SMART CONTRACT AUDIT REPORT

for

LuckyChip Staking

Prepared By: Yiqun Chen

PeckShield

December 9, 2021

Document Properties

Client	LuckyChip
Title	Smart Contract Audit Report
Target	LuckyChip Staking
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	December 9, 2021	Xuxian Jiang	Final Release
1.0-rc1	November 27, 2021	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction **4**

1.1 About LuckyChip Staking . . . 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 Timely massUpdatePools During Pool Multiplier Changes . . . 11

3.2 Sandwiched updatePower() For Higher Quantity . . . 12

3.3 Trust Issue of Admin Keys . . . 14

4 Conclusion **16**

References **17**

1 Introduction

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

`LuckyChip` is a `Defi Casino` that everyone can `play-to-win` and `bank-to-earn`. The protocol designs an incentive mechanism to reward the betting by `Bet Mining`. As compared to traditional `Farming` projects, the protocol rewards users by their unclaimed rewards, which is called as `LuckyPower`.

The basic information of audited contracts is as follows:

Table 1.1: Basic Information of LuckyChip Staking

Item	Description
Target	LuckyChip Staking
Website	https://luckychip.io/
Type	Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	December 9, 2021

In the following, we list the reviewed files and the commit hash values used in this audit.

- https://github.com/luckychip-io/core/blob/master/contracts/pools/BetMining.sol (6345df1)
- https://github.com/luckychip-io/core/blob/master/contracts/pools/LuckyPower.sol (6345df1)

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

- https://github.com/luckychip-io/core/blob/master/contracts/pools/BetMining.sol (37a34d5)
- https://github.com/luckychip-io/core/blob/master/contracts/pools/LuckyPower.sol (37a34d5)

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 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 accordingly classified into four categories, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

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

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 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 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 design and implementation of the `LuckyChip 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 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 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 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Timely massUpdatePools During Pool Multiplier Changes	Business Logic	Fixed
PVE-002	Low	Sandwiched updatePower() For Higher Quantity	Business Logic	Fixed
PVE-003	Medium	Trust Issue of Admin Keys	Security Features	Confirmed

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 Timely massUpdatePools During Pool Multiplier Changes

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: BetMining
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

The LuckyChip Staking protocol has a BetMining contract that provides incentive mechanisms that reward the staking of supported assets. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. And staking users are rewarded in proportional to their share of LP tokens in the reward pool.

The multiplier of supported pools can be dynamically adjusted via updateMultiplier(). When analyzing the pool multiplier update routine updateMultiplier(), we notice the need of timely invoking massUpdatePools() to update the reward distribution before the new pool weight becomes effective.

```
function updateMultiplier(uint256 multiplierNumber) public onlyOwner {
    BONUS_MULTIPLIER = multiplierNumber;
}
```

Listing 3.1: BetMining::updateMultiplier()

If the call to massUpdatePools() is not immediately invoked before updating the pool multiplier, certain situations may be crafted to create an unfair reward distribution. Fortunately, this interface is restricted to the owner (via the onlyOwner modifier), which greatly alleviates the concern.

Recommendation Timely invoke massUpdatePools() when any pool's multiplier has been updated.

Status This issue has been fixed in the commit: 37a34d5.

3.2 Sandwiched updatePower() For Higher Quantity

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

Description

The LuckyChip Staking protocol has a LuckyPower contract that provides an incentive mechanism that rewards the unclaimed rewards from other farming pools. The reward is calculated as 100% x "Unclaimed LC"+ 50% x "staked LC", where "Unclaimed LC" is the LC tokens earned from Farming and Bet Mining but not been claimed yet, "Staked LC" is the LC tokens staked in Liquidity Pool or Dice Table Banker.

