SMART CONTRACT AUDIT REPORT

for

Lucky Dice

Prepared By: Yiqun Chen

PeckShield

September 5, 2021

Document Properties

Client	LuckyChip
Title	Smart Contract Audit Report
Target	Lucky Dice
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	September 5, 2021	Xuxian Jiang	Final Release
1.0-rc1	September 1, 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

Given the opportunity to review the design document and related smart contract source code of the Lucky Dice 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 Lucky Dice

LuckyChip is a Defi Casino that everyone can play-to-win and bank-to-earn. Users can participate as Player or Banker in the PLAY part of LuckyChip. In each game, a small amount of betting reward is collected from the winners as Lucky Bonus. Lucky Bonus is the only income of the LuckyChip protocol, and will be totally distributed to all LC builders. The first game in the PLAY part is Lucky Dice.

The basic information of audited contracts is as follows:

Table 1.1: Basic Information of Lucky Dice

Item	Description
Target	Lucky Dice
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	September 5, 2021

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

- https://github.com/luckychip-io/dice/blob/master/contracts/Dice.sol (70e4405)
- https://github.com/luckychip-io/staking/blob/master/contracts/MasterChef.sol (23e5db6)

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/dice/blob/master/contracts/Dice.sol (de3090c)
- https://github.com/luckychip-io/staking/blob/master/contracts/MasterChef.sol (6e43aa1)

1.2 About PeckShield

PeckShield Inc. [12] 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 [11]:

- 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) [10], 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 Lucky Dice 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	1	■
Medium	2	■ ■
Low	5	■ ■ ■ ■ ■
Informational	1	■
Total	9	

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 high-severity vulnerability, 2 medium-severity vulnerabilities, 5 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Predictable Results For Dice Rolling	Business Logic	Fixed
PVE-002	Medium	Logic Error For MaxExposure Limit Check	Business Logic	Fixed
PVE-003	Low	Improved Validation of manualStartRound()	Coding Practices	Fixed
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-005	Informational	Suggested Event Generation For setAdmin()/setBlocks()	Coding Practices	Fixed
PVE-006	Low	Possible Sandwich/MEV Attacks For Reduced Return	Time and State	Fixed
PVE-007	Low	Timely massUpdatePools During Pool Weight Changes	Business Logic	Fixed
PVE-008	Low	Duplicate Pool/Bonus Detection and Prevention	Business Logics	Fixed
PVE-009	Low	Incompatibility With Deflationary Tokens	Business Logics	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 Predictable Results For Dice Rolling

- ID: PVE-001
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `Dice`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

In the `Dice` contract, there is an `Admin` account acting as croupier for the game. The `Admin` plays a critical role in starting/ending a dice rolling round and sending the `secret` to reveal the dice rolling result. To elaborate, we show below the `sendSecret()` and `_safeSendSecret()` routines in the `Dice` contract.

```
    function sendSecret(uint256 epoch, uint256 bankSecret) public onlyAdmin whenNotPaused{
        Round storage round = rounds[epoch];
        require(round.lockBlock != 0, "End round after round has locked");
        require(round.status == Status.Lock, "End round after round has locked");
        require(block.number >= round.lockBlock, "Send secret after lockBlock");
        require(block.number <= round.lockBlock.add(intervalBlocks), "Send secret within intervalBlocks");
        require(round.bankSecret == 0, "Already revealed");
        require(keccak256(abi.encodePacked(bankSecret)) == round.bankHash, "Bank reveal not matching commitment");

        _safeSendSecret(epoch, bankSecret);
        _calculateRewards(epoch);
    }

    function _safeSendSecret(uint256 epoch, uint256 bankSecret) internal whenNotPaused {
        Round storage round = rounds[epoch];
        round.secretSentBlock = block.number;
        round.bankSecret = bankSecret;
        uint256 random = round.bankSecret ^ round.betUsers ^ block.difficulty;
```

```
        round.finalNumber = uint32(random % 6);
        round.status = Status.Claimable;

        emit SendSecretRound(epoch, block.number, bankSecret, round.finalNumber);
    }
```

