Farm

Smart Contract Audit Report
Prepared for LuckyLion

Date Issued:	Sep 21, 2021
Project ID:	AUDIT2021024
Version:	v1.0
Confidentiality Level:	Public

Report Information

Project ID	AUDIT2021024
Version	v1.0
Client	LuckyLion
Project	Farm
Auditor(s)	Suvicha Buakhom Peeraphut Punsuwan
Author	Suvicha Buakhom
Reviewer	Weerawat Pawanawiwat
Confidentiality Level	Public

Version History

Version	Date	Description	Author(s)
1.0	Sep 21, 2021	Full report	Suvicha Buakhom

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 Farm smart contracts between Sep 14, 2021 and Sep 15, 2021. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of Farm 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.

LuckyLion Farm smart contract has the migration mechanism implemented with the ability to transfer the LP token to any address. Furthermore, the ownership of the $LUCKY can also be transferred, granting the minting privilege to another address. These mechanisms have high impacts on the users; however, 2 days minimum timelock is used by the LuckyLion team to delay the execution of these privileged functions. Inspex recommends the platform users to closely monitor the timelock contract for the execution of these functions.

1.1. Audit Result

In the initial audit, Inspex found 1 high, 3 medium, 1 low, 1 very low, and 2 info-severity issues. With the project team's prompt response, 1 high, 3 medium, 1 very low, and 2 info-severity issues were resolved or mitigated in the reassessment, while 1 low-severity issue was acknowledged by the team. Therefore, Inspex trusts that Farm 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

Lucky Lion 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.

Farm is the main feature responsible for distributing $LUCKY on the platform. The users can deposit tokens to the pools added in the farm and earn $LUCKY as a reward.

Scope Information:

Project Name	Farm
Website	https://app.luckylion.io/farm
Smart Contract Type	Ethereum Smart Contract
Chain	Binance Smart Chain
Programming Language	Solidity

Audit Information:

Audit Method	Whitebox
Audit Date	Sep 14, 2021 - Sep 15, 2021
Reassessment Date	Sep 18, 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:

Initial Audit: (Commit: db276805128df07538bbd8ee0ec837584194901a)

Contract	Location (URL)
MasterChef	https://github.com/LuckyLionIO/Lucky-farm/blob/db27680512/contracts/Masterchef.sol
SyrupBar	https://github.com/LuckyLionIO/Lucky-farm/blob/db27680512/contracts/SyrupBar.sol

Reassessment: (Commit: 5aa5780d15ce4b471d49abb3cba09ac7203975f2)

Contract	Location (URL)
MasterChef	https://github.com/LuckyLionIO/Lucky-farm/blob/5aa5780d15/contracts/Masterchef.sol
SyrupBar	https://github.com/LuckyLionIO/Lucky-farm/blob/5aa5780d15/contracts/SyrupBar.sol

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

The `transferLuckyOwnership()` function has been added in the reassessment commit. This function is out of the audit scope but may cause risks to users that the owner can change the owner of the $LUCKY and manually mint an arbitrary number of $LUCKY.

We have notified the LuckyLion team of our concern, and the team has decided to leave this function as is and clarified that this function is designed for migrating the minter to a new `MasterChef` contract if there is any major problem with the contract in the future, so it is very unlikely that this function will be used.

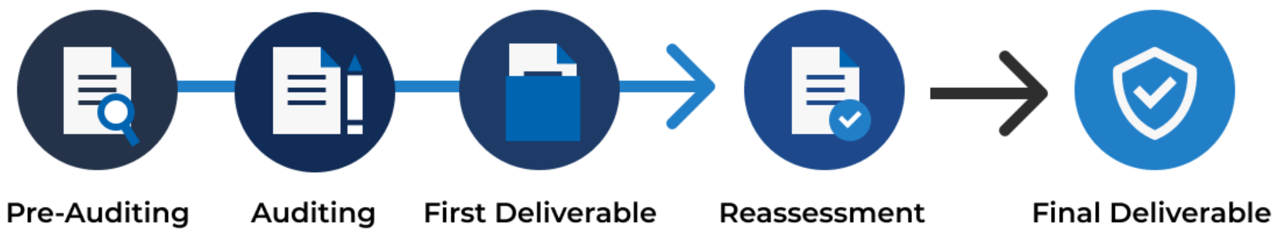
Moreover, the `transferLuckyOwnership()` function can only be called by the contract owner, and the owner of the `MasterChef` contract is the `Timelock` contract with 2 days minimum delay. Therefore, Inspex trusts that this risk should be known to the users.

Finally, we strongly recommend that the platform users should monitor the execution of functions in the timelock and act accordingly.

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
Advanced
Business Logic Flaw
Ownership Takeover
Broken Access Control
Broken Authentication
Use of Upgradable Contract Design
Insufficient Logging for Privileged Functions
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