Our analysis shows the current incentive mechanism logic of providing rewards based on amount of "staked LC" could be sandwiched for higher quantity, thus higher rewards. To elaborate, we show below the related routines for the quantity calculation.

```
    function updatePower(address account) public override{
        require(account != address(0), "LuckyPower: account is zero address");

        for(uint256 i = 0; i < bonusInfo.length; i ++){
            BonusInfo storage bonus = bonusInfo[i];
            if(bonus.token != address(lcToken)){
                oracle.update(bonus.token, address(lcToken));
            }
        }

        UserInfo storage user = userInfo[account];
        addPendingRewards(account);

        uint256 tmpQuantity = user.quantity;
        uint256 newQuantity = 0;
        if(address(masterChef) != address(0) && address(oracle) != address(0)){
            (address[] memory tokens, uint256[] memory amounts, uint256[] memory
                pendingLcAmounts, uint256 devPending, uint256 poolLength) = masterChef.
                getLuckyPower(account);
            uint256 tmpLpQuantity = 0;
            uint256 tmpBankerQuantity = 0;
            uint256 tmpValue = 0;
            for(uint256 i = 0; i < poolLength; i ++){
                if(amounts[i] > 0){
                    if(EnumerableSet.contains(_lpTokens, tokens[i])){
                        tmpValue = oracle.getLpTokenValue(tokens[i], amounts[i]);
                        tmpLpQuantity = tmpLpQuantity.add(tmpValue.mul(lpPercent).div(
                            PERCENT_DEC)).add(pendingLcAmounts[i]);
```

```
207                        newQuantity = newQuantity.add(tmpValue.mul(lpPercent).div(PERCENT_DEC)).add(pendingLcAmounts[i]);
208                    }else if(EnumerableSet.contains(_diceTokens, tokens[i])){
209                        tmpValue = oracle.getDiceTokenValue(tokens[i], amounts[i]);
210                        tmpBankerQuantity = tmpBankerQuantity.add(tmpValue).add(pendingLcAmounts[i]);
211                        newQuantity = newQuantity.add(tmpValue).add(pendingLcAmounts[i]);
212                    }
213                }
214            }
215            user.lpQuantity = tmpLpQuantity;
216            user.bankerQuantity = tmpBankerQuantity;
217            if(devPending > 0){
218                newQuantity = newQuantity.add(devPending);
219            }
220        }else{
221            user.bankerQuantity = 0;
222            user.lpQuantity = 0;
223        }
224        ...
```

Listing 3.2: `LuckyPower::updatePower()`

We notice the `tmpValue` is calculated by `oracle.getLpTokenValue(tokens[i], amounts[i])` (line 205), where `amounts[i]` is derived from the staked token amounts from `MasterChef`. However, a bad actor could stake large amount tokens into the `Masterchef` before the calling of `updateBonus()` and getting a higher `amounts[i]` when calculating the reward from `LuckyPower()`. Then the bad actor would unstake the large amount of tokens from `Masterchef` afterwards.

Recommendation Only take `pendingLcAmounts` from `MasterChef` into the `LuckyPower` rewards calculation.

Status This issue has been fixed in the commit: 37a34d5.

3.3 Trust Issue of Admin Keys

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

Description

In the `LuckyChip Staking` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the system-wide operations (e.g. parameter setting). 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.

To elaborate, we show below the `set()` routine in the `BetMining` contract. This routine allows the `owner` account to adjust `allocPoint`, which could result different amounts of rewards received by each pool.

```
    // Update the given pool's reward token allocation point. Can only be called by the
        owner.
    function set(uint256 _pid, uint256 _allocPoint) public onlyOwner validPool(_pid) {
        massUpdatePools();

        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint
            );
        poolInfo[_pid].allocPoint = _allocPoint;
    }
```

Listing 3.3: `BetMining::set()`

We emphasize that the privilege assignments are necessary and required for proper protocol operations. However, it is worrisome if the `owner` is not governed by a `DAO`-like structure. 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. 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 confirmed by the team. The team clarifies that they will transfer the privileged owner account to the `TimeLock` contract in the future.

4 Conclusion

In this audit, we have analyzed the `LuckyChip Staking` design and implementation. The protocol designs a incentive mechanism to reward the betting by `Bet Mining`. As compared to traditional `Farming` projects, the protocol rewards users by their unclaimed rewards. The current code base is clearly organized and those identified issues are promptly confirmed and resolved.

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.