Revenue Sharing Pool & Shield

Smart Contract Audit Report
Prepared for LuckyLion

Date Issued:	Nov 29, 2021
Project ID:	AUDIT2021037
Version:	v2.0
Confidentiality Level:	Public

inspeX
CYBERSECURITY PROFESSIONAL SERVICE

Report Information

Project ID	AUDIT2021037
Version	v2.0
Client	LuckyLion
Project	Revenue Sharing Pool & Shield
Auditor(s)	Weerawat Pawanawiwat Patipon Suwanbol
Author	Patipon Suwanbol
Reviewer	Suvicha Buakhom
Confidentiality Level	Public

Version History

Version	Date	Description	Author(s)
2.0	Nov 29, 2021	Update reassessment details	Patipon Suwanbol
1.0	Nov 26, 2021	Full report	Patipon Suwanbol

Contact Information

Company	Inspex
Phone	(+66) 90 888 7186
Telegram	t.me/inspexco
Email	audit@inspex.co

Table of Contents

1. Executive Summary

As requested by LuckyLion, Inspex team conducted an audit to verify the security posture of the Revenue Sharing Pool & Shield smart contracts between Sep 26, 2021 and Sep 27, 2021. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of Revenue Sharing Pool & Shield smart contracts. Static code analysis, dynamic analysis, and manual review were done in conjunction to identify smart contract vulnerabilities together with technical & business logic flaws that may be exposed to the potential risk of the platform and the ecosystem. Practical recommendations are provided according to each vulnerability found and should be followed to remediate the issue.

1.1. Audit Result

In the initial audit, Inspex found 2 medium, 1 low, 2 very low, and 4 info-severity issues. With the project team's prompt response, 2 medium, 2 very low and 3 info-severity issues were resolved in the reassessment, while 1 low and 1 info-severity issues were acknowledged by the team. Therefore, Inspex trusts that Revenue Sharing Pool & Shield smart contracts have sufficient protections to be safe for public use. However, in the long run, Inspex suggests resolving all issues found in this report.

1.2. Disclaimer

This security audit is not produced to supplant any other type of assessment and does not guarantee the discovery of all security vulnerabilities within the scope of the assessment. However, we warrant that this audit is conducted with goodwill, professional approach, and competence. Since an assessment from one single party cannot be confirmed to cover all possible issues within the smart contract(s), Inspex suggests conducting multiple independent assessments to minimize the risks. Lastly, nothing contained in this audit report should be considered as investment advice.

2. Project Overview

2.1. Project Introduction

LuckyLion is the latest addition to the portfolio of APAC's leading iGaming brands with over 200,000 loyal monthly active users, allowing players to yield users' tokens on the decentralized yield farm, play industry leading iGaming, and stake the reward through the revenue sharing pool to earn even more amazing rewards.

Revenue Sharing Pool is one of the new features introduced by LuckyLion. It allows the platform users to invest in the platform and receive the platform profit as a share in return.

Scope Information:

Project Name	Revenue Sharing Pool & Shield
Website	https://app.luckylion.io/revenue
Smart Contract Type	Ethereum Smart Contract
Chain	Binance Smart Chain
Programming Language	Solidity

Audit Information:

Audit Method	Whitebox
Audit Date	Sep 26, 2021 - Sep 27, 2021
Reassessment Date	Nov 29, 2021

The audit method can be categorized into two types depending on the assessment targets provided:

1. **Whitebox**: The complete source code of the smart contracts are provided for the assessment.
2. **Blackbox**: Only the bytecodes of the smart contracts are provided for the assessment.

2.2. Scope

The following smart contracts were audited and reassessed by Inspex in detail:

2.2.1. Revenue Sharing Pool

Initial Audit (Commit: 342db925e80c9e62fce7505d24431e3d82199b87)

Contract	Location (URL)
RevenueSharingPool	https://github.com/LuckyLionIO/LuckyLion-RevenueSharing/blob/342db925e8/contracts/RevenueSharingPool.sol

Reassessment (Commit: 5b9970794f29d562ad1f219b646bb64afceadfb5)

Contract	Location (URL)
RevenueSharingPool	https://github.com/LuckyLionIO/LuckyLion-RevenueSharing/blob/5b9970794f/contracts/RevenueSharingPool.sol

2.2.2. Shield

Initial Audit (Commit: 5c8e9e7ebd3c897c920675dda3c3621b42d9d0dd)

Contract	Location (URL)
MasterchefShield	https://github.com/LuckyLionIO/Lucky-farm/blob/5c8e9e7ebd/contracts/MasterchefShield.sol

Reassessment (Commit: 5c8e9e7ebd3c897c920675dda3c3621b42d9d0dd)

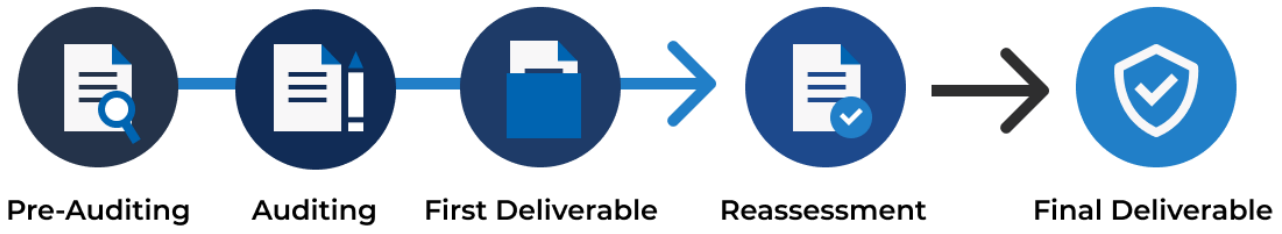
Contract	Location (URL)
MasterchefShield	https://github.com/LuckyLionIO/Lucky-farm/blob/5c8e9e7ebd/contracts/MasterchefShield.sol

The assessment scope covers only the in-scope smart contracts and the smart contracts that they inherit from.