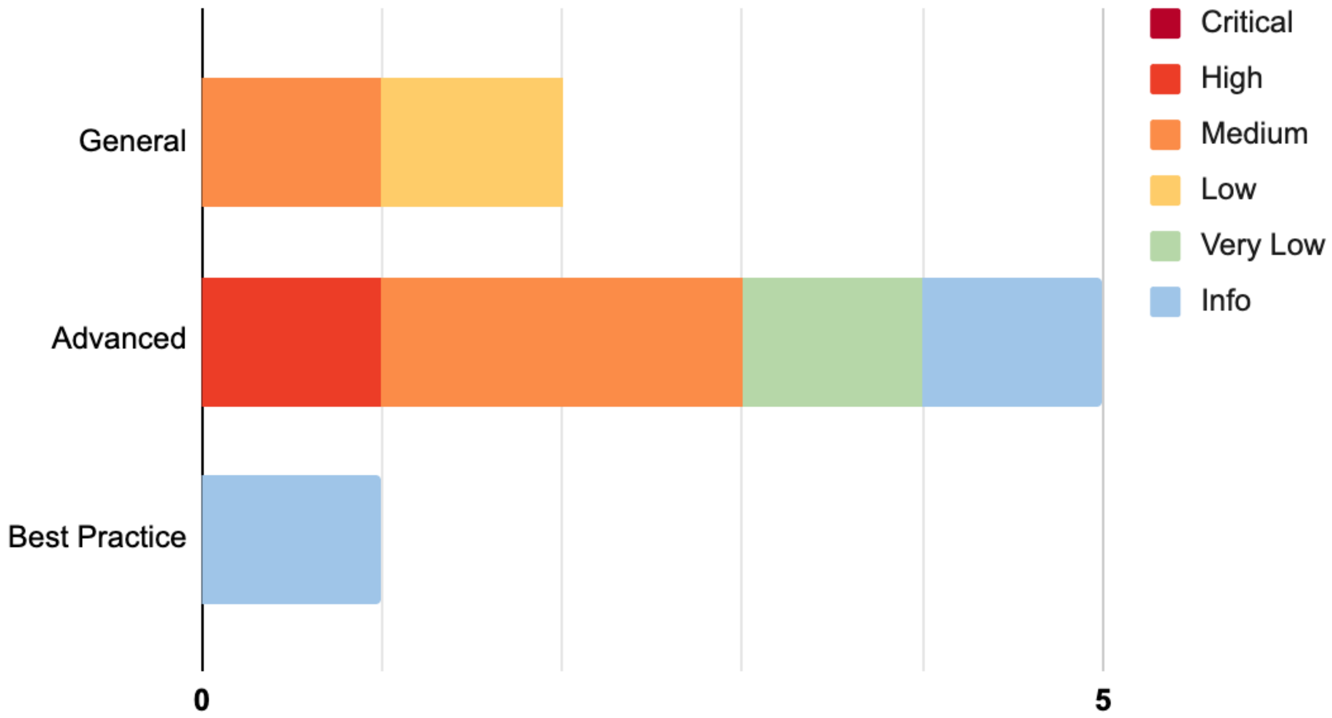
Both likelihood and impact can be categorized into three levels: **Low**, **Medium**, and **High**.

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**.

Likelihood / Impact	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 8 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	Token Draining Using migrate() Function	Advanced	High	Resolved *
IDX-002	Improper Reward Calculation (Duplicated LP Token)	Advanced	Medium	Resolved
IDX-003	Improper Reward Calculation (withUpdate)	Advanced	Medium	Resolved
IDX-004	Centralized Control of State Variable	General	Medium	Resolved
IDX-005	Design Flaw in massUpdatePools() Function	General	Low	Acknowledged
IDX-006	Insufficient Logging for Privileged Functions	Advanced	Very Low	Resolved
IDX-007	Unsupported Design for Deflationary Token	Advanced	Info	Resolved
IDX-008	Improper Function Visibility	Best Practice	Info	Resolved

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

5. Detailed Findings Information

5.1. Token Draining Using migrate() Function

ID	IDX-001
Target	MasterChef
Category	Advanced Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity: High** **Impact: High** The owner of the `MasterChef` contract can steal all `lpToken` from the contract. **Likelihood: Medium** Only the contract owner can set the `migrator` address; however, there is no restriction to prevent the owner from performing this attack.
Status	Resolved * LuckyLion team has decided to keep this functionality and mitigated this issue by implementing a timelock mechanism. The `MasterChef` contract is owned by the `Timelock` contract with 7 days delay and 2 days minimum delay. `Timelock` contract with 2 days minimum delay: https://bscscan.com/address/0x4b6c8959a41475347226d51f37ec9a1e09f39a92#code `MasterChef` contract: https://bscscan.com/address/0xb6fe67c8a28d50c50f65fdb5847ee4477c550568#code Ownership transfer of `MasterChef` to `Timelock` contract: https://bscscan.com/tx/0xb54a48f780f6912f283b0113dfbb9fbef4d0f9e421bc532bb9c41a43cc15140f#eventlog Platform users should monitor the execution of functions in the timelock and act accordingly.

5.1.1. Description

In the `MasterChef` contract, the `setMigrator()` function can be used by the contract owner to set the `migrator` address.

Masterchef.sol

```
function setMigrator(IMigratorChef _migrator) public onlyOwner {
    migrator = _migrator;
}
```

The `migrate()` function can be called by anyone. When the `migrate()` function is called, the `MasterChef` contract will allow the `migrator` to spend all `lpToken` balance in the contract.

Masterchef.sol

```
function migrate(uint256 _pid) public {
    require(address(migrator) != address(0), "migrate: no migrator");
    PoolInfo storage pool = poolInfo[_pid];
    IERC20 lpToken = pool.lpToken;
    uint256 bal = lpToken.balanceOf(address(this));
    lpToken.safeApprove(address(migrator), bal);
    IERC20 newLpToken = migrator.migrate(lpToken);
    require(bal == newLpToken.balanceOf(address(this)), "migrate: bad");
    pool.lpToken = newLpToken;
}
```

The contract owner can steal all `lpToken` in the contract by setting the `migrator` address to a malicious address and use `transferFrom()` function to transfer all `lpToken` from `MasterChef` to any address.

5.1.2. Remediation

Inspex suggests removing the migration mechanism from the `MasterChef` contract.

However, if the migration is needed, Inspex suggests mitigating this issue by implementing a timelock mechanism with a sufficient length of time to delay the changes. This allows the platform users to monitor the timelock and be notified of the potential changes being done on the smart contracts.