Listing 3.1: `dice::sendSecret()and dice::_safeSendSecret()`

Before each round, the `Admin` will provide a hashed `secret` and the value will be stored at `round.bankHash`. After the round is locked, the `Admin` will send the `bankSecret` by calling `sendSecret()` to check if the hashed value of `bankSecret` matches the the stored `round.bankHash`, and then it would trigger the `_safeSendSecret()` to reveal the `finalNumber`. However, if we take a close look at `_safeSendSecret()`, this specific routine computes the `round.finalNumber` based on a random number generated from `round.bankSecret ^ round.betUsers ^ block.difficulty`. Since the `round.bankSecret` is provided by the `Admin`, the `block.difficulty` is hard-coded in certain blockchains (e.g. `BSC`), and the `round.betUsers` is possibly colluding with `Admin`, the result for the dice rolling may become predictable. If so, the game will become unfair and `Banker`'s funds may be be drained round by round as the `Admin` would inform the colluding users to bet a maximum amount allowed on the `finalNumber`.

Recommendation Add the `block.timestamp` to feed the random seed.

Status This issue has been fixed in the commit: de3090c. Although there is no real randomness on Ethereum, the change could ensure that the `Dice Rolling` results are not predictable from the `Admin`'s side.

3.2 Logic Error For MaxExposure Limit Check

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Dice`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

There are two roles of users in the `Lucky Dice` contract: `Banker` and `Player`. In `Banker` time, the users can bank/unbank certain tokens into the protocol to receive LP tokens. In `Player` time, the users can bet on the dice rolling result and claim the betting rewards if they bet on the correct `finalNumber`.

However, since the betting rewards would be 5 times the amount of the user's betting amounts, if we do not limit the user's betting amounts, the banker may face a big lost and what's more, the protocol may fail to pay the rewards to the winners.

While reviewing the `betNumber()` routine, we do see there are some logic checks that are in place to constrain the `betAmount` by checking if the banker's `maxExposureRatio` is exceeded (line 292 from `betNumber()`). However, there is a missing multiplication of 5 for the `betAmount` so the current limitation may not work properly in preventing above situation.

```
272     function betNumber(bool[6] calldata numbers, uint256 amount) external payable
            whenNotPaused notContract nonReentrant {
273         Round storage round = rounds[currentEpoch];
274         require(msg.value >= feeAmount, "msg.value > feeAmount");
275         require(round.status == Status.Open, "Round not Open");
276         require(block.number > round.startBlock && block.number < round.lockBlock, "
                Round not bettable");
277         require(ledger[currentEpoch][msg.sender].amount == 0, "Bet once per round");
278         uint16 numberCount = 0;
279         uint256 maxSingleBetAmount = 0;
280         for (uint32 i = 0; i < 6; i ++) {
281             if (numbers[i]) {
282                 numberCount = numberCount + 1;
283                 if(round.betAmounts[i] > maxSingleBetAmount){
284                     maxSingleBetAmount = round.betAmounts[i];
285                 }
286             }
287         }
288         require(numberCount > 0, "numberCount > 0");
289         require(amount >= minBetAmount.mul(uint256(numberCount)), "BetAmount >=
                minBetAmount * numberCount");
290         require(amount <= round.maxBetAmount.mul(uint256(numberCount)), "BetAmount <=
                round.maxBetAmount * numberCount");
291         if(numberCount == 1){
292             require(maxSingleBetAmount.add(amount).sub(round.totalAmount.sub(
                    maxSingleBetAmount)) < bankerAmount.mul(maxExposureRatio).div(TOTAL_RATE
                    ), 'MaxExposure Limit');
293         }
294         ...
295     }
```

Listing 3.2: `Dice::betNumber()`

Recommendation Improved the `betNumber()` routine to properly check `BetAmount` against `maxExposureRatio`.

Status This issue has been fixed in the commit: de3090c.