3. Methodology

Inspex conducts the following procedure to enhance the security level of our clients' smart contracts:

1. **Pre-Auditing**: Getting to understand the overall operations of the related smart contracts, checking for readiness, and preparing for the auditing
2. **Auditing**: Inspecting the smart contracts using automated analysis tools and manual analysis by a team of professionals
3. **First Deliverable and Consulting**: Delivering a preliminary report on the findings with suggestions on how to remediate those issues and providing consultation
4. **Reassessment**: Verifying the status of the issues and whether there are any other complications in the fixes applied
5. **Final Deliverable**: Providing a full report with the detailed status of each issue



3.1. Test Categories

Inspex smart contract auditing methodology consists of both automated testing with scanning tools and manual testing by experienced testers. We have categorized the tests into 3 categories as follows:

1. **General Smart Contract Vulnerability (General)** - Smart contracts are analyzed automatically using static code analysis tools for general smart contract coding bugs, which are then verified manually to remove all false positives generated.
2. **Advanced Smart Contract Vulnerability (Advanced)** - The workflow, logic, and the actual behavior of the smart contracts are manually analyzed in-depth to determine any flaws that can cause technical or business damage to the smart contracts or the users of the smart contracts.
3. **Smart Contract Best Practice (Best Practice)** - The code of smart contracts is then analyzed from the development perspective, providing suggestions to improve the overall code quality using standardized best practices.

3.2. Audit Items

The following audit items were checked during the auditing activity.

General
Reentrancy Attack
Integer Overflows and Underflows
Unchecked Return Values for Low-Level Calls
Bad Randomness
Transaction Ordering Dependence
Time Manipulation
Short Address Attack
Outdated Compiler Version
Use of Known Vulnerable Component
Deprecated Solidity Features
Use of Deprecated Component
Loop with High Gas Consumption
Unauthorized Self-destruct
Redundant Fallback Function
Insufficient Logging for Privileged Functions
Invoking of Unreliable Smart Contract
Use of Upgradable Contract Design
Advanced
Business Logic Flaw
Ownership Takeover
Broken Access Control
Broken Authentication
Improper Kill-Switch Mechanism

Improper Front-end Integration
Insecure Smart Contract Initiation
Denial of Service
Improper Oracle Usage
Memory Corruption
Best Practice
Use of Variadic Byte Array
Implicit Compiler Version
Implicit Visibility Level
Implicit Type Inference
Function Declaration Inconsistency
Token API Violation
Best Practices Violation

3.3. Risk Rating

OWASP Risk Rating Methodology[1] is used to determine the severity of each issue with the following criteria:

- **Likelihood**: a measure of how likely this vulnerability is to be uncovered and exploited by an attacker.
- **Impact**: a measure of the damage caused by a successful attack

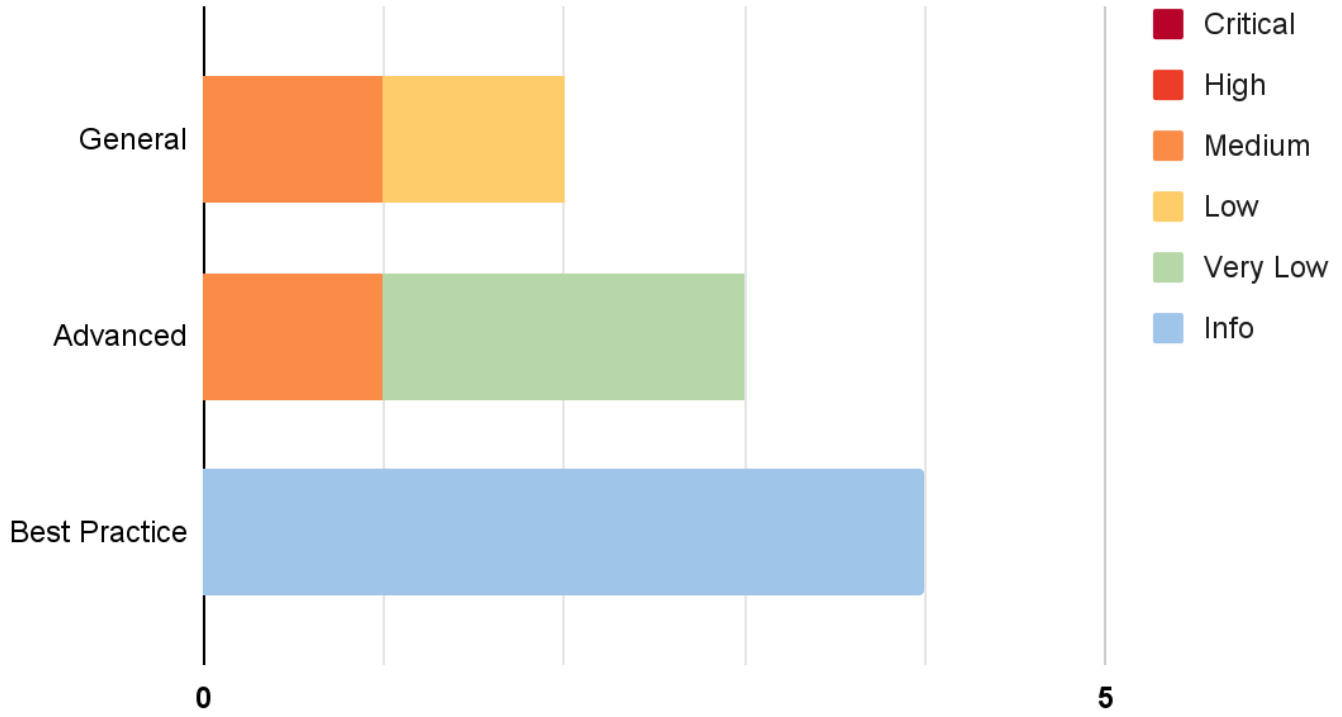
Both likelihood and impact can be categorized into three levels: **Low**, **Medium**, and **High**.