5.2. Improper Reward Calculation (Duplicated LP Token)

ID	IDX-002
Target	MasterChef
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** Medium The $LUCKY reward miscalculation can lead to an unfair $LUCKY token distribution to the users. **Likelihood:** Medium It is possible that the contract owner will add or migrate a new pool that uses the same token as another pool since there is no restriction.
Status	Resolved LuckyLion team has resolved this issue as suggested in commit `5aa5780d15ce4b471d49abb3cba09ac7203975f2`.

5.2.1. Description

In the `MasterChef` contract, a new staking pool can be added using the `add()` function. The staking token for the new pool is defined using the `_lpToken` variable; however, there is no additional checking whether the `_lpToken` is already used in other pools or not.

Masterchef.sol

```
function add(uint256 _allocPoint, IERC20 _lpToken, uint256
_harvestIntervalInMinutes,uint256 _farmStartIntervalInMinutes, bool
_withUpdate) public onlyOwner {
    uint256 _harvestTimestampInUnix = block.timestamp +
(_harvestIntervalInMinutes *60); //*60 to convert from minutes to second.
    uint256 _farmStartTimestampInUnix = block.timestamp +
(_farmStartIntervalInMinutes *60);
    if (_withUpdate) {
        massUpdatePools();
    }
    uint256 lastRewardBlock = block.number > startBlock ? block.number :
startBlock;
    totalAllocPoint = totalAllocPoint.add(_allocPoint);
    poolInfo.push(PoolInfo({
        lpToken: _lpToken,
        allocPoint: _allocPoint,
        lastRewardBlock: lastRewardBlock,
```

```
        accLuckyPerShare: 0,
        harvestTimestamp: _harvestTimestampInUnix,
        farmStartDate : _farmStartTimestampInUnix
    }));
    emit
PoolAdded(_lpToken,_allocPoint,_harvestTimestampInUnix,_farmStartTimestampInUni
x);
}
```

There is also the `migrate()` function that can change `lpToken` without checking if they are duplicates.

Masterchef.sol

```
function migrate(uint256 _pid) public {
    require(address(migrator) != address(0), "migrate: no migrator");
    PoolInfo storage pool = poolInfo[_pid];
    IERC20 lpToken = pool.lpToken;
    uint256 bal = lpToken.balanceOf(address(this));
    lpToken.safeApprove(address(migrator), bal);
    IERC20 newLpToken = migrator.migrate(lpToken);
    require(bal == newLpToken.balanceOf(address(this)), "migrate: bad");
    pool.lpToken = newLpToken;
}
```

In the `updatePool()` function, the balance of `pool.lpToken` in the contract is used as a denominator to calculate `pool.accLuckyPerShare`.

Masterchef.sol

```
function updatePool(uint256 _pid) public {
    PoolInfo storage pool = poolInfo[_pid];
    if (block.number <= pool.lastRewardBlock) {
        return;
    }
    uint256 lpSupply = pool.lpToken.balanceOf(address(this));
    if (lpSupply == 0 || pool.allocPoint == 0) {
        pool.lastRewardBlock = block.number;
        return;
    }
    uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
    uint256 luckyReward =
multiplier.mul(luckyPerBlock).mul(pool.allocPoint).div(totalAllocPoint);
    //new one
    // check at final to mint exact lucky to complete the round 9 million and
100 millions totalsupply
    uint256 luckyRewardForDev = luckyReward.mul(devMintingRatio).div(10000);
    //logic to prevent the minting exceeds the capped totalsupply
    //1st case, reward for dev will exceed Lucky's totalSupply so we limit the
```

```
minting amount to syrup.
    if (luckyRewardForDev.add(lucky.totalSupply()) > lucky.cap() ) {
        uint256 remainingReward = lucky.cap().sub(lucky.totalSupply());
        //in case that remainingReward > capped reward for dev.
        if (remainingReward.add(accumulatedRewardForDev) > capRewardForDev) {
            uint256 lastRemainingRewardForDev =
capRewardForDev.sub(accumulatedRewardForDev);
            lucky.mint(devAddress,lastRemainingRewardForDev);
            accumulatedRewardForDev =
accumulatedRewardForDev.add(lastRemainingRewardForDev);
            //the rest is minted to users.
            lucky.mint(address(syrup),lucky.cap().sub(lucky.totalSupply()));
        }
        //normal case that dev's caped reward has not been reached yet, but the
totalSupply of Lucky is reached.
        else {
            lucky.mint(devAddress, remainingReward);
            //track the token that is minted to dev.
            accumulatedRewardForDev =
accumulatedRewardForDev.add(remainingReward);
        }

    }
    //supply cap was not reached and capRewardForDevev still has room to mint
for.
    else {
        //capRewardForDev is reached.
        if (luckyRewardForDev.add(accumulatedRewardForDev) > capRewardForDev) {
            uint256 lastRemainingRewardForDev =
capRewardForDev.sub(accumulatedRewardForDev);
            lucky.mint(devAddress,lastRemainingRewardForDev);
            //track the token that is minted to dev.
            accumulatedRewardForDev =
accumulatedRewardForDev.add(lastRemainingRewardForDev);

            //mint the left portion of dev to the pools.
            lucky.mint(address(syrup),luckyRewardForDev
.sub(lastRemainingRewardForDev));

            if (luckyReward.add(lucky.totalSupply()) > lucky.cap() ){
                lucky.mint(address(syrup),lucky.cap()
.sub(lucky.totalSupply()));
            }
            else {
                lucky.mint(address(syrup),luckyReward);
            }
        }
```

```
        else {

            lucky.mint(devAddress,luckyRewardForDev);
            accumulatedRewardForDev =
accumulatedRewardForDev.add(luckyRewardForDev);

            if (luckyReward.add(lucky.totalSupply()) > lucky.cap() ){
                lucky.mint(address(syrup),lucky.cap()
.sub(lucky.totalSupply()));
            }
            else{
                lucky.mint(address(syrup),luckyReward);
            }

        }
    }
    pool.accLuckyPerShare =
pool.accLuckyPerShare.add(luckyReward.mul(1e12).div(lpSupply));
    pool.lastRewardBlock = block.number;
}
```

When the owner of `MasterChef` adds a pool with the same `lpToken` as another pool, the `lpToken` value is counted from all pools using the same `lpToken`, resulting in a higher value of denominator (`lpSupply`) than it should be.

