SMART CONTRACT AUDIT REPORT

for

Pod Finance

Prepared By: Xiaomi Huang

PeckShield

April 22, 2024

Document Properties

Client	Pod Finance
Title	Smart Contract Audit Report
Target	Pod Finance
Version	1.0
Author	Xuxian Jiang
Auditors	Jason Shen, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	April 22, 2024	Xuxian Jiang	Final Release
1.0-rc1	April 18, 2024	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 **4**

1.1 About Pod Finance 4

1.2 About PeckShield 5

1.3 Methodology 5

1.4 Disclaimer 7

2 Findings **9**

2.1 Summary 9

2.2 Key Findings 10

3 Detailed Results **11**

3.1 Incorrect accRewardsPerShare Calculation Logic in _updatePod() 11

3.2 Improved Validation of Function Arguments in Pod 13

3.3 Improved Boundary Handling in Pod 14

4 Conclusion **16**

References **17**

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the Pod 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 Pod Finance

Pod is a fully permissionless protocol that enables users to create tokenized vaults by following the ERC4626{ standard with any ERC20 -like token as underlying. When underlying is deposited, a proportional share to vaults' ratio is minted and can be staked to earn tokens from fixed reward pools according to their time-weighted contribution to the vault. Reward pools may be created by the Pod owner - if the Pod is ownable - or by any other third-party if is created as permissionless. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The Pod Finance Protocol

Item	Description
Name	Pod Finance
Website	https://pod.finance
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	April 22, 2024

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

- https://github.com/stredbasjio/AUDIT_pod-contracts.git (ab93038)

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

- https://github.com/stredbasjio/AUDIT_pod-contracts.git (70b49e9)

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

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on 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 list of check 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

Table 1.3: The Full List of Check Items

Category	Check Item
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

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) [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.

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 Logics	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 `Pod` 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 logics, 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	2	■ ■
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 that 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 2 low-severity vulnerabilities.

Table 2.1: Key Pod Finance Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Incorrect accRewardsPerShare Calculation Logic in _updatePod()	Business Logic	Resolved
PVE-002	Medium	Improved Validation of Function Arguments in Pod	Coding Practices	Resolved
PVE-003	Low	Improved Boundary Handling in Pod	Coding Practices	Resolved

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that 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 need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Incorrect accRewardsPerShare Calculation Logic in _updatePod()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Pod`
- Category: Business Logic [4]
- CWE subcategory: CWE-770 [2]

Description

As mentioned earlier, `Pod` enables users to create tokenized vaults by following the `ERC4626` standard with any `ERC20` -like token as underlying. The tokenized vault has the built-in reward with the accumulated index (internally saved as `accRewardsPerShare`). While examining its calculation when a new reward token is added, we notice current calculation should be improved.

In the following, we show the implementation of the related routine `_updatePod()`. As the name indicates, this routine is used to update `Pod` by refreshing the above `accRewardsPerShare` index. We notice an internal variable `_inBetweenTime` is used to represent the time elapse between `_lastValidTime` and `currentBlockTimestamp` (line 379). However, it does not consider the situation where the reward may have its `endTime` smaller than `currentBlockTimestamp`. If that is the case, the `_inBetweenTime` variable is incorrectly calculated, leading to unfair reward distribution.

```
359    function _updatePod() internal {
360        uint256 currentBlockTimestamp = _currentBlockTimestamp();

362        if (currentBlockTimestamp <= _lastRewardTimestamp) return;

364        if (_totalSharesLocked == 0) {
365            _lastRewardTimestamp = currentBlockTimestamp;
366            emit UpdatePod(currentBlockTimestamp);
367            return;
368        }
```

```
        for (uint256 i = 0; i < _rewardsCount; i++) {
            RewardToken storage _rewardToken = rewards[_reverseRewards[i]];
            if (_rewardToken.remainingAmount == 0  currentBlockTimestamp < _rewardToken.
                settings.startTime) continue;
            uint256 _lastValidTime = _lastRewardTimestamp;

            if(_lastValidTime < _rewardToken.settings.startTime) {
                _lastValidTime = _rewardToken.settings.startTime;
            }

            (bool successTime, uint256 _inBetweenTime) = currentBlockTimestamp.trySub(
                _lastValidTime);
            if (!successTime) revert PodMathError();

            (bool success, uint256 _rewardAmount) =
                (_rewardsPerSecond(_rewardToken.settings.endTime, _lastValidTime,
                    _rewardToken.remainingAmount)).tryMul(_inBetweenTime);

            if (!success) revert PodMathError();

            if (_rewardAmount > _rewardToken.remainingAmount) _rewardAmount =
                _rewardToken.remainingAmount;

            _rewardToken.remainingAmount = _rewardToken.remainingAmount - _rewardAmount;
            _rewardToken.accRewardsPerShare =
                _rewardToken.accRewardsPerShare + ((_rewardAmount.mulDiv(10 ** decimals
                    (), _totalSharesLocked)));
        }

        _lastRewardTimestamp = currentBlockTimestamp;
        emit UpdatePod(currentBlockTimestamp);
    }
```

Listing 3.1: `Pod::_updatePod()`

Recommendation Improve the above-mentioned routine to properly update in `accRewardsPerShare`
.

Status This issue has been fixed by the following commit: `d46cdb4`.

3.2 Improved Validation of Function Arguments in Pod

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Pod
- Category: Coding Practices [3]
- CWE subcategory: CWE-1126 [1]

Description

The `Pod` vault has a key `modifyRewardToken()` function to modify the given token's reward configuration or add additional token amount for distribution. While reviewing its logic, we notice it can be improved by more thoroughly validating the given input arguments.