Severity is the overall risk of the issue. It can be categorized into five levels: **Very Low**, **Low**, **Medium**, **High**, and **Critical**. It is calculated from the combination of likelihood and impact factors using the matrix below. The severity of findings with no likelihood or impact would be categorized as **Info**.

Impact \ Likelihood	Low	Medium	High
Low	**Very Low**	**Low**	**Medium**
Medium	**Low**	**Medium**	**High**
High	**Medium**	**High**	**Critical**

4. Summary of Findings

From the assessments, Inspex has found 9 issues in three categories. The following chart shows the number of the issues categorized into three categories: **General**, **Advanced**, and **Best Practice**.



The statuses of the issues are defined as follows:

Status	Description
Resolved	The issue has been resolved and has no further complications.
Resolved *	The issue has been resolved with mitigations and clarifications. For the clarification or mitigation detail, please refer to Chapter 5.
Acknowledged	The issue's risk has been acknowledged and accepted.
No Security Impact	The best practice recommendation has been acknowledged.

The information and status of each issue can be found in the following table:

ID	Title	Category	Severity	Status
IDX-001	Centralized Control of State Variable	General	**Medium**	Resolved
IDX-002	Design Flaw in Reward Calculation	Advanced	**Medium**	Resolved
IDX-003	Loop Over Unbounded Data Structure	General	**Low**	Acknowledged
IDX-004	Unchecked Swapping Path For Reward Token	Advanced	**Very Low**	Resolved
IDX-005	Insufficient Logging for Privileged Functions	Advanced	**Very Low**	Resolved
IDX-006	Unused Function Parameter	Best Practice	**Info**	No Security Impact
IDX-007	Improper Function Visibility	Best Practice	**Info**	Resolved
IDX-008	Inexplicit State Variable Visibility	Best Practice	**Info**	Resolved
IDX-009	Inexplicit Solidity Compiler Version	Best Practice	**Info**	Resolved

* The mitigations or clarifications by LuckyLion can be found in Chapter 5.

5. Detailed Findings Information

5.1 Centralized Control of State Variable

ID	IDX-001
Target	RevenueSharingPool Shield
Category	General Smart Contract Vulnerability
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	**Severity:** Medium **Impact:** Medium The controlling authorities can change the critical state variables to gain additional profit. Thus, it is unfair to the other users. **Likelihood:** Medium Only the contract owner can execute these functions, but there is nothing to restrict the changes from being done.
Status	Resolved LuckyLion team has mitigated this issue by applying deployed `Timelock` contract to the deployed `MasterchefShield` contract. This creates a room for the user to monitor the changes of the contract. `MasterchefShield` contract address is `0xf6d883b0da58171642a27062b096f655f663c4f1`. `Timelock` contract address is `0x4b6c8959a41475347226d51f37ec9a1e09f39a92`. Furthermore, since the `RevenueSharingPool` contract has been modified with a new logic implementation, the `updateMaxDate()` function has no direct impact to the platform's users. Therefore, the `updateMaxDate()` function is no longer considered as an issue.

5.1.1 Description

Critical state variables can be updated any time by the controlling authorities. Changes in these variables can cause impacts to the users, so the users should accept or be notified before these changes are effective.

However, as the contract is not yet deployed, there is potentially no constraint to prevent the authorities from modifying these variables without notifying the users.