5.2.2. Remediation

Inspex suggests validating the `_lpToken` address in `add()` and `migrate()` functions to prevent duplicated `_lpToken` when adding a new pool as shown in the following example:

Masterchef.sol

```
mapping(address => bool) public isAddedPool;
```

Masterchef.sol

```
function add(uint256 _allocPoint, IERC20 _lpToken, uint256
_harvestIntervalInMinutes,uint256 _farmStartIntervalInMinutes, bool
_withUpdate) public onlyOwner {
    require(!isAddedPool[address(_lpToken)], "add: Duplicated LP Token");
    uint256 _harvestTimestampInUnix = block.timestamp +
(_harvestIntervalInMinutes *60); //*60 to convert from minutes to second.
    uint256 _farmStartTimestampInUnix = block.timestamp +
(_farmStartIntervalInMinutes *60);
    if (_withUpdate) {
        massUpdatePools();
    }
```

```
    uint256 lastRewardBlock = block.number > startBlock ? block.number :
startBlock;
    totalAllocPoint = totalAllocPoint.add(_allocPoint);
    poolInfo.push(PoolInfo({
        lpToken: _lpToken,
        allocPoint: _allocPoint,
        lastRewardBlock: lastRewardBlock,
        accLuckyPerShare: 0,
        harvestTimestamp: _harvestTimestampInUnix,
        farmStartDate : _farmStartTimestampInUnix
    }));
    emit PoolAdded(_lpToken,_allocPoint,_harvestTimestampInUnix,
_farmStartTimestampInUnix);
    isAddedPool[address(_lpToken)] = true;
}
```

Masterchef.sol

```
function migrate(uint256 _pid) public {
    require(address(migrator) != address(0), "migrate: no migrator");
    PoolInfo storage pool = poolInfo[_pid];
    IERC20 lpToken = pool.lpToken;
    uint256 bal = lpToken.balanceOf(address(this));
    lpToken.safeApprove(address(migrator), bal);
    IERC20 newLpToken = migrator.migrate(lpToken);

    require(!isAddedPool[address(newLpToken)], "migrate: Duplicated LP Token");
    require(bal == newLpToken.balanceOf(address(this)), "migrate: bad");

    isAddedPool[address(pool.lpToken)] = false;
    pool.lpToken = newLpToken;
    isAddedPool[address(pool.lpToken)] = true;
}
```

5.3. Improper Reward Calculation (_withUpdate)

ID	IDX-003
Target	MasterChef
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** Medium The $LUCKY reward miscalculation can lead to an unfair $LUCKY token distribution to the users. **Likelihood:** Medium The `add()` and the `set()` functions can only be called by the contract owner, but it is possible that the `totalAllocPoint` state will be changed without setting the `_withUpdate` parameter to true .
Status	Resolved LuckyLion team has resolved this issue as suggested in commit `5aa5780d15ce4b471d49abb3cba09ac7203975f2`.

5.3.1. Description

The `totalAllocPoint` variable is used to determine the portion that each pool would get from the total reward minted, so it is one of the main factors used in the rewards calculation. Therefore, whenever the `totalAllocPoint` variable is modified without updating the pending reward first, the reward of each pool will be incorrectly calculated.

In the `add()` and `set()` functions shown below, if `_withUpdate` is set to false, the `totalAllocPoint` variable will be modified without updating the rewards (`massUpdatePools()`).

Masterchef.sol

```
function add(uint256 _allocPoint, IERC20 _lpToken, uint256
_harvestIntervalInMinutes,uint256 _farmStartIntervalInMinutes, bool
_withUpdate) public onlyOwner {
    uint256 _harvestTimestampInUnix = block.timestamp +
(_harvestIntervalInMinutes *60); //*60 to convert from minutes to second.
    uint256 _farmStartTimestampInUnix = block.timestamp +
(_farmStartIntervalInMinutes *60);
    if (_withUpdate) {
        massUpdatePools();
    }
    uint256 lastRewardBlock = block.number > startBlock ? block.number :
startBlock;
```

```
    totalAllocPoint = totalAllocPoint.add(_allocPoint);
    poolInfo.push(PoolInfo({
        lpToken: _lpToken,
        allocPoint: _allocPoint,
        lastRewardBlock: lastRewardBlock,
        accLuckyPerShare: 0,
        harvestTimestamp: _harvestTimestampInUnix,
        farmStartDate : _farmStartTimestampInUnix
    }));
    emit PoolAdded(_lpToken,_allocPoint,_harvestTimestampInUnix, _farmStartTimestampInUnix);
}

// Update the given pool's lucky allocation point. Can only be called by the owner.
function set(uint256 _pid, uint256 _allocPoint, uint256 _harvestIntervalInMinutes,uint256 _farmStartIntervalInMinutes, bool _withUpdate) public onlyOwner {
    uint256 _harvestTimestampInUnix = block.timestamp + (_harvestIntervalInMinutes *60); //*60 to convert from minutes to second.
    uint256 _farmStartTimestampInUnix = block.timestamp + (_farmStartIntervalInMinutes *60);
    if (_withUpdate) {
        massUpdatePools();
    }
    totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint);
    poolInfo[_pid].allocPoint = _allocPoint;
    poolInfo[_pid].harvestTimestamp = _harvestTimestampInUnix;
    poolInfo[_pid].farmStartDate = _farmStartTimestampInUnix;
    emit PoolSet(_pid,_allocPoint,_harvestTimestampInUnix, _farmStartTimestampInUnix);
}
```

For example:

Assuming that on block 1000000, `luckyPerBlock` is 5 $LUCKY per block, `totalAllocPoint` is 5000, and `allocPoint` of pool id 0 is 500.