3.3 Improved Validation Of manualStartRound()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Dice`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1041 [1]

Description

In the `Dice` contract, there is a public function `manualStartRound()` which is used by the `Admin` of the contract to start a new round manually. To elaborate, we show below the related code snippet.

```
function manualStartRound(bytes32 bankHash) external onlyAdmin whenNotPaused {
    require(block.number >= rounds[currentEpoch].lockBlock, "Manual start new round
        after current round lock");
    currentEpoch = currentEpoch + 1;
    _startRound(currentEpoch, bankHash);
}
```

Listing 3.3: `Dice::manualStartRound()`

```
207     // Start the next round n, lock for round n-1
208     function executeRound(uint256 epoch, bytes32 bankHash) external onlyAdmin
            whenNotPaused{
209         require(epoch == currentEpoch, "epoch == currentEpoch");
210
211         // CurrentEpoch refers to previous round (n-1)
212         lockRound(currentEpoch);
213
214         // Increment currentEpoch to current round (n)
215         currentEpoch = currentEpoch + 1;
216         _startRound(currentEpoch, bankHash);
217         require(rounds[currentEpoch].startBlock < playerEndBlock, "startBlock <
                playerEndBlock");
218         require(rounds[currentEpoch].lockBlock <= playerEndBlock, "lockBlock <
                playerEndBlock");
219     }
```

Listing 3.4: `Dice::executeRound()`

It comes to our attention that the `manualStartRound()` function has the inherent assumption that the `Player`'s time is not ended. However, this is only enforced inside the `executeRound()` function (line 217). We suggest to add the `rounds[currentEpoch].startBlock < playerEndBlock` check also in the `manualStartRound()` function.

Recommendation Improve the validation of of `manualStartRound()` following above suggestion.

Status This issue has been fixed in the commit: de3090c.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Dice
- Category: Security Features [6]
- CWE subcategory: CWE-287 [2]

Description

In the Dice protocol, there is a privileged Admin account that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting and game management). 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 setRatios() routine in the Dice contract. This routine allows the Admin account to adjust the maxBetRatio and maxExposureRatio without any limitations.

```
    function setRatios(uint256 _maxBetRatio, uint256 _maxExposureRatio) external
        onlyAdmin {
        maxBetRatio = _maxBetRatio;
        maxExposureRatio = _maxExposureRatio;
        emit RatiosUpdated(block.number, maxBetRatio, maxExposureRatio);
    }
```

Listing 3.5: Dice::setRatios()

We emphasize that the privilege assignments are necessary and required for proper protocol operations. However, it is worrisome if the Admin is not governed by a DAO-like structure. We point out that a compromised Admin account would set the value of maxExposureRatio to TOTAL_RATE, which puts the Banker's funds in big risk. Note that a multi-sig account or adding the maximum limitation of these parameters 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. And add the limitation of maximum value for maxBetRatio and maxExposureRatio.

Status This issue has been fixed in the commit: de3090c.

3.5 Suggested Event Generation For setAdmin()

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Dice
- Category: Coding Practices [7]
- CWE subcategory: CWE-563 [3]

Description

In Ethereum, the event is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an event is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. Events can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed.

While examining the events that reflect the Dice dynamics, we notice there is a lack of emitting an event to reflect adminAddress changes and playerTimeBlocks changes. To elaborate, we show below the related code snippet of the contract.

```
    // set admin address
    function setAdmin(address _adminAddress, address _lcAdminAddress) external onlyOwner {
        require(_adminAddress != address(0) && _lcAdminAddress != address(0), "Cannot be zero address");
        adminAddress = _adminAddress;
        lcAdminAddress = _lcAdminAddress;
    }
```

Listing 3.6: Dice::setAdmin()

```
    // set blocks
    function setBlocks(uint256 _intervalBlocks, uint256 _playerTimeBlocks, uint256 _bankerTimeBlocks) external onlyAdmin {
        intervalBlocks = _intervalBlocks;
        playerTimeBlocks = _playerTimeBlocks;
        bankerTimeBlocks = _bankerTimeBlocks;
    }
```

Listing 3.7: Dice::setBlocks()

Recommendation Properly emit the above-mentioned events with accurate information to timely reflect state changes. This is very helpful for external analytics and reporting tools.

Status This issue has been fixed in the commit: de3090c.

