SMART CONTRACT AUDIT REPORT

for

ZeroBase Finance

Prepared By: Xiaomi Huang

PeckShield
December 20, 2024

Document Properties

Client	ZeroBase Finance
Title	Smart Contract Audit Report
Target	ZeroBase Finance
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	December 20, 2024	Xuxian Jiang	Final Release
1.0-rc	December 20, 2024	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Xiaomi Huang
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the `ZeroBase` 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

`ZeroBase` is a real-time `zero-knowledge (ZK)prover network` designed for rapid proof generation, decentralization, and regulatory compliance. It generates `ZK` proofs within hundreds of milliseconds, enabling large-scale commercial applications. The audited vault is a secure staking and rewards management system built on the `Ethereum Virtual Machine (EVM)` blockchain. It allows users to stake supported tokens and earn rewards. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of ZeroBase Finance

Item	Description
Name	ZeroBase Finance
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	December 20, 2024

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

- https://github.com/github.com:ZeroBase-Pro/zerobase-vault.git (acc403f)

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

- https://github.com/github.com:ZeroBase-Pro/zerobase-vault.git (80fe22a)

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 \ 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 [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

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

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) [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.

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.

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.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `ZeroBase` 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	1	■
Low	3	■ ■ ■
Informational	0	
Total	4	

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 1 medium-severity vulnerability and 3 low-severity vulnerabilities.

Table 2.1: Key ZeroBase Finance Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Claim-Requesting Logic in Vault	Business Logic	Resolved
PVE-002	Low	Revisited Denominator in Vault::getCurrentRewardRate()	Business Logic	Resolved
PVE-003	Low	Improved Claimable Reward Amount Calculation	Coding Practices	Resolved
PVE-004	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 Improved Claim-Requesting Logic in Vault

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Vault`
- Category: Business Logic [4]
- CWE subcategory: CWE-837 [2]

Description

The audited `Vault` is a secure staking and rewards management system. While reviewing the logic to claim the rewards, we notice current implementation can be improved.

To elaborate, we show below the code snippet from the related `requestClaim_8135334()` function. It implements a rather straightforward logic in validating and queuing the user request for withdrawal. However, we notice current implementation makes repeated storage reads of `assetsInfo.accumulatedReward`, which can be optimized with cache. Also, the `lastRewardUpdateTime` state only needs to be updated once.

```
    function requestClaim_8135334(
        address _token,
        uint256 _amount
    ) external onlySupportedToken(_token) whenNotPaused returns(uint256 _returnID) {
        _updateRewardState(msg.sender, _token);

        AssetsInfo storage assetsInfo = userAssetsInfo[msg.sender][_token];
        uint256 currentStakedAmount = assetsInfo.stakedAmount;
        uint256 currentAccumulatedRewardAmount = assetsInfo.accumulatedReward;

        require(
            Utils.MustGreaterThanZero(_amount) &&
            (_amount <= Utils.Add(currentStakedAmount, currentAccumulatedRewardAmount)
                _amount == type(uint256).max),
            "Invalid amount"
        );

```

```
        ClaimItem storage queueItem = claimQueue[lastClaimQueueID];

        // Withdraw from reward first; if insufficient, continue withdrawing from principal
        uint256 totalAmount = _amount;
        if(_amount == type(uint256).max){
            totalAmount = Utils.Add(assetsInfo.accumulatedReward, assetsInfo.stakedAmount);

            queueItem.rewardAmount = assetsInfo.accumulatedReward;
            assetsInfo.accumulatedReward = 0;

            queueItem.principalAmount = assetsInfo.stakedAmount;
            assetsInfo.stakedAmount = 0;
        }else if(currentAccumulatedRewardAmount >= _amount) {
            assetsInfo.accumulatedReward -= _amount;

            queueItem.rewardAmount = _amount;
        } else {
            queueItem.rewardAmount = assetsInfo.accumulatedReward;
            assetsInfo.accumulatedReward = 0;

            uint256 difference = _amount - currentAccumulatedRewardAmount;
            assetsInfo.stakedAmount -= difference;
            queueItem.principalAmount = difference;
        }
        ...
    }
```

Listing 3.1: `Vault::requestClaim_8135334()`

Recommendation Revise the above logic to avoid repeated storaged reads (`assetsInfo.accumulatedReward`) and writes (`lastRewardUpdateTime`).

Status This issue has been fixed in the following commit: `80fe22a`.