Block	Action
1000000	All pools' rewards are updated
1100000	A new pool is added using the `add()` function, causing the `totalAllocPoint` to be changed from 5000 to 10000
1200000	The pools' rewards are updated once again.

From current logic, the total rewards allocated to the pool id 0 during block 1000000 to 1200000 is equal to 50,000 $LUCKY, calculated using the following equation:

Block	Total Reward Block	Total Allocation Point	Total $LUCKY per block for pool 0 (luckyPerBlock*pool0allocPoint/ totalAllocPoint)	Total pool 0 $LUCKY Reward
1000000 - 1200000	200000	10,000	0.25 $LUCKY per block	50,000 $LUCKY

However, the rewards should be calculated by accounting for the original `totalAllocPoint` value during the period when it is not yet updated as follows:

Block	Total Reward Block	Total Allocation Point	Total $LUCKY per block for pool 0 (luckyPerBlock*pool0allocPoint/ totalAllocPoint)	Total pool 0 $LUCKY Reward
1000000 - 1100000	100000	5,000	0.5 $LUCKY per block	50,000 $LUCKY
1100000 - 1200000	100000	10,000	0.25 $LUCKY per block	25,000 $LUCKY

The correct total $LUCKY reward is 75,000 $LUCKY, which is different from the miscalculated reward by 25,000 $LUCKY.

5.3.2. Remediation

Inspex suggests removing the `_withUpdate` variable in the `add()` and `set()` functions and always calling the `massUpdatePools()` function before updating `totalAllocPoint` variable as shown in the following example:

Masterchef.sol

```
function add(uint256 _allocPoint, IERC20 _lpToken, uint256 _harvestIntervalInMinutes,uint256 _farmStartIntervalInMinutes, bool _withUpdate) public onlyOwner {
    uint256 _harvestTimestampInUnix = block.timestamp + (_harvestIntervalInMinutes *60); //*60 to convert from minutes to second.
    uint256 _farmStartTimestampInUnix = block.timestamp + (_farmStartIntervalInMinutes *60);
    massUpdatePools();
    uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
    totalAllocPoint = totalAllocPoint.add(_allocPoint);
    poolInfo.push(PoolInfo({
        lpToken: _lpToken,
        allocPoint: _allocPoint,
        lastRewardBlock: lastRewardBlock,
        accLuckyPerShare: 0,
```

```
        harvestTimestamp: _harvestTimestampInUnix,
        farmStartDate : _farmStartTimestampInUnix
    }));
    emit PoolAdded(_lpToken,_allocPoint,_harvestTimestampInUnix,
_farmStartTimestampInUnix);
}

// Update the given pool's lucky allocation point. Can only be called by the
owner.
function set(uint256 _pid, uint256 _allocPoint, uint256
_harvestIntervalInMinutes,uint256 _farmStartIntervalInMinutes, bool
_withUpdate) public onlyOwner {
    uint256 _harvestTimestampInUnix = block.timestamp +
(_harvestIntervalInMinutes *60); //*60 to convert from minutes to second.
    uint256 _farmStartTimestampInUnix = block.timestamp +
(_farmStartIntervalInMinutes *60);
    massUpdatePools();
    totalAllocPoint =
totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint);
    poolInfo[_pid].allocPoint = _allocPoint;
    poolInfo[_pid].harvestTimestamp = _harvestTimestampInUnix;
    poolInfo[_pid].farmStartDate = _farmStartTimestampInUnix;
    emit PoolSet(_pid,_allocPoint,_harvestTimestampInUnix,
_farmStartTimestampInUnix);
}
```

5.4. Centralized Control of State Variable

ID	IDX-004
Target	MasterChef
Category	General Smart Contract Vulnerability
CWE	CWE-710: Improper Adherence to Coding Standard
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 These functions can only be called by the contract owner; however, there is nothing to restrict the changes from being done by the owner.
Status	Resolved LuckyLion team has resolved this issue by implementing a timelock mechanism. The `MasterChef` contract is owned by the `Timelock` contract with 7 days delay and 2 days minimum delay. `Timelock` contract with 2 days minimum delay: https://bscscan.com/address/0x4b6c8959a41475347226d51f37ec9a1e09f39a92#code `MasterChef` contract: https://bscscan.com/address/0xb6fe67c8a28d50c50f65fdb5847ee4477c550568#code Ownership transfer of `MasterChef` to `Timelock` contract: https://bscscan.com/tx/0xb54a48f780f6912f283b0113dfbb9fbef4d0f9e421bc532bb9c41a43cc15140f#eventlog Platform users should monitor the execution of functions in the timelock and act accordingly.

5.4.1. 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
Masterchef.sol (L:128)	MasterChef	add()	onlyOwner
Masterchef.sol (L:148)	MasterChef	set()	onlyOwner
Masterchef.sol (L:196)	MasterChef	setMigrator()	onlyOwner
Masterchef.sol (L:363)	MasterChef	setDevAddress()	onlyOwner
Masterchef.sol (L:369)	MasterChef	updateLuckyPerBlock()	onlyOwner
Ownable.sol (L:53)	MasterChef	renounceOwnership()	onlyOwner
Ownable.sol (L:61)	MasterChef	transferOwnership()	onlyOwner

Please note that the `Ownable` contract is inherited from the OpenZeppelin's library.