3.6 Possible Sandwich/MEV Attacks For Reduced Returns

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Dice`
- Category: Time and State [9]
- CWE subcategory: CWE-682 [4]

Description

The `Dice` contract has a helper routine, i.e., `_calculateRewards()`, that is designed to calculate rewards for a round. It has a rather straightforward logic in swapping the `token` to `lcToken` when calculating the `lcBackAmount`.

```
function _calculateRewards(uint256 epoch) internal {
  require(lcBackRate.add(bonusRate) <= TOTAL_RATE, "lcBackRate + bonusRate <=
      TOTAL_RATE");
  require(rounds[epoch].bonusAmount == 0, "Rewards calculated");
  Round storage round = rounds[epoch];
  ...
      if(address(token) == address(lcToken)){
          round.swapLcAmount = lcBackAmount;
      }else if(address(swapRouter) != address(0)){
          address[] memory path = new address[](2);
          path[0] = address(token);
          path[1] = address(lcToken);
          uint256 lcAmout = swapRouter.swapExactTokensForTokens(round.lcBackAmount, 0,
               path, address(this), block.timestamp + (5 minutes))[1];
          round.swapLcAmount = lcAmout;
      }
      totalBonusAmount = totalBonusAmount.add(bonusAmount);
      ...
}
```

Listing 3.8: `Dice::_calculateRewards()`

To elaborate, we show above the `_calculateRewards()` routine. We notice the token swap is routed to `swapRouter` and the actual swap operation `swapExactTokensForTokens()` essentially does not specify any restriction (with `amountOutMin=0`) on possible slippage and is therefore vulnerable to possible front-running attacks, resulting in a smaller gain for this round of yielding.

Note that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user because the swap rate is lowered by the preceding sell. As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade or referencing the `TWAP` or `time-weighted average price` of `UniswapV2`. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

Recommendation Develop an effective mitigation to the above front-running attack to better protect the interests of farming users.

Status This issue has been fixed in the commit: de3090c.

3.7 Timely massUpdatePools During Pool Weight Changes

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `MasterChef`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

As mentioned in Section 3.6, the `Dice` protocol 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 reward pools can be dynamically added via `add()` and the weights of supported pools can be adjusted via `set()`. When analyzing the pool weight update routine `set()`, we notice the need of timely invoking `massUpdatePools()` to update the reward distribution before the new pool weight becomes effective.

```
// Update the given pool's LC allocation point. Can only be called by the owner.
function set( uint256 _pid, uint256 _allocPoint, bool _withUpdate) public onlyOwner {
    if (_withUpdate) {
        massUpdatePools();
    }
    totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint);
    poolInfo[_pid].allocPoint = _allocPoint;
}
```

Listing 3.9: `MasterChef::set()`

If the call to `massUpdatePools()` is not immediately invoked before updating the pool weights, certain situations may be crafted to create an unfair reward distribution. Moreover, a hidden pool without any weight can suddenly surface to claim unreasonable share of rewarded tokens. 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 weight has been updated. In fact, the third parameter (`_withUpdate`) to the `set()` routine can be simply ignored or removed.

```
// Update the given pool's LC allocation point. Can only be called by the owner.
function set( uint256 _pid, uint256 _allocPoint, bool _withUpdate) public onlyOwner {
    massUpdatePools();
    totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint);
    poolInfo[_pid].allocPoint = _allocPoint;
}
```

Listing 3.10: `MasterChef::set()`

Status This issue has been fixed in the commit: 6e43aa1.

3.8 Duplicate Pool/Bonus Detection and Prevention

- ID: PVE-008
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `MasterChef`
- Category: Business Logics [8]
- CWE subcategory: CWE-841 [5]

Description

The `MasterChef` protocol provides incentive mechanisms that reward the staking of supported assets with certain reward tokens. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. Each pool has its `allocPoint*100%/totalAllocPoint` share of scheduled rewards and the rewards for stakers are proportional to their share of LP tokens in the pool.

In current implementation, there are a number of concurrent pools that share the rewarded tokens and more can be scheduled for addition (via a proper governance procedure). To accommodate these new pools, the design has the necessary mechanism in place that allows for dynamic additions of new staking pools that can participate in being incentivized as well.

