SMART CONTRACT AUDIT REPORT

for

LaunchpadStaking

Prepared By: Xiaomi Huang

PeckShield
April 23, 2024

Document Properties

Client	Unchain-X
Title	Smart Contract Audit Report
Target	LaunchpadStaking
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 23, 2024	Xuxian Jiang	Final Release
1.0-rc1	April 23, 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

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

As a staking reward contract, `LaunchpadStaking` follows a typical staking reward system but has a unique feature called the refund option. Basically, the refund option is a selectable value for the percentage of the deposit to be refunded after the staking period ends. Setting the refund option lower allows users to receive additional bonus rewards in addition to the rewards allocated to the staking pool. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The LaunchpadStaking Protocol

Item	Description
Name	Unchain-X
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	April 23, 2024

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

- https://github.com/UNCHAIN-X-Labs/launchpad-staking-contract.git (7d4d738)

1.2 About PeckShield

PeckShield Inc. [9] 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

		Likelihood		
		High	*Medium*	*Low*
Impact	*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 OWASP Risk Rating Methodology [8]:

- 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 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 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

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) [7], 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 `LaunchpadStaking` contract. 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	1	🟦
Informational	1	🟦
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, 1 low-severity vulnerability, and 1 informational suggestion.

Table 2.1: Key LaunchpadStaking Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Suggested Adherence of Checks-Effects-Interactions	Time And State	Confirmed
PVE-002	Low	Configuration Dependency Enforcement in LaunchpadStaking	Coding Practices	Resolved
PVE-003	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

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 Suggested Adherence of Checks-Effects-Interactions

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: LaunchpadStaking
- Category: Time and State [6]
- CWE subcategory: CWE-663 [3]

Description

A common coding best practice in Solidity is the adherence of checks-effects-interactions principle. This principle is effective in mitigating a serious attack vector known as re-entrancy. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the DAO [11] exploit, and the Uniswap/Lendf.Me hack [10].

We notice there is an occasion where the checks-effects-interactions principle is violated. Using the LaunchpadStaking as an example, the _withdrawRefund() function (see the code snippet below) is provided to withdraw users' deposit tokens. However, the invocation of an external contract requires extra care in avoiding the above re-entrancy. For example, the interaction with the external contract (line 626) start before effecting the update on internal state (of batchTAOAmt), hence violating the principle. Fortunately, the use of nonReentrant makes the re-entrancy impossible.

```
623     function _withdrawRefund(address account, address stakingToken) internal returns (
            uint256 refund) {
624         refund = refundOf[account][stakingToken];
625         if(refund > 0) {
626             stakingToken != address(0) ? _transferERC20(account, stakingToken, refund) :
                    _transferETH(account, refund);
627             refundOf[account][stakingToken] = 0;
628         }
```

```
        isWithdrawn[account][stakingToken] = true;
    }
```

Listing 3.1: `LaunchpadStaking::_withdrawRefund()`

Recommendation Revisit the above routine to ensure the adherence of the `checks-effects-interactions` principle and make the re-entrancy impossible. Note current use of `nonReentrant` also makes it impossible for re-entrancy.

Status This issue has been confirmed.

3.2 Configuration Dependency Enforcement in LaunchpadStaking

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low

- Target: `LaunchpadStaking`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

The `LaunchpadStaking` contract allows for on-chain adjustment of various parameters, including `stakingStartBlock`, `stakingEndBlock`, and `bonusRewardSupply`. While reviewing the related setters, we notice inherent dependency among them and these dependency may need to be enforced for improvement.

To elaborate, we show below the implementation of two related setters `setMiningPeriod()` and `setPool()`. The first setter configures `stakingStartBlock` and `stakingEndBlock` while the second setter updates the supported pools. We notice the second setter depends on the first one. If the first one has configured an improper set of `stakingStartBlock` and `stakingEndBlock`, the second setter will always revert in the calculation of `maxValue` (line 382).

```
    function setMiningPeriod(uint256 startBlock, uint256 endBlock) public onlyOwner {
        if(stakingStartBlock > 0) {
            _verifyDeadline(block.number, stakingStartBlock - 1);
        }

        if(startBlock <= block.number) {
            revert BelowStandard(block.number + 1, startBlock);
        }

        if(startBlock >= endBlock) {
            revert BelowStandard(startBlock + 1, endBlock);
        }

        stakingStartBlock = startBlock;
        stakingEndBlock = endBlock;
    }
```

