SMART CONTRACT AUDIT REPORT

for

Zerobase Staking

Prepared By: Xiaomi Huang

PeckShield

February 24, 2025

Document Properties

Client	Zerobase
Title	Smart Contract Audit Report
Target	Zerobase Staking
Version	1.0
Author	Xuxian Jiang
Auditors	Daisy Cao, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	February 24, 2025	Xuxian Jiang	Final Release
1.0-rc1	February 20, 2025	Xuxian Jiang	Release Candidate #1

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 the `Zerobase Staking` 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 Zerobase Staking

`Zerobase Staking` is an incentive and constraint system designed to ensure the security and reliability of prover nodes during `ZKP` generation. Prover nodes must stake stablecoins to join the proof network. These staked stablecoins are used for trading arbitrage via `CEFFU`, generating additional returns. The basic information of Zerobase Staking is as follows:

Table 1.1: Basic Information of Zerobase Staking

Item	Description
Issuer	Zerobase
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	February 24, 2025

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note the given repository has a number of directories and files and this audit only covers the smart contracts located under the `V2/src` subdirectory.

- https://github.com/ZeroBase-Pro/ZKFi.git (564867b)

1.2 About PeckShield

PeckShield Inc. [7] 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			
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low
	High	*Medium*	*Low*
		Likelihood	

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [6]:

- 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.

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

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
	Revert 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

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) [5], 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 `Zerobase Staking` 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	0	
Medium	3	🟦🟦🟦
Low	0	
Informational	0	
Total	3	

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 3 medium-severity vulnerabilities.

Table 2.1: Key Zerobase Staking Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Revisited Mint Amount Calculation Logic in Vault	Business Logic	Confirmed
PVE-002	Medium	Improper Claim-Cancelling Logic in Vault	Business Logic	Resolved
PVE-003	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

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 Revisited Mint Amount Calculation Logic in Vault

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium

- Target: Vault
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

The Zerobase Staking protocol has a core Vault contract that serves as the main entry for users to stake and claim rewards. In the process of examining the staking logic, we notice the staking share calculation can be improved.

In the following, we show the implementation of the related stake_66380860() routine. It has a rather straightforward logic in receiving the user stake and calculating respective share amount to mint. The share amount should be computed in proportion to the stake amount with the precision preference in favor of the staking protocol. With that, it comes to our attention that the exchange rate logic needs to be enhanced by taking into account the precision preference, i.e., one extra argument to indicate the ceiling/floor arithmetic calculation. For this specific staking routine, the exchange rate should be computed by taking the ceiling preference. Note the same exchange rate calculation in other routines should be adjusted accordingly.

```
    function stake_66380860(address _token, uint256 _stakedAmount) external
        onlySupportedToken(_token) whenNotPaused {
        AssetsInfo storage assetsInfo = userAssetsInfo[msg.sender][_token];
        uint256 currentStakedAmount = assetsInfo.stakedAmount;

        require(Utils.Add(currentStakedAmount, _stakedAmount) >= minStakeAmount[_token]);
        require(Utils.Add(currentStakedAmount, _stakedAmount) <= maxStakeAmount[_token]);

        IERC20(_token).safeTransferFrom(msg.sender, address(this), _stakedAmount);
        _updateRewardState(msg.sender, _token);
        uint256 exchangeRate = _getExchangeRate(_token);

        totalStakeAmountByToken[_token] += _stakedAmount;
        uint256 mintAmount = _stakedAmount * 1e18 / exchangeRate;
        supportedTokenToZkToken[_token].mint(msg.sender, mintAmount);

        // update status
        assetsInfo.stakeHistory.push(
            StakeItem({
                stakeTimestamp: block.timestamp,
                amount: _stakedAmount,
                token: _token,
                user: msg.sender
            })
        );
        unchecked {
            assetsInfo.stakedAmount += _stakedAmount;
            tvl[_token] += _stakedAmount;
        }

        emit Stake(msg.sender, _token, _stakedAmount);
    }
```

Listing 3.1: Vault::stake_66380860()

Recommendation Revise the above routine to properly adjust the precision preference for the exchange rate calculation. The following routines can also be similarly improved, including `requestClaim_8135334()`, `cancelClaim()`, and `flashWithdrawWithPenalty()`.

Status This issue has been confirmed and the team is aware of the rounding errors that are extremely minimal and can be tolerated.