The addition of a new pool is implemented in `add()`, whose code logic is shown below. It turns out it did not perform necessary sanity checks in preventing a new pool but with a duplicate token from being added. Though it is a privileged interface (protected with the modifier `onlyOwner`), it is still desirable to enforce it at the smart contract code level, eliminating the concern of wrong pool introduction from human omissions.

```
// Add a new lp to the pool. Can only be called by the owner.
// XXX DO NOT add the same LP token more than once. Rewards will be messed up if you do.
function add(uint256 _allocPoint, uint256 _bonusPoint, IBEP20 _lpToken, bool _withUpdate) public onlyOwner {
    if (_withUpdate) {
        massUpdatePools();
    }
    uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
    totalAllocPoint = totalAllocPoint.add(_allocPoint);
    totalBonusPoint = totalBonusPoint.add(_bonusPoint);

    poolInfo.push(
        PoolInfo({
            lpToken: _lpToken,
            allocPoint: _allocPoint,
            bonusPoint: _bonusPoint,
            lastRewardBlock: lastRewardBlock,
            accLCPerShare: 0
        })
    );
}
```

Listing 3.11: `MasterChef::add()`

Recommendation Detect whether the given pool for addition is a duplicate of an existing pool. The pool addition is only successful when there is no duplicate.

```
    function checkPoolDuplicate(IBEP20 _lpToken) public {
        uint256 length = poolInfo.length;
        for (uint256 pid = 0; pid < length; ++pid) {
            require(poolInfo[_pid].lpToken != _lpToken, "add: existing pool?");
        }
    }

    // Add a new lp to the pool. Can only be called by the owner.
    // XXX DO NOT add the same LP token more than once. Rewards will be messed up if you do.
    function add(uint256 _allocPoint, IBEP20 _lpToken, bool _withUpdate) public onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }
        checkPoolDuplicate(_lpToken);
        uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
        totalAllocPoint = totalAllocPoint.add(_allocPoint);
        totalBonusPoint = totalBonusPoint.add(_bonusPoint);

        poolInfo.push(
            PoolInfo({
                lpToken: _lpToken,
                allocPoint: _allocPoint,
```

```
                bonusPoint: _bonusPoint,
                lastRewardBlock: lastRewardBlock,
                accLCPerShare: 0
            })
        );
    }
```

Listing 3.12: Revised `MasterChef::add()`

We point out that if a new pool with a duplicate LP token can be added, it will likely cause a havoc in the distribution of rewards to the pools and the stakers. Note that `addBonus()` shares the very same issue.

Status This issue has been fixed in the commit: 6e43aa1.

3.9 Incompatibility with Deflationary Tokens

- ID: PVE-009
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `MasterChef`
- Category: Business Logics [8]
- CWE subcategory: CWE-841 [5]

Description

In the `LuckyChip` protocol, the `MasterChef` contract is designed to take user's asset and deliver rewards depending on the user's share. In particular, one interface, i.e., `deposit()`, accepts asset transfer-in and records the depositor's balance. Another interface, i.e, `withdraw()`, allows the user to withdraw the asset with necessary bookkeeping under the hood. For the above two operations, i.e., `deposit()` and `withdraw()`, the contract using the `safeTransferFrom()` routine to transfer assets into or out of its pool. This routine works as expected with standard ERC20 tokens: namely the pool's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

```
function deposit(uint256 _pid, uint256 _amount, address _referrer) public nonReentrant {
  PoolInfo storage pool = poolInfo[_pid];
  UserInfo storage user = userInfo[_pid][msg.sender];
  updatePool(_pid);
  if(_amount > 0 && address(luckychipReferral) != address(0) && _referrer != address(0) && _referrer != msg.sender){
      luckychipReferral.recordReferral(msg.sender, _referrer);
  }
  payPendingLC(_pid, msg.sender);
  if (pool.bonusPoint > 0){
```

```
        payPendingBonus(_pid, msg.sender);
}
    if(_amount > 0){
        pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
        user.amount = user.amount.add(_amount);
    }
    user.rewardDebt = user.amount.mul(pool.accLCPerShare).div(1e12);
    if (pool.bonusPoint > 0){
        for(uint256 i = 0; i < bonusInfo.length; i ++) {
    userBonusDebt[i][msg.sender] = user.amount.mul(poolBonusPerShare[_pid][i]).div(1e12);
        }
    }

    emit Deposit(msg.sender, _pid, _amount);
}

// Withdraw LP tokens from MasterChef.
function withdraw(uint256 _pid, uint256 _amount) public nonReentrant {

    PoolInfo storage pool = poolInfo[_pid];
    UserInfo storage user = userInfo[_pid][msg.sender];
    require(user.amount >= _amount, "withdraw: not good");
    updatePool(_pid);
    payPendingLC(_pid, msg.sender);
    if (pool.bonusPoint > 0){
        payPendingBonus(_pid, msg.sender);
}
    if(_amount > 0){
        user.amount = user.amount.sub(_amount);
        pool.lpToken.safeTransfer(address(msg.sender), _amount);
    }
    user.rewardDebt = user.amount.mul(pool.accLCPerShare).div(1e12);
    if (pool.bonusPoint > 0){
        for(uint256 i = 0; i < bonusInfo.length; i ++) {
    userBonusDebt[i][msg.sender] = user.amount.mul(poolBonusPerShare[_pid][i]).div(1e12);
        }
    }
    emit Withdraw(msg.sender, _pid, _amount);
}
```