The controllable privileged state update functions are as follows:

File	Contract	Function	Modifier
RevenueSharingPool.sol (L: 157)	RevenueSharingPool	updateMaxDate()	onlyOwner
MasterchefShield.sol (L: 14)	Shield	add()	onlyOwner
MasterchefShield.sol (L: 18)	Shield	set()	onlyOwner
MasterchefShield.sol (L: 26)	Shield	updateLuckyPerBlock()	onlyOwner
@openzeppelin/contracts/access/Ownable.sol (L: 53)	RevenueSharingPool, Shield	renounceOwnership()	onlyOwner
@openzeppelin/contracts/access/Ownable.sol (L: 61)	RevenueSharingPool, Shield	transferOwnership()	onlyOwner

5.1.2 Remediation

In the ideal case, the critical state variables should not be modifiable to keep the integrity of the smart contract. However, if modifications are needed, Inspex suggests limiting the use of these functions via the following options:

- Implementing a community-run governance to control the use of these functions
- Using a Timelock contract to delay the changes for a sufficient amount of time

5.2 Design Flaw in Reward Calculation

ID	IDX-002
Target	RevenueSharingPool
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** Medium The reward distribution for the users will be improperly modified, leading to an unfair reward distribution for the users. This results in reputation damage to the platform. **Likelihood:** Medium Only the contract owner can execute this function, and it is possible that the platform will shorten the pool's reward distribution period (`MAX_DATE`).
Status	Resolved LuckyLion team has resolved this issue as suggested in commit `69441c516bd3eaa83d07a928d0d246d83c3a8362` by modifying the logic of the `updateMaxDate()` function. The changing of `MAX_DATE` will have an effect on the current reward distribution round only without affecting the previous rounds.

5.2.1 Description

In `RevenueSharingPool` contract, it allows the users to deposit the LUCKY-BUSD LP token to acquire $LUCKY as the reward. By design, the users will be able to claim the reward only if the round of depositing the LUCKY-BUSD LP token is passed to the new round, which the new round will be decided when the current round surpasses the amount of day required (`MAX_DATE`). This means if the users decide to withdraw the deposited LUCKY-BUSD LP token before the required time, they will not gain any reward.

RevenueSharingPool.sol

```
96  // deposit LUCKY-BUSD LP token
97  function depositToken(uint256 amount) external {
98      UserInfo storage user = userInfo[msg.sender];
99      require(amount > 0, "Insufficient deposit amount!");
100     luckyBusd.safeTransferFrom(msg.sender, address(this), amount);
101     uint256 depositDate = getDepositDate();
102     uint256 roundId = getCurrentRoundId();
103
104     if (!isStakeUpToDate(roundId)) {
105         updatePendingStake();
106     }
107
```

```
108     user.amount += amount;
109     updateStake(roundId, depositDate, amount);
110     emit DepositStake(msg.sender, amount, block.timestamp);
111 }
```

When the users deposit the token, the deposited amount will be registered in `stakeAmount` state, starting from the date that the users start to deposit (`depositDate`) until the last day of the required depositing time (`MAX_DATE`).

RevenueSharingPool.sol

```
186 function updateStake(uint256 roundId, uint256 depositDate, uint256 amount)
    internal {
187     for(uint256 i = depositDate; i <= MAX_DATE; i++) {
188         stakeAmount[roundId][i][msg.sender] += amount;
189         totalStake[roundId][i] += amount;
190     }
191 }
```

However, the platform owner can modify the `MAX_DATE` state variable to set the number of days in each round.

RevenueSharingPool.sol

```
156 // Update max number of day in a round (default 7 days)
157 function updateMaxDate(uint256 newMaxDate) external onlyOwner {
158     MAX_DATE = newMaxDate;
159 }
```

The setting of `MAX_DATE` will affect the reward calculation since it is applied to the reward calculation in the `calculateLuckyReward()` and `getTotalLuckyRewardPerDay()` functions.

RevenueSharingPool.sol

```
222 function calculateLuckyReward(address account, uint256 roundId) internal view
    returns (uint256) {
223     uint256 luckyReward;
224     uint256 totalLuckyRevenuePerDay = getTotalLuckyRewardPerDay(roundId);
225     if (totalLuckyRevenuePerDay == 0) {
226         return 0;
227     }
228     for (uint256 i = 1; i <= MAX_DATE; i++) {
229         uint256 amount = stakeAmount[roundId][i][account];
230         if (amount == 0) continue;
231         uint256 _totalStake = totalStake[roundId][i];
232         uint256 userSharesPerDay = (amount * 1e18) / _totalStake;
233         luckyReward += (totalLuckyRevenuePerDay * userSharesPerDay) / 1e18;
234     }
```

```
235     return luckyReward;
236 }
```

RevenueSharingPool.sol

```
273 // return total LUCKY reward per day of specific round
274 function getTotalLuckyRewardPerDay(uint256 roundId) public view returns
    (uint256) {
275     return (totalLuckyRevenue[roundId] / MAX_DATE);
276 }
```

Hence, with the current design, the reward distribution will be unfairly modified when the `MAX_DATE` is updated.

The following scenario represents the improper reward distribution after the `MAX_DATE` is updated:

Assume that a total reward in the pool is 300 with 3 day in each round (`MAX_DATE` is 3) and the total staked token (`_totalStake`) is 100.

In the round 1:

User A deposits 10 LUCKY-BUSD LP token to the pool on day 1 and another 10 LUCKY-BUSD LP token to the pool on day 3.

`stakeAmount` state:

	Day 1	Day 2	Day 3
User A	10	10	20

After that, the platform owner updates the `MAX_DATE` to 2.