Listing 3.2: `LaunchpadStaking:setMiningPeriod()`

```
    function setPool(PoolConfig calldata params) public onlyOwner {
        if(stakingStartBlock > 0) {
            _verifyDeadline(block.number, stakingStartBlock - 1);
        }

        uint256 prevAllocation = pools[params.stakingToken].allocation;
        uint256 maxValue = (IERC20(rewardToken).balanceOf(address(this)) - ((totalAllocationPerBlock - prevAllocation) * (stakingEndBlock - stakingStartBlock + 1) + bonusRewardSupply)) / (stakingEndBlock - stakingStartBlock + 1);

        if(maxValue < params.allocation) {
            revert OverTheLimit(maxValue, params.allocation);
        }

        totalAllocationPerBlock = totalAllocationPerBlock - prevAllocation + params.allocation;
        pools[params.stakingToken] = PoolInfo(params.allocation, stakingStartBlock, 0, 0, true);

        emit CreatePool(params.stakingToken, params.allocation);
    }
```

Listing 3.3: `LaunchpadStaking:setPool()`

Recommendation Revise the above-mentioned setters to ensure inherent dependency is resolved. Also, current initialization routine allows for repeated invocations, which can be avoided by enforcing one-time initialization only.

Status This issue has been resolved as the team confirms they are part of design.

3.3 Trust Issue Of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium

- Target: `LaunchpadStaking`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the LaunchpadStaking contract, there is a privileged account (owner) that plays a critical role in governing and regulating the protocol-wide operations (e.g., configuring various system parameters, withdraw funds, and pause the contract). In the following, we show the representative functions potentially affected by the privilege of the privileged account.

```
    function withdrawReward(address to, uint256 amount)
        external
        onlyOwner
        nonReentrant
    {
        _transferERC20(to, rewardToken, amount);
    }
    ...
    function pause() external onlyOwner {
        _pause();
        stakingEndBlock = block.number;
    }
    ...
    function setPool(PoolConfig calldata params) public onlyOwner {
        if(stakingStartBlock > 0) {
            _verifyDeadline(block.number, stakingStartBlock - 1);
        }

        uint256 prevAllocation = pools[params.stakingToken].allocation;
        uint256 maxValue = (IERC20(rewardToken).balanceOf(address(this)) - ((totalAllocationPerBlock - prevAllocation) * (stakingEndBlock - stakingStartBlock + 1) + bonusRewardSupply)) / (stakingEndBlock - stakingStartBlock + 1);

        if(maxValue < params.allocation) {
            revert OverTheLimit(maxValue, params.allocation);
        }

        totalAllocationPerBlock = totalAllocationPerBlock - prevAllocation + params.allocation;
        pools[params.stakingToken] = PoolInfo(params.allocation, stakingStartBlock, 0, 0, true);

        emit CreatePool(params.stakingToken, params.allocation);
    }

    /**
     * @dev Set {bonusRewardSupply}.
     * @param amount Amount of bonus reward supply.
     */
    function setBonusRewardSupply(uint256 amount) public onlyOwner {
        _verifyDeadline(block.number, stakingStartBlock > 0 ? stakingStartBlock - 1 : 0);
        uint256 maxValue = IERC20(rewardToken).balanceOf(address(this)) - (totalMiningRewards() - bonusRewardSupply);

        if(maxValue < amount) {
            revert OverTheLimit(maxValue, amount);
        }

        bonusRewardSupply = amount;
    }
```

Listing 3.4: Example Privileged Operations in LaunchpadStaking

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the privileged account is not governed by a DAO-like structure. Note that a compromised account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the protocol design.

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 The issue has been confirmed and will be mitigated with the use of a multi-sig to manage the privileged account.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `LaunchpadStaking` contract, which follows a typical staking reward system but has a unique feature called the refund option. Basically, the refund option is a selectable value for the percentage of the deposit to be refunded after the staking period ends. Setting the refund option lower allows users to receive additional bonus rewards in addition to the rewards allocated to the staking pool. 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[3] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

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

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

[6] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

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

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

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

[10] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[11] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.