5.4.2. 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

5.5. Design Flaw in massUpdatePools() Function

ID	IDX-005
Target	MasterChef
Category	General Smart Contract Vulnerability
CWE	CWE-400: Uncontrolled Resource Consumption
Risk	**Severity:** Low **Impact:** Medium The `massUpdatePools()` function will eventually be unusable due to excessive gas usage. **Likelihood:** Low It is very unlikely that the `poolInfo` size will be raised until the `massUpdatePools()` function is unusable.
Status	**Acknowledged** LuckyLion team has acknowledged this issue. The team has prepared a testing process on the local network (environment with the same settings as the main network) which has the same number of pools as the mainnet, so this problem can be proactively prevented.

5.5.1. Description

The `massUpdatePools()` function executes the `updatePool()` function, which is a state modifying function for all added pools as shown below:

Masterchef.sol

```
function massUpdatePools() public {
    uint256 length = poolInfo.length;
    for (uint256 pid = 0; pid < length; ++pid) {
        updatePool(pid);
    }
}
```

With the current design, the added pools cannot be removed. They can only be disabled by setting the `pool.allocPoint` to 0. Even if a pool is disabled, the `updatePool()` function for this pool is still called. Therefore, if new pools continue to be added to this contract, the `poolInfo.length` will continue to grow and this function will eventually be unusable due to excessive gas usage.

5.5.2. Remediation

Inspex suggests making the contract capable of removing unnecessary or ended pools to reduce the loop round in the `massUpdatePools()` function.

5.6. Insufficient Logging for Privileged Functions

ID	IDX-006
Target	MasterChef
Category	Advanced 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 `5aa5780d15ce4b471d49abb3cba09ac7203975f2`.

5.6.1. 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 modify the `migrator` address by executing `setMigrator()` function in the `MasterChef` contract, and no event is emitted.

Masterchef.sol

```
363 function setMigrator(IMigratorChef _migrator) public onlyOwner {
364     migrator = _migrator;
365 }
```

The privileged functions without sufficient logging are as follows:

File	Contract	Function	Modifier
Masterchef.sol (L:196)	MasterChef	setMigrator()	onlyOwner
Masterchef.sol (L:363)	MasterChef	setDevAddress()	onlyOwner

5.6.2. Remediation

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

Masterchef.sol

```
event SetMigrator(IMigratorChef _oldmigrator, IMigratorChef _migrator);
function setMigrator(IMigratorChef _migrator) public onlyOwner {
    emit SetMigrator(migrator, _migrator);
    migrator = _migrator;
}
```

5.7. Unsupported Design for Deflationary Token

ID	IDX-007
Target	MasterChef
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved LuckyLion team has resolved this issue as suggested in commit `5aa5780d15ce4b471d49abb3cba09ac7203975f2`.

5.7.1. Description

In `MasterChef` contract, the users can deposit their tokens to acquire rewards ($LUCKY). The deposited tokens can be a normal token or LP token depending on the pools added by the contract owner.

However, in the `deposit()` function, an issue could arise when the pool uses a deflationary token (the token that reduces the circulating supply itself when it is transferred).

This means the `_amount` that the user deposits will be reduced due to the deflationary mechanism, but the contract recognizes it as the full amount as in line 299.

Masterchef.sol

```
287 function deposit(uint256 _pid, uint256 _amount) public nonReentrant {
288     PoolInfo storage pool = poolInfo[_pid];
289     UserInfo storage user = userInfo[_pid][msg.sender];
290     require(pool.farmStartDate <= block.timestamp,"unable to deposit before the
farm starts.");
291     //can not harvest(deposit 0) before the harvestTimestamp.
292     if (!canHarvest(_pid) && _amount==0){
293         require(pool.harvestTimestamp <= block.timestamp,"can not harvest
before the harvestTimestamp" ); //newly added
294     }
295     updatePool(_pid);
296     payOrLockupPendingLucky(_pid);
297     if (_amount > 0) {
298         pool.lpToken.safeTransferFrom(address(msg.sender), address(this),
_amount);
299         user.amount = user.amount.add(_amount);
```

```
    }
    user.rewardDebt = user.amount.mul(pool.accLuckyPerShare).div(1e12);
    emit Deposit(msg.sender, _pid, _amount);
}
```

The failure of recognizing the token amount could lead to the following scenarios:

Scenario 1: Unable to withdraw staking tokens

Assuming that there is a pool in the `MasterChef` contract which receives a deflationary token ($TOKEN) with 10% burn rate when the token is transferred.

Currently, there is only User A who stakes $TOKEN to the $TOKEN pool in the `MasterChef` contract.

Holder	Balance
User A	100

Total $TOKEN in the `MasterChef` contract: 90

User B deposits 100 $TOKEN to the $TOKEN pool in the `MasterChef` contract. The `MasterChef` contract will receive 90 $TOKEN since $TOKEN is 10% deduction from the deflationary mechanism, in this case 10 $TOKEN.

Holder	Balance
User A	100
User B	100

Total $TOKEN in the `MasterChef` contract: 180

User B then withdraws 100 $TOKEN from the `MasterChef` contract. The `MasterChef` contract will validate whether the withdrawn `_amount` exceeds the `user.amount`.

Masterchef.sol

```
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);
    payOrLockupPendingLucky(_pid);
    if (_amount > 0) {
        user.amount = user.amount.sub(_amount);
        pool.lpToken.safeTransfer(address(msg.sender), _amount);
    }
```

```
    user.rewardDebt = user.amount.mul(pool.accLuckyPerShare).div(1e12);
    emit Withdraw(msg.sender, _pid, _amount);
}
```

Since User B deposited 100 $TOKEN and the balance of $TOKEN in the contract is greater than 100, User B is allowed to withdraw 100 $TOKEN.