In the round 2:

User A is now eligible to claim the reward of round 1 through the `claimReward()` function.

The calculation in the `calculateLuckyReward()` function is done as follows:

If the `MAX_DATE` is 3 from the `Round 1`:

Day 1 Reward:

```
totalLuckyRevenuePerDay = 300 / 3 = 100
userSharesPerDay = 10 / 100 = 0.1
luckyReward = totalLuckyRevenuePerDay * userSharesPerDay
            = 100 * 0.1 = 10
```

Day 2 Reward:

```
totalLuckyRevenuePerDay = 300 / 3 = 100
userSharesPerDay = 10 / 100 = 0.1
luckyReward = totalLuckyRevenuePerDay * userSharesPerDay
            = 100 * 0.1 = 10
```

Day 3 Reward:

```
totalLuckyRevenuePerDay = 300 / 3 = 100
userSharesPerDay = 20 / 100 = 0.2
luckyReward = totalLuckyRevenuePerDay * userSharesPerDay
            = 100 * 0.2 = 20
```

As a result, the `luckyReward` will be `10 + 10 + 20 = 40`.

If the `MAX_DATE` is 2 since it get updated from the owner:

Day 1 Reward:

```
totalLuckyRevenuePerDay = 300 / 2 = 150
userSharesPerDay = 10 / 100 = 0.1
luckyReward = totalLuckyRevenuePerDay * userSharesPerDay
            = 150 * 0.1 = 15
```

Day 2 Reward:

```
totalLuckyRevenuePerDay = 300 / 2 = 150
userSharesPerDay = 10 / 100 = 0.1
luckyReward = totalLuckyRevenuePerDay * userSharesPerDay
            = 150 * 0.1 = 15
```

As a result, the `luckyReward` will be `15 + 15 = 30`.

This means that the amount that the user deposited on day 3 is ignored, even when that amount should be considered in the reward distribution.

All in all, modifying the `MAX_DATE` state variable affects the reward distribution for the users, causing the distribution to be unfair.

5.2.2 Remediation

Inspex suggests making the reward distribution period (`MAX_DATE`) to have effect on the future rounds only.

5.3 Loop Over Unbounded Data Structure

ID	IDX-003
Target	RevenueSharingPool
Category	General Smart Contract Vulnerability
CWE	CWE-400: Uncontrolled Resource Consumption
Risk	**Severity:** Low **Impact:** Medium When the number of rounds grows to a certain number, the gas required will exceed the block gas limit, causing the transactions that call the `calculateTotalLuckyReward()` function to be reverted. This causes disruption to the service of the platform. **Likelihood:** Low The number of rounds required for the gas usage to grow high enough to exceed the block gas limit is very high.
Status	**Acknowledged** LuckyLion team has acknowledged this issue as it is very unlikely that the number of rounds, which is one of the factors to cause this issue, will grow so much that the gas limit will be exceeded with the current business design.

5.3.1 Description

In the `RevenueSharingPool` contract, the `calculateTotalLuckyReward()` function is used to calculate the total reward of the user by looping through each round.

RevenueSharingPool.sol

```
238 function calculateTotalLuckyReward() internal view returns (uint256) {
239     uint256 totalLuckyReward = 0;
240     uint256 roundId = getCurrentRoundId();
241     for (uint256 i = 0; i < roundId; i++) {
242         totalLuckyReward += calculateLuckyReward(msg.sender, i);
243     }
244     return totalLuckyReward;
245 }
```

The reward of each round is calculated in the `calculateLuckyReward()` function, which loops through all dates of each round.

RevenueSharingPool.sol

```
222 function calculateLuckyReward(address account, uint256 roundId) internal view
    returns (uint256) {
```

```
223     uint256 luckyReward;
224     uint256 totalLuckyRevenuePerDay = getTotalLuckyRewardPerDay(roundId);
225     if (totalLuckyRevenuePerDay == 0) {
226         return 0;
227     }
228     for (uint256 i = 1; i <= MAX_DATE; i++) {
229         uint256 amount = stakeAmount[roundId][i][account];
230         if (amount == 0) continue;
231         uint256 _totalStake = totalStake[roundId][i];
232         uint256 userSharesPerDay = (amount * 1e18) / _totalStake;
233         luckyReward += (totalLuckyRevenuePerDay * userSharesPerDay) / 1e18;
234     }
235     return luckyReward;
236 }
```

These calculations retrieve state variables from the storage, which cost a substantial amount of gas when the number of rounds grows higher. The number of rounds is incremented every time the `depositRevenue()` function is called, causing the number of loop rounds to grow. When the number of rounds is high enough, the gas required will exceed the block gas limit, causing the transactions that call the `calculateTotalLuckyReward()` function to be reverted.

5.3.2 Remediation

Inspex suggests redesigning the function in consideration of the increasing number of rounds. For example, storing the accumulated total lucky reward for the previous rounds calculated instead of looping every round.