3.2 Revisited Denominator in Vault::getCurrentRewardRate()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Vault`
- Category: Business Logic [4]
- CWE subcategory: CWE-837 [2]

Description

The `ZeroBase` protocol has a core `Vault` contract to allow users to stake and claim rewards. In the process of examining the related getter function to query current reward rate, we notice it returns an incorrect denominator.

In particular, we show below the implementation of this related getter routine, i.e., `getCurrentRewardRate()`. In essence, it is used to query the reward rate as a percentage. Note the returned reward rate has the denominator of `10000`, not current `100` (line 566).

```
561     // Measure the reward rate as a percentage, and return the numerator and denominator
562     function getCurrentRewardRate(address _token) external view returns (uint256,
            uint256) {
563         RewardRateState[] memory rewardRateStateArray = rewardRateState[_token];
564         RewardRateState memory currentRewardRateState = rewardRateStateArray[
                rewardRateStateArray.length - 1];
565
566         return (currentRewardRateState.rewardRate, 100);
567     }
```

Listing 3.2: `Vault::getCurrentRewardRate()`

Recommendation Revise the above getter to ensure the right denominator is returned.

Status This issue has been fixed in the following commit: `91d9ff3`.

3.3 Improved Claimable Reward Amount Calculation

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Vault`
- Category: Business Logic [4]
- CWE subcategory: CWE-837 [2]

Description

As mentioned earlier, `ZeroBase` has a core `Vault` contract to allow users to stake and claim rewards. In the process of examining the reward amount calculation, we notice a corner case that can be better addressed.

In particular, we show below the code snippet of this related routine, i.e., `_getClaimableRewards()`. As the name indicates, it is used to compute the claimable rewards. It comes to our attention that the use of `beginIndex` is to find the corresponding index in the reward rate array based on the reward rate at the time of the last stake. In the corner of having `beginIndex = 0`, the iteration of `beginIndex-1` reward rate will result in an arithmetic underflow (line 442). With that, we suggest to add the following statement, i.e., `if (i==0)continue;`. Fortunately, this corner case may not be triggered in current state-modifying callers.

```
426         uint256 beginIndex = 0;
427         for (uint256 i = 0; i < rewardRateLength; i++) {
428             if (lastRewardUpdate < rewardRateArray[i].updatedTime) {
429                 beginIndex = i;
430                 break;
431             }
432         }
433 
434         // b. iterate to the latest-1 reward rate
435         uint256 tempLastRewardUpdateTime = lastRewardUpdate;
436         for (uint256 i = beginIndex; i < rewardRateLength; i++) {
437             if(i == 0) continue;
438 
439             uint256 tempElapsedTime = rewardRateArray[i].updatedTime -
                    tempLastRewardUpdateTime;
440             uint256 tempReward = calculateReward(
441                 currentStakedAmount,
442                 rewardRateArray[i - 1].rewardRate,
443                 tempElapsedTime
444             );
445             tempLastRewardUpdateTime = rewardRateArray[i].updatedTime;
446             unchecked{
447                 assetsInfo.accumulatedReward += tempReward;
448             }
```

```
        }
```

Listing 3.3: `Vault::_getClaimableRewards()`

Recommendation Revise the above routine to properly compute the claimable reward amount.

Status This issue has been fixed in the following commit: `80fe22a`.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Vault`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

Description

In the `ZeroBase` protocol, there is a privileged account (with the `DEFAULT_ADMIN_ROLE` role) that plays a critical role in governing and regulating the system-wide operations (e.g., configure vaults parameter, assign roles, and execute privileged operations). 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 their related privileged accesses in current contracts.

```
    function transferToCeffu(
        address _token,
        uint256 _amount
    ) external onlySupportedToken(_token) onlyRole(BOT_ROLE) {
        require(Utils.MustGreaterThanZero(_amount), "Amount must be greater than zero");
        require(_amount <= IERC20(_token).balanceOf(address(this)), "Not enough balance"
            );

        IERC20(_token).safeTransfer(ceffu, _amount);

        emit CeffuReceive(_token, ceffu, _amount);
    }
    ...
    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 3.4: Example Privileged Functions in the `Vault` Contract

If the privileged account is managed by 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 the team has confirmed that these privileged functions should be called by a trusted multi-sig account, not a plain EOA account.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the `ZeroBase` protocol, which is a real-time `zero-knowledge (ZK)prover network` designed for rapid proof generation, decentralization, and regulatory compliance. It generates `ZK` proofs within hundreds of milliseconds, enabling large-scale commercial applications. The audited vault is a secure staking and rewards management system built on the `Ethereum Virtual Machine (EVM)` blockchain. It allows users to stake supported tokens and earn rewards. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and fixed.

Moreover, 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-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.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.