To elaborate, we show below the implementation of this `modifyRewardToken()` routine. We notice the given argument of `_settings` is not properly validated and allows for an input with its `endTime` smaller than `startTime`. If such an incorrect setting is applied, it will revert every call to `_updatePod()` (Section 3.1).

```
    function modifyRewardToken(address token, uint256 amount, Settings calldata _settings) external nonReentrant checkOwnable {
        RewardToken storage _reward = rewards[token];
        if(!_reward.active) revert PodInvalidReward();
        uint256 _currentTimestamp = _currentBlockTimestamp();

        if (_currentTimestamp > _settings.endTime) {
            revert PodInvalidSettings();
        }

        Settings memory _validatedSettings = Settings({startTime: _settings.startTime, endTime: _settings.endTime});

        if(_reward.settings.endTime < _currentTimestamp) {
            if(_currentTimestamp > _settings.startTime) revert PodInvalidSettings();

            _reward.settings.startTime = _validatedSettings.startTime;
            _reward.accRewardsPerShare = 0;
        }

        if(factory.feeOnAddReward() > 0) {
            uint256 fee = amount.mulDiv(factory.feeOnAddReward(), factory.FEE_BASIS());
            IERC20(token).transferFrom(msg.sender, factory.feeCollector(), fee);
            amount = amount - fee;
        }


        if (token == asset()) {
            _rewardsInUnderlying = _rewardsInUnderlying + amount;
```

```
        }

        IERC20(token).transferFrom(msg.sender, address(this), amount);
        uint256 _newAmount = _reward.remainingAmount + amount;

        _reward.amount = _newAmount;
        _reward.remainingAmount = _newAmount;
        _reward.settings.endTime = _validatedSettings.endTime;

        _updatePod();

        emit ModifyReward(token, _newAmount, _reward.settings.startTime, _reward.
            settings.endTime);
    }
```

Listing 3.2: `Pod:modifyRewardToken()`

Recommendation Revise the above-mentioned routine to ensure `_settings.startTime < _settings.endTime`.

Status This issue has been fixed by the following commit: `19b9e2c`.

3.3 Improved Boundary Handling in Pod

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Pod`
- Category: Coding Practices [3]
- CWE subcategory: CWE-1126 [1]

Description

To facilitate the vault management, `Pod` provides a number of helper routines to add a new reward token or send out reward tokens to protocol users. While reviewing these helper routines, we notice two specific corner cases can be better handled.

The first corner case occurs in the following `addRewardToken()` function, which is used to add a new reward token. Note `Pod` enforces a protocol-wide parameter `MAX_REWARDS_COUNT` – the max amount of rewards accepted at a given time. However, our analysis shows that current implementation allows for `MAX_REWARDS_COUNT + 1` reward tokens.

```
    function addRewardToken(address token, uint256 amount, Settings calldata _settings)
        external
        nonReentrant
        checkOwnable
    {
```

```
        if (token == address(this)  rewards[token].active  _rewardsCount >
            MAX_REWARDS_COUNT) revert PodInvalidReward();
        Settings memory _validatedSettings = _validateSettings(_settings);

        if(factory.feeOnAddReward() > 0) {
            uint256 fee = amount.mulDiv(factory.feeOnAddReward(), factory.FEE_BASIS());
            IERC20(token).transferFrom(msg.sender, factory.feeCollector(), fee);
            amount = amount - fee;
        }
        ...
    }
```

Listing 3.3: `Pod::addRewardToken()`

The second corner case is part of the `_safeRewardsTransfer()` function when a reward token is being deleted. By design, a reward token will be deleted from rewards mapping when a reward token is fully distributed. With that, the related `if`-conditions should be adjusted to be inclusive (lines 408 and 417).

```
402     function _safeRewardsTransfer(IERC20 token, address to, uint256 amount) internal
            virtual {
403         if (amount == 0) return;
404
405         uint256 balance = token.balanceOf(address(this));
406         address _tokenAddress = address(token);
407         // cap to available balance
408         if (amount > balance) {
409             amount = balance;
410
411             if(_tokenAddress != asset()) {
412                 delete rewards[_tokenAddress];
413                 emit DeleteReward(_tokenAddress);
414             }
415         }
416         if (_tokenAddress == asset()) {
417             if (amount > _rewardsInUnderlying) {
418                 amount = _rewardsInUnderlying;
419                 delete rewards[_tokenAddress];
420                 emit DeleteReward(_tokenAddress);
421             }
422             _rewardsInUnderlying = _rewardsInUnderlying - amount;
423         }
424         token.transfer(to, amount);
425     }
```

Listing 3.4: `Pod::_safeRewardsTransfer()`

Recommendation Revise the above-mentioned routines to better handling various corner cases.

Status This issue has been fixed by the following commits: `c9ddfee` and `cb175a7`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Pod` protocol, which is a fully permissionless protocol that enables users to create tokenized vaults by following the `ERC4626` standard with any `ERC20` -like token as underlying. When underlying is deposited, a proportional share to vaults' ratio is minted and can be staked to earn tokens from fixed reward pools according to their time-weighted contribution to the vault. Reward pools may be created by the `Pod` owner - if the `Pod` is ownable - or by any other third-party if is created as permissionless. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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-770: Allocation of Resources Without Limits or Throttling. https://cwe.mitre.org/data/definitions/770.html.

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