5.4 Unchecked Swapping Path For Reward Token

ID	IDX-004
Target	RevenueSharingPool
Category	Advanced Smart Contract Vulnerability
CWE	CWE-20: Improper Input Validation
Risk	**Severity:** Very Low **Impact:** Low Incorrectly specified path results in less amount of reward ($LUCKY) for the users. However, swapping tokens with incorrect path does not have a permanent impact on the platform since the platform can also swap tokens again with the correct path. **Likelihood:** Low It is unlikely that the whitelisted addresses will provide an incorrect swapping path.
Status	Resolved LuckyLion team has resolved this issue by changing the business flow. The reward will be sent to the users through the platform's backend, resulting in no swapping of tokens in the contract.

5.4.1 Description

In the `RevenueSharingPool` contract, the users can stake the LUCKY-BUSD LP token to acquire the reward as $LUCKY. These rewards will be transferred to the pool by a list of white-listed addresses through the `depositRevenue()` function. This function will swap all the tokens from the `inputTokens` parameter to $LUCKY.

RevenueSharingPool.sol

```
312 // for owner to deposit revenue (any tokens) to RevenueSharingPool contract
313 function depositRevenue(
314     InputToken[] calldata inputTokens,
315     address[] calldata BUSDToOutputPath,
316     uint256 minOutputAmount,
317     uint256 winRate,
318     uint256 TPV
319 ) external payable isWhitelisted(msg.sender) {
320     uint256 luckyRevenue;
321     uint256 roundId = getCurrentRoundId();
322
323     // specify a correct swap path for NATIVE-to-BUSD
324     address[] memory _path = new address[](2);
325         _path[0] = address(wNative);
326         _path[1] = address(BUSD);
```

```
327
328     // if owner deposit native coin contract will swap native to BUSD token
    first
329     if (msg.value > 0) {
330         wNative.deposit{value: msg.value}();
331         uint256 wNativeBalance = wNative.balanceOf(address(this));
332         _swapExactNativeForBUSD(wNativeBalance, _path);
333     }
334
335     if (inputTokens.length > 0 ) {
336         for (uint256 i; i < inputTokens.length; i++) {
337             if (inputTokens[i].token == address(lucky)) { // if token is LUCKY
    just transfer to contract directly
338                 IERC20(lucky).safeTransferFrom(msg.sender, address(this),
    inputTokens[i].amount);
339                 totalLuckyRevenue[roundId] += inputTokens[i].amount;
340                 luckyRevenue += inputTokens[i].amount;
341             } else if (inputTokens[i].token == address(BUSD)) { // if token is
    BUSD just transfer to contract first
342                 IERC20(BUSD).safeTransferFrom(msg.sender, address(this),
    inputTokens[i].amount);
343             } else {  // if token is not both LUCKY or BUSD let's swap it's to
    BUSD first
344                 _transferTokensToCave(inputTokens[i]);
345                 _swapTokensForBUSD(inputTokens[i]);
346             }
347         }
348     }
349
350     uint256 BUSDBalance = BUSD.balanceOf(address(this));
351     uint256 amountOut = _swapBUSDForToken( // swap BUSD to LUCKY token
352         BUSDBalance,
353         BUSDToOutputPath
354     );
355
356     require(
357         amountOut >= minOutputAmount,
358         "Expect amountOut to be greater than minOutputAmount."
359     );
360
361     updatePoolInfo(winRate, TPV, BUSDBalance, roundId); // update round pool
    info
362     totalLuckyRevenue[roundId] += amountOut;
363     luckyRevenue += amountOut;
364     START_ROUND_DATE = block.timestamp;
365     updateRoundId();
366     uint256 currentRoundId = getCurrentRoundId();
```

```
367     updateTotalStake(currentRoundId); // update new round total stake
368     emit DistributeLuckyRevenue(msg.sender, address(this), luckyRevenue);
369 }
```

The `inputTokens` parameter has to be specified on the router path (`tokenToBUSDPath`) for the function to swap correctly.

RevenueSharingPool.sol

```
62  struct InputToken {
63      address token;
64      uint256 amount;
65      address[] tokenToBUSDPath;
66  }
```

During swapping the `depositRevenue()` function, these internal functions, `_swapTokensForBUSD()` and `_swapBUSDForToken()` functions, are called without a path validation. This means if the `inputTokens.tokenToBUSDPath` parameter is specified incorrectly, it will affect the rewards for the users in the platform.

For the `_swapTokensForBUSD()` function, there is no $BUSD input validation. Since the `depositRevenue()` will convert any token in to $BUSD first and swap to $LUCKY, this means there will be no $BUSD to swap to $LUCKY, resulting in no $LUCKY in the reward pool.

RevenueSharingPool.sol

```
395 // swap any ERC20 / BEP20 tokens to BUSD token
396 function _swapTokensForBUSD(InputToken calldata inputTokens) private {
397     if (inputTokens.token != address(BUSD)) {
398         IERC20(inputTokens.token).approve(address(exchangeRouter), MAX_NUMBER);
399         exchangeRouter.swapExactTokensForTokens(
400             inputTokens.amount,
401             0, //minimum amount out can optimize by cal slippage
402             inputTokens.tokenToBUSDPath,
403             address(this),
404             block.timestamp + 60
405         );
406     }
407 }
```

For the `_swapTokensForBUSD()` function, there is no path validation whether the starting swapping token is $BUSD and the wanted token is $LUCKY, resulting in incorrectly swapping token result if the path is incorrect.

