCompoundControllers`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `Zunami` protocol is no exception. Specifically, if we examine the `ZunamiPoolCompoundControllers` contract, it has defined a number of protocol-wide risk parameters, such as `feeTokenId` and `managementFeePercent`. In the following, we show the corresponding routines that allow for their changes.

```
    function setManagementFeePercent(
        uint256 newManagementFeePercent
    ) external onlyRole(DEFAULT_ADMIN_ROLE) {
        if (newManagementFeePercent > MAX_FEE) revert WrongFee();
        emit ManagementFeePercentSet(managementFeePercent, newManagementFeePercent);
        managementFeePercent = newManagementFeePercent;
    }

    function setFeeTokenId(uint256 _tokenId) external onlyRole(DEFAULT_ADMIN_ROLE) {
        if (collectedManagementFee != 0) revert FeeMustBeWithdrawn();

        feeTokenId = _tokenId;
```

```
        emit SetFeeTokenId(_tokenId);
    }
```

Listing 3.10: ZunamiPoolCompoundControllers::setManagementFeePercent()/setFeeTokenId()

We notice the management fee is collected when selling the reward tokens back to the fee token. When the management fee is updated, there is a need to ensure the reward is timely collected before new management fee is applied. By doing so, we can ensure new management fee is only applicable on yields from now on.

Recommendation Timely collect rewards before new management fee is applied.

Status The issue has been fixed by this commit: `bdcf1a8`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Zunami` protocol, which is a decentralized protocol that issues aggregated stablecoins, whose collateral is utilized in `omnipools` and differentiated among various profit-generating strategies. The protocol utilizes decentralized revenue aggregator to select the most profitable stablecoin pools and optimally balance funds between them, eliminating the need for constant market research and manual transfers. This allows for users to generate passive income with minimal effort. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and fixed

Meanwhile, we need to emphasize that `Solidity-based` smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[3] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

[4] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[5] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[6] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[7] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[8] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

[9] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[10] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[11] PeckShield. PeckShield Inc. https://www.peckshield.com.