Listing 3.13: `MasterChef::deposit()`and `MasterChef::withdraw()`

However, there exist other ERC20 tokens that may make certain customization to their ERC20 contracts. One type of these tokens is deflationary tokens that charge certain fee for every transfer or transferFrom. As a result, this may not meet the assumption behind asset-transferring routines. In other words, the above operations, such as `deposit()` and `withdraw()`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts. Apparently, these balance inconsistencies are damaging to accurate and precise portfolio management of the pool and affects protocol-wide operation and maintenance.

Specially, if we take a look at the `updatePool()` routine. This routine calculates `pool.accLCPerShare` via dividing `LCReward` by `lpSupply`, where the `lpSupply` is derived from `pool.lpToken.balanceOf(address(this))` (line 259). Because the balance inconsistencies of the pool, the `lpSupply` could be 1 `Wei` and may give a big `pool.accLCPerShare` as the final result, which dramatically inflates the pool's reward.

```
253 // Update reward variables of the given pool to be up-to-date.
254 function updatePool(uint256 _pid) public {
255     PoolInfo storage pool = poolInfo[_pid];
256     if (block.number <= pool.lastRewardBlock) {
257         return;
258     }
259     uint256 lpSupply = pool.lpToken.balanceOf(address(this));
260     if (lpSupply <= 0) {
261         pool.lastRewardBlock = block.number;
262         return;
263     }
264     uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
265     uint256 LCReward = multiplier.mul(LCPerBlock).mul(pool.allocPoint).div(
        totalAllocPoint).mul(stakingPercent).div(percentDec);
266     LC.mint(address(this), LCReward);
267     pool.accLCPerShare = pool.accLCPerShare.add(LCReward.mul(1e12).div(lpSupply));
268     pool.lastRewardBlock = block.number;
269 }
```

Listing 3.14: `MasterChef::updatePool()`

One mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of bluntly assuming the amount parameter in `safeTransfer` or `safeTransferFrom` will always result in full transfer, we need to ensure the increased or decreased amount in the pool before and after the `safeTransfer` or `safeTransferFrom` is expected and aligned well with our operation. Though these additional checks cost additional gas usage, we consider they are necessary to deal with deflationary tokens or other customized ones if their support is deemed necessary.

Another mitigation is to regulate the set of ERC20 tokens that are permitted into Lucky Dice for indexing. However, certain existing stable coins may exhibit control switches that can be dynamically exercised to convert into deflationary.

Recommendation Check the balance before and after the `safeTransfer()` or `safeTransferFrom()` call to ensure the book-keeping amount is accurate. An alternative solution is using non-deflationary tokens as collateral but some tokens (e.g., USDT) allow the admin to have the deflationary-like features kicked in later, which should be verified carefully.

Status This issue has been confirmed.

4 Conclusion

In this audit, we have analyzed the `Lucky Dice` design and implementation. The system presents a unique `play-to-win`, `bank-to-earn Defi Casino` on blockchain, where users can participate in as `Player` or `Banker`. 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-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

[2] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[3] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[4] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

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

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

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

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

[9] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

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

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

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