RevenueSharingPool.sol

```
409 // swap BUSD token to LUCKY token
410 function _swapBUSDForToken(uint256 amount, address[] memory path)
```

```
411     private
412     returns (uint256)
413 {
414     if (amount == 0 || path[path.length - 1] == address(BUSD)) {
415         return amount;
416     }
417     BUSD.approve(address(exchangeRouter), MAX_NUMBER);
418     uint256[] memory amountOuts = exchangeRouter.swapExactTokensForTokens(
419         amount,
420         0,//minimum amount out can optimize by cal slippage
421         path,
422         address(this),
423         block.timestamp + 60
424     );
425     return amountOuts[amountOuts.length - 1];
426 }
```

RevenueSharingPool.sol

```
395 // swap any ERC20 / BEP20 tokens to BUSD token
396 function _swapTokensForBUSD(InputToken calldata inputTokens) private {
397     if (inputTokens.token != address(BUSD)) {
398         IERC20(inputTokens.token).approve(address(exchangeRouter), MAX_NUMBER);
399         exchangeRouter.swapExactTokensForTokens(
400             inputTokens.amount,
401             0, //minimum amount out can optimize by cal slippage
402             inputTokens.tokenToBUSDPath,
403             address(this),
404             block.timestamp + 60
405         );
406     }
407 }
```

5.4.2 Remediation

Inspex suggests implementing a path validation in `_swapBUSDForToken()` and `_swapTokensForBUSD()` functions to ensure that the token is swapped correctly, for example:

RevenueSharingPool.sol

```
409 // swap BUSD token to LUCKY token
410 function _swapBUSDForToken(uint256 amount, address[] memory path)
411     private
412     returns (uint256)
413 {
414     if (amount == 0 || path[path.length - 1] == address(BUSD)) {
415         return amount;
416     }
```

```
417     require (path[0] == address(BUSD), 'tokenInput is not BUSD');
418     BUSD.approve(address(exchangeRouter), MAX_NUMBER);
419     uint256[] memory amountOuts = exchangeRouter.swapExactTokensForTokens(
420         amount,
421         0,//minimum amount out can optimize by cal slippage
422         path,
423         address(this),
424         block.timestamp + 60
425     );
426     return amountOuts[amountOuts.length - 1];
427 }
```

RevenueSharingPool.sol

```
395 // swap any ERC20 / BEP20 tokens to BUSD token
396 function _swapTokensForBUSD(InputToken calldata inputTokens) private {
397     if (inputTokens.token != address(BUSD) &&
    inputTokens.tokenToBUSDPath[inputTokens.tokenToBUSDPath.length - 1] ==
    address(BUSD)) {
398         IERC20(inputTokens.token).approve(address(exchangeRouter), MAX_NUMBER);
399         exchangeRouter.swapExactTokensForTokens(
400             inputTokens.amount,
401             0, //minimum amount out can optimize by cal slippage
402             inputTokens.tokenToBUSDPath,
403             address(this),
404             block.timestamp + 60
405         );
406     }
407 }
```

5.5 Insufficient Logging for Privileged Functions

ID	IDX-005
Target	RevenueSharingPool
Category	General Smart Contract Vulnerability
CWE	CWE-778: Insufficient Logging
Risk	**Severity:** Very Low **Impact:** Low Privileged functions' executions cannot be monitored easily by the users. **Likelihood:** Low It is not likely that the execution of the privileged functions will be a malicious action.
Status	Resolved LuckyLion team has resolved this issue as suggested in commit `1bc56dc598b7a9887dbed4ea21bd57cca40c9d3a` by emitting necessary events, allowing the community to monitor them easily.

5.5.1 Description

Privileged functions that are executable by the controlling parties are not logged properly by emitting events. Without events, it is not easy for the public to monitor the execution of those privileged functions, allowing the controlling parties to perform actions that cause big impacts on the platform.

For example, the owner can set the reward distribution period (`MAX_DATE`) by executing the `updateMaxDate()` function in the `RevenueSharingPool` contract, and no event is emitted.

The privileged functions without sufficient logging are as follows:

Target	Function
RevenueSharingPool.sol (L: 157)	updateMaxDate()
RevenueSharingPool.sol (L: 161)	addWhitelist()
RevenueSharingPool.sol (L: 165)	removeWhitelist()

5.5.2 Remediation

Inspex suggests emitting events for the execution of the privileged functions, for example:

RevenueSharingPool.sol

```
156 // Update max number of day in a round (default 7 days)
```

```
157 event UpdateMaxDate(uint256 newMaxDate);
158 function updateMaxDate(uint256 newMaxDate) external onlyOwner {
159     MAX_DATE = newMaxDate;
160     emit UpdateMaxDate(newMaxDate);
161 }
```

5.6 Unused Function Parameter

ID	IDX-006
Target	Shield
Category	Smart Contract Best Practice
CWE	CWE-1104: Use of Unmaintained Third Party Components
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	No Security Impact LuckyLion team has acknowledged this issue since the current unused parameters could be utilized in the future, and leaving those parameters as is right now has no direct impact.

5.6.1 Description

There are parameters defined in multiple functions of the smart contracts that are not used anywhere, causing unnecessary gas usage.