3.2 Improper Claim-Cancelling Logic in Vault

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Vault`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

As mentioned in Section 3.1, staking users mainly interact with the `Vault` contract to stake and claim rewards. Note an existing claim to unstake may be cancelled. Our analysis on the claim-cancellation logic shows an issue that needs to be resolved.

To elaborate, we show below the implementation of the related `cancelClaim()` routine. As the name indicates, this routine is designed to cancel an existing claim. The cancellation logic needs to timely accumulate user rewards. With that, there is a need to call `_updateRewardState()` (line 293) upon the entry of this routine. Current logic calls it after updating the user's `lastRewardUpdateTime` (line 291), which unfortunately reduces the user rewards.

```
265     function cancelClaim(uint256 _queueId, address _token) external whenNotPaused OnlyCancelEnable{
266         ClaimItem memory claimItem = claimQueue[_queueId];
267         delete claimQueue[_queueId];

269         address token = claimItem.token;
270         AssetsInfo storage assetsInfo = userAssetsInfo[msg.sender][token];
271         uint256[] memory pendingClaimQueueIDs = userAssetsInfo[msg.sender][token].pendingClaimQueueIDs;

273         require(Utils.MustGreaterThanZero(claimItem.totalAmount));
274         require(claimItem.user == msg.sender);
275         require(!claimItem.isDone, "claimed");
276         require(token == _token, "wrong token");

278         for(uint256 i = 0; i < pendingClaimQueueIDs.length; i++) {
279             if(pendingClaimQueueIDs[i] == _queueId) {
280                 assetsInfo.pendingClaimQueueIDs[i] = pendingClaimQueueIDs[pendingClaimQueueIDs.length-1];
281                 assetsInfo.pendingClaimQueueIDs.pop();
282                 break;
283             }
284         }

286         uint256 principal = claimItem.principalAmount;
287         uint256 reward = claimItem.rewardAmount;

289         assetsInfo.stakedAmount += principal;
290         assetsInfo.accumulatedReward += reward;
291         assetsInfo.lastRewardUpdateTime = block.timestamp;

293         _updateRewardState(msg.sender, _token);
294         uint256 exchangeRate = _getExchangeRate(_token);
295         uint256 amountToMint = (principal + reward) * 1e18 / exchangeRate;

297         totalStakeAmountByToken[_token] += principal;
298         totalRewardsAmountByToken[_token] += reward;

300         supportedTokenToZkToken[_token].mint(msg.sender, amountToMint);

302         emit CancelClaim(msg.sender, _token, principal + reward, _queueId);
303     }
```

Listing 3.2: `Vault::cancelClaim()`

Recommendation Revise the above routine for accurately and timely accumulate the user rewards.

Status This issue has been resolved as the team confirms it is part of the design.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Low
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

Description

In the `Zerobase Staking` protocol, there is a privileged account (with the `DEFAULT_ADMIN_ROLE` role) that plays a critical role in governing and regulating the protocol-wide operations (e.g., assign roles, pause/unpause staking, and recover funds). 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 the related privileged accesses in current contracts.

```
    function emergencyWithdraw(address _token, address _receiver) external onlyRole(
        DEFAULT_ADMIN_ROLE) {
        // `_token` could be not supported, so that we could sweep the tokens which are
            sent to this contract accidentally
        Utils.CheckIsZeroAddress(_token);
        Utils.CheckIsZeroAddress(_receiver);

        IERC20(_token).safeTransfer(_receiver, IERC20(_token).balanceOf(address(this)));
        emit EmergencyWithdrawal(_token, _receiver);
    }

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

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

Listing 3.3: Example Privileged Function(s) in `Vault`

Note that if the privileged `owner` account is a plain EOA account, this may be worrisome and pose counter-party risk to the exchange users. 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. In the meantime, a timelock-based mechanism can also be considered as mitigation.

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 mitigated as a multisig account is used to hold the `owner` account.

4 Conclusion

In this audit, we have analyzed the design and implementation of the Zerobase Staking protocol, which is an incentive and constraint system designed to ensure the security and reliability of prover nodes during ZKP generation. Prover nodes must stake stablecoins to join the proof network. These staked stablecoins are used for trading arbitrage via CEFFU, generating additional returns. 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

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

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

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

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

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

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