Holder	Balance
User A	100
User B	0

Total $TOKEN in the `MasterChef` contract: 80

As a result, if User A decides to withdraw 100 $TOKEN, this transaction will be reverted since the balance in the contract is insufficient.

Scenario 2: Reward Calculation Exploit

Assuming that there is a pool in the `MasterChef` contract which receives a deflationary token ($TOKEN) with 10% burn rate when the token is transferred.

Currently, there are several users who stake $TOKEN to the $TOKEN pool in the `MasterChef` contract with a total supply of 100 $TOKEN.

User A deposits 100 $TOKEN to the contract, and the contract receives 90 $TOKEN due to the deflationary mechanism, resulting in a total supply of 190 $TOKEN.

After that, User A withdraws 100 $TOKEN from staking, the `MasterChef` contract will then calculate the rewards as in line 337.

Masterchef.sol

```
333 function payOrLockupPendingLucky(uint256 _pid) internal {
334     PoolInfo storage pool = poolInfo[_pid];
335     UserInfo storage user = userInfo[_pid][msg.sender];
336
337     uint256 pending =
user.amount.mul(pool.accLuckyPerShare).div(1e12).sub(user.rewardDebt);
338     if (canHarvest(_pid)) {
339         if (pending > 0 || user.rewardLockedUp > 0) {
340             uint256 totalRewards = pending.add(user.rewardLockedUp);
341
342             // reset lockup
343             totalLockedUpRewards =
totalLockedUpRewards.sub(user.rewardLockedUp);
```

```
                user.rewardLockedUp = 0;

                // send rewards
                safeLuckyTransfer(msg.sender, totalRewards);
                emit RewardPaid(msg.sender,totalRewards);
            }
        } else if (pending > 0) {
            user.rewardLockedUp = user.rewardLockedUp.add(pending);
            totalLockedUpRewards = totalLockedUpRewards.add(pending);
            emit RewardLockedUp(msg.sender, _pid, pending);
        }
    }
```

During the calculation, the reward is affected by the total amount of $TOKEN (`lpSupply`) as in line 218.

Masterchef.sol

```
213 function updatePool(uint256 _pid) public {
214     PoolInfo storage pool = poolInfo[_pid];
215     if (block.number <= pool.lastRewardBlock) {
216         return;
217     }
218     uint256 lpSupply = pool.lpToken.balanceOf(address(this));
219     if (lpSupply == 0 || pool.allocPoint == 0) {
220         pool.lastRewardBlock = block.number;
221         return;
222     }
223     uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
224     uint256 luckyReward =
multiplier.mul(luckyPerBlock).mul(pool.allocPoint).div(totalAllocPoint);
225     //new one
226     // check at final to mint exact lucky to complete the round 9 million and
100 millions totalsupply
227     uint256 luckyRewardForDev = luckyReward.mul(devMintingRatio).div(10000);
228     //logic to prevent the minting exceeds the capped totalsupply
229     //1st case, reward for dev will exceed Lucky's totalSupply so we limit the
minting amount to syrup.
230     if (luckyRewardForDev.add(lucky.totalSupply()) > lucky.cap() ) {
231         uint256 remainingReward = lucky.cap().sub(lucky.totalSupply());
232         //in case that remainingReward > capped reward for dev.
233         if (remainingReward.add(accumulatedRewardForDev) > capRewardForDev) {
234             uint256 lastRemainingRewardForDev =
capRewardForDev.sub(accumulatedRewardForDev);
235             lucky.mint(devAddress,lastRemainingRewardForDev);
236             accumulatedRewardForDev =
accumulatedRewardForDev.add(lastRemainingRewardForDev);
237             //the rest is minted to users.
238             lucky.mint(address(syrup),lucky.cap().sub(lucky.totalSupply()));
```

```
        }
        //normal case that dev's caped reward has not been reached yet, but the totalSupply of Lucky is reached.
        else {
            lucky.mint(devAddress, remainingReward);
            //track the token that is minted to dev.
            accumulatedRewardForDev = accumulatedRewardForDev.add(remainingReward);
        }

    }
    //supply cap was not reached and capRewardForDevev still has room to mint for.
    else {
        //capRewardForDev is reached.
        if (luckyRewardForDev.add(accumulatedRewardForDev) > capRewardForDev) {
            uint256 lastRemainingRewardForDev = capRewardForDev.sub(accumulatedRewardForDev);
            lucky.mint(devAddress,lastRemainingRewardForDev);
            //track the token that is minted to dev.
            accumulatedRewardForDev = accumulatedRewardForDev.add(lastRemainingRewardForDev);

            //mint the left portion of dev to the pools.
            lucky.mint(address(syrup),luckyRewardForDev.sub(lastRemainingRewardForDev));

            if (luckyReward.add(lucky.totalSupply()) > lucky.cap() ){
                lucky.mint(address(syrup),lucky.cap().sub(lucky.totalSupply()));
            }
            else {
                lucky.mint(address(syrup),luckyReward);
            }
        }

        else {

            lucky.mint(devAddress,luckyRewardForDev);
            accumulatedRewardForDev = accumulatedRewardForDev.add(luckyRewardForDev);

            if (luckyReward.add(lucky.totalSupply()) > lucky.cap() ){
                lucky.mint(address(syrup),lucky.cap().sub(lucky.totalSupply()));
            }
            else{
```

```
                lucky.mint(address(syrup),luckyReward);
            }

        }
    }
    pool.accLuckyPerShare = pool.accLuckyPerShare.add(luckyReward.mul(1e12).div(lpSupply));
    pool.lastRewardBlock = block.number;
}
```

Since the `MasterChef` contract registers the `user.amount` of User A as 100 $TOKEN, the withdrawn $TOKEN amount will be 100, resulting in reducing the total amount of $TOKEN in the contract to 90 $TOKEN.