In the `Shield` contract, the `_harvestIntervalInMinutes` and `_farmStartIntervalInMinutes` parameters in `add()` and `set()` functions are declared but not used anywhere in the functions.

MasterchefShield.sol

```
14  function add(uint256 _allocPoint, IERC20 _lpToken, uint256
    _harvestIntervalInMinutes,uint256 _farmStartIntervalInMinutes) external
    onlyOwner {
15      masterchef.add(_allocPoint, _lpToken, 0, 0);
16  }
17
18  function set(uint256 _pid, uint256 _allocPoint, uint256
    _harvestIntervalInMinutes,uint256 _farmStartIntervalInMinutes) external
    onlyOwner {
19      masterchef.set(_pid, _allocPoint, 0, 0);
20  }
```

5.6.2 Remediation

Inspex suggests removing the unused parameter to reduce unnecessary gas usage.

5.7 Improper Function Visibility

ID	IDX-007
Target	RevenueSharingPool
Category	Smart Contract Best Practice
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved LuckyLion team has resolved this issue as suggested in commit `1bc56dc598b7a9887dbed4ea21bd57cca40c9d3a` by changing the function's visibility to optimize the gas usage.

5.7.1 Description

Functions with public visibility copy calldata to memory when being executed, while external functions can read directly from calldata. Memory allocation uses more resources (gas) than reading directly from calldata.

The following source code shows that the `getStakeAmount()` function in the RevenueSharingPool contract is set to public visibility and it is never called from any internal function.

RevenueSharingPool.sol

```
254 // return user stake amount of specific round and date
255 function getStakeAmount(uint256 roundId, uint256 day) public view returns
    (uint256) {
256     return stakeAmount[roundId][day][msg.sender];
257 }
```

5.7.2 Remediation

Inspex suggests changing the `getStakeAmount()` functions' visibility to external if they are not called from any internal function as shown in the following example:

RevenueSharingPool.sol

```
254 // return user stake amount of specific round and date
255 function getStakeAmount(uint256 roundId, uint256 day) external view returns
    (uint256) {
256     return stakeAmount[roundId][day][msg.sender];
257 }
```

5.8 Inexplicit State Variable Visibility

ID	IDX-008
Target	RevenueSharingPool
Category	Smart Contract Best Practice
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved LuckyLion team has resolved this issue as suggested in commit `1bc56dc598b7a9887dbed4ea21bd57cca40c9d3a` by changing the state variable's visibility in accordance with the business design.

5.8.1 Description

A state variable with an inexplicit visibility is set to `internal` by default. The variables with `internal` visibility are not directly accessible through the getter function. This causes inconveniences for the platform users on the retrieval of the data if that state is designed to be publicly accessible.

The following source code shows that the visibility of the `stakeAmount` state variable in the `RevenueSharingPool` contract is not set, causing its visibility to be set as `internal` by default.

RevenueSharingPool.sol

```
33  mapping(uint256 => mapping(uint256 => mapping(address => uint256)))
    stakeAmount;
```

5.8.2 Remediation

Inspex suggests explicitly defining the `stakeAmount` state variable's visibility to match the business design of the platform, for example if the intended visibility is `public`, the visibility should be defined as follows:

RevenueSharingPool.sol

```
33  mapping(uint256 => mapping(uint256 => mapping(address => uint256))) public
    stakeAmount;
```

5.9 Inexplicit Solidity Compiler Version

ID	IDX-009
Target	RevenueSharingPool
Category	Smart Contract Best Practice
CWE	CWE-1104: Use of Unmaintained Third Party Components
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved LuckyLion team has resolved this issue as suggested in commit b4da56051b6b6e17e2b9ce84deae9dc2f3514454 by changing the solidity compiler to the latest version explicitly.

5.9.1 Description

The Solidity compiler versions declared in the smart contracts were not explicit. Each compilation may be done using different compiler versions, which may potentially result in compatibility issues.

RevenueSharingPool.sol

```
1 // SPDX-License-Identifier: UNLICENSED
2 pragma solidity ^0.8.0;
```

5.9.2 Recommendation

Inspex suggests fixing the solidity compiler to the latest stable version. At the time of the audit, the latest stable version of Solidity compiler in major 0.8 is v0.8.10 [2].

RevenueSharingPool.sol

```
1 // SPDX-License-Identifier: UNLICENSED
2 pragma solidity 0.8.10;
```

6. Appendix

6.1. About Inspex

Inspex is formed by a team of cybersecurity experts highly experienced in various fields of cybersecurity. We provide blockchain and smart contract professional services at the highest quality to enhance the security of our clients and the overall blockchain ecosystem.

Follow Us On:

Website	https://inspex.co
Twitter	@InspexCo
Facebook	https://www.facebook.com/InspexCo
Telegram	@inspex_announcement

6.2. References

[1] "OWASP Risk Rating Methodology." [Online]. Available: https://owasp.org/www-community/OWASP_Risk_Rating_Methodology. [Accessed: 08-May-2021]

[2] ethereum, "Releases · ethereum/solidity." [Online]. Available: https://github.com/ethereum/solidity/releases. [Accessed: 24-November-2021]

inspeX

CYBERSECURITY PROFESSIONAL SERVICE