Hence, the value of `pool.accLuckyPerShare` can be increased dramatically by manipulating the total amount of $TOKEN (`lpSupply`) to be as low as possible.

User A can repeatedly execute `withdraw()` and `deposit()` functions to drain the $TOKEN in the contract until it is as low as possible, for example, 1 `wei`, causing `accLuckyPerShare` state to be overly inflated, so the users can claim an exceedingly large amount of reward ($LUCK) from the contract.

However, since only LP tokens are planned to be used in `MasterChef` pools, there is no direct impact for this issue.

5.7.2. Remediation

Inspex suggests modifying the logic of the `deposit()` function to validate the amount of the received token from the user instead of using the value of `_amount` parameter directly.

Masterchef.sol

```
function deposit(uint256 _pid, uint256 _amount) public nonReentrant {
    PoolInfo storage pool = poolInfo[_pid];
    UserInfo storage user = userInfo[_pid][msg.sender];
    require(pool.farmStartDate <= block.timestamp,"unable to deposit before the farm starts.");
    //can not harvest(deposit 0) before the harvestTimestamp.
    if (!canHarvest(_pid) && _amount==0){
        require(pool.harvestTimestamp <= block.timestamp,"can not harvest before the harvestTimestamp" ); //newly added
    }
    updatePool(_pid);
    payOrLockupPendingLucky(_pid);
    if (_amount > 0) {
        uint256 currentBal = pool.lpToken.balanceOf(address(this));
        pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
        uint256 receivedAmount = pool.lpToken.balanceOf(address(this)) -
```

```
currentBal;
            user.amount = user.amount.add(receivedAmount);
      }
      user.rewardDebt = user.amount.mul(pool.accLuckyPerShare).div(1e12);
      emit Deposit(msg.sender, _pid, _amount);
}
```

5.8. Improper Function Visibility

ID	IDX-008
Target	Masterchef SyrupBar
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 `5aa5780d15ce4b471d49abb3cba09ac7203975f2`.

5.8.1. 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.

For example, the following source code shows that the `add()` function of the `MasterChef` contract is set to public and it is never called from any internal function.

Masterchef.sol

```
function add(uint256 _allocPoint, IERC20 _lpToken, uint256 _harvestIntervalInMinutes,uint256 _farmStartIntervalInMinutes, bool _withUpdate) public onlyOwner {
    uint256 _harvestTimestampInUnix = block.timestamp + (_harvestIntervalInMinutes *60); //*60 to convert from minutes to second.
    uint256 _farmStartTimestampInUnix = block.timestamp + (_farmStartIntervalInMinutes *60);
    if (_withUpdate) {
        massUpdatePools();
    }
    uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
    totalAllocPoint = totalAllocPoint.add(_allocPoint);
    poolInfo.push(PoolInfo({
        lpToken: _lpToken,
        allocPoint: _allocPoint,
        lastRewardBlock: lastRewardBlock,
        accLuckyPerShare: 0,
```

```
        harvestTimestamp: _harvestTimestampInUnix,
        farmStartDate : _farmStartTimestampInUnix
    }));
    emit PoolAdded(_lpToken,_allocPoint,_harvestTimestampInUnix,
_farmStartTimestampInUnix);
}
```

The following table contains all functions that have `public` visibility and are never called from any internal function.

File	Contract	Function
Masterchef.sol (L:148)	MasterChef	set()
Masterchef.sol (L:196)	MasterChef	setMigrator()
Masterchef.sol (L:201)	MasterChef	migrate()
Masterchef.sol (L:287)	MasterChef	deposit()
Masterchef.sol (L:306)	MasterChef	withdraw()
Masterchef.sol (L:321)	MasterChef	emergencyWithdraw()
Masterchef.sol (L:363)	MasterChef	setDevAddress()
Masterchef.sol (L:360)	MasterChef	updateLuckyPerBlock()
SyrupBar.sol (L:22)	SyrupBar	safeLuckyTransfer()

5.8.2. Remediation

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

Masterchef.sol

```
function add(uint256 _allocPoint, IERC20 _lpToken, uint256
_harvestIntervalInMinutes,uint256 _farmStartIntervalInMinutes, bool
_withUpdate) external onlyOwner {
    uint256 _harvestTimestampInUnix = block.timestamp +
(_harvestIntervalInMinutes *60); //*60 to convert from minutes to second.
    uint256 _farmStartTimestampInUnix = block.timestamp +
(_farmStartIntervalInMinutes *60);
    if (_withUpdate) {
        massUpdatePools();
    }
    uint256 lastRewardBlock = block.number > startBlock ? block.number :
startBlock;
```

```
    totalAllocPoint = totalAllocPoint.add(_allocPoint);
    poolInfo.push(PoolInfo({
        lpToken: _lpToken,
        allocPoint: _allocPoint,
        lastRewardBlock: lastRewardBlock,
        accLuckyPerShare: 0,
        harvestTimestamp: _harvestTimestampInUnix,
        farmStartDate : _farmStartTimestampInUnix
    }));
    emit PoolAdded(_lpToken,_allocPoint,_harvestTimestampInUnix,
_farmStartTimestampInUnix);
}
```

6. Appendix

6.1. About Inspex

CYBERSECURITY PROFESSIONAL SERVICE

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]

inspex

CYBERSECURITY PROFESSIONAL SERVICE