AMM, Farm & Wrapped Tokens

Smart Contract Audit Report
Prepared for Foodcourt Finance

Date Issued:	Aug 9, 2021
Project ID:	AUDIT2021006
Version:	v1.0
Confidentiality Level:	Public

CYBERSECURITY PROFESSIONAL SERVICE

Report Information

Project ID	AUDIT2021006
Version	v1.0
Client	Foodcourt Finance
Project	AMM, Farm & Wrapped Tokens
Auditor(s)	Weerawat Pawanawiwat Pongsakorn Sommalai Suvicha Buakhom Patipon Suwanbol
Author	Weerawat Pawanawiwat
Reviewer	Pongsakorn Sommalai
Confidentiality Level	Public

Version History

Version	Date	Description	Author(s)
1.0	Aug 9, 2021	Full report	Weerawat Pawanawiwat

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 Foodcourt Finance, Inspex team conducted an audit to verify the security posture of the AMM, Farm & Wrapped Tokens smart contracts between Jul 5, 2021 and Jul 8, 2021. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of AMM, Farm & Wrapped Tokens 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 critical, 3 high, 2 medium, 3 low, and 2 info-severity issues. With the project team's prompt response, 2 critical, 3 high, 2 medium, 2 low, and 2 info-severity issues were resolved in the reassessment, while 1 low-severity issue was acknowledged by the team. Therefore, Inspex trusts that AMM, Farm & Wrapped Tokens 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), Inpex 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

Foodcourt Finance is an Automated Market Maker (AMM) and Yield Farming protocol that is forked from Pancakeswap V2 with deflationary tokens wrapping feature added.

AMM, Farm & Wrapped Tokens are the core functionalities of Foodcourt, allowing the users to swap tokens on the platform, gain rewards from yield farming, and wrap deflationary tokens to farm on special pools.

Scope Information:

Project Name	AMM, Farm & Wrapped Tokens
Website	https://foodcourt.finance/
Smart Contract Type	Ethereum Smart Contract
Programming Language	Solidity

Audit Information:

Audit Method	Whitebox
Audit Date	Jul 05, 2021 - Jul 08, 2021
Reassessment Date	Jul 21, 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 is 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: c43ad98c58c518bc9faa350246ce33a94983f638)

Contract	Location (URL)
Cafeteria	https://github.com/foodcourtofficial/contracts/blob/c43ad98c58c518bc9faa350246ce33a94983f638/contracts/Cafeteria.sol
CouponToken	https://github.com/foodcourtofficial/contracts/blob/c43ad98c58c518bc9faa350246ce33a94983f638/contracts/CouponToken.sol
FoodcourtFactory	https://github.com/foodcourtofficial/contracts/blob/c43ad98c58c518bc9faa350246ce33a94983f638/contracts/FoodcourtFactory.sol
FoodcourtRouter	https://github.com/foodcourtofficial/contracts/blob/c43ad98c58c518bc9faa350246ce33a94983f638/contracts/FoodcourtRouter.sol
Mintable	https://github.com/foodcourtofficial/contracts/blob/c43ad98c58c518bc9faa350246ce33a94983f638/contracts/Mintable.sol
RSafeToken	https://github.com/foodcourtofficial/contracts/blob/c43ad98c58c518bc9faa350246ce33a94983f638/contracts/RSafeToken.sol
SnackBar	https://github.com/foodcourtofficial/contracts/blob/c43ad98c58c518bc9faa350246ce33a94983f638/contracts/SnackBar.sol
SnackBarFactory	https://github.com/foodcourtofficial/contracts/blob/c43ad98c58c518bc9faa350246ce33a94983f638/contracts/SnackBarFactory.sol
WSafeToken	https://github.com/foodcourtofficial/contracts/blob/c43ad98c58c518bc9faa350246ce33a94983f638/contracts/WSafeToken.sol

Reassessment: (Commit: 8c2107a33f453cc6876ea5d9929cbff68d2d45a9)

Contract	Location (URL)
Cafeteria	https://github.com/foodcourtofficial/contracts/blob/8c2107a33f453cc6876ea5d9929cbff68d2d45a9/contracts/Cafeteria.sol
CouponToken	https://github.com/foodcourtofficial/contracts/blob/8c2107a33f453cc6876ea5d9929cbff68d2d45a9/contracts/CouponToken.sol
FoodcourtFactory	https://github.com/foodcourtofficial/contracts/blob/8c2107a33f453cc6876ea5d9929cbff68d2d45a9/contracts/FoodcourtFactory.sol
FoodcourtRouter	https://github.com/foodcourtofficial/contracts/blob/8c2107a33f453cc6876ea5d9929cbff68d2d45a9/contracts/FoodcourtRouter.sol

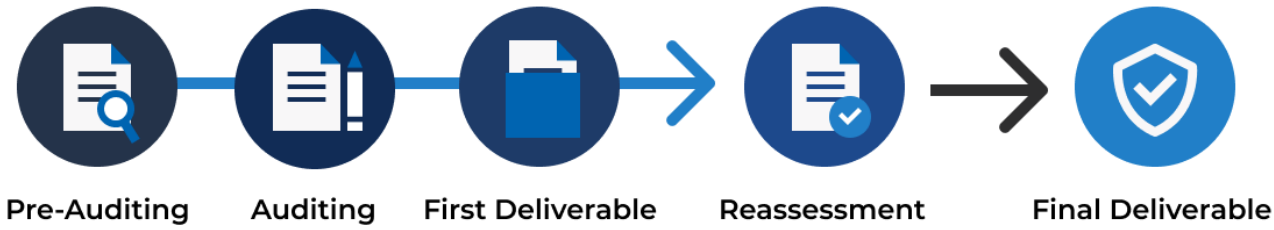
Mintable	https://github.com/foodcourtofficial/contracts/blob/8c2107a33f453cc6876ea5d9929cbff68d2d45a9/contracts/Mintable.sol
RSafeToken	https://github.com/foodcourtofficial/contracts/blob/8c2107a33f453cc6876ea5d9929cbff68d2d45a9/contracts/wSafe/RSafeToken.sol
SnackBar	https://github.com/foodcourtofficial/contracts/blob/8c2107a33f453cc6876ea5d9929cbff68d2d45a9/contracts/SnackBar.sol
SnackBarFactory	https://github.com/foodcourtofficial/contracts/blob/8c2107a33f453cc6876ea5d9929cbff68d2d45a9/contracts/SnackBarFactory.sol
WSafeToken	https://github.com/foodcourtofficial/contracts/blob/8c2107a33f453cc6876ea5d9929cbff68d2d45a9/contracts/wSafe/WSafeToken.sol

The assessment scope covers only the in-scope smart contracts and the smart contracts that they are inherited 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
Advanced
Business Logic Flaw
Ownership Takeover
Broken Access Control
Broken Authentication
Upgradable Without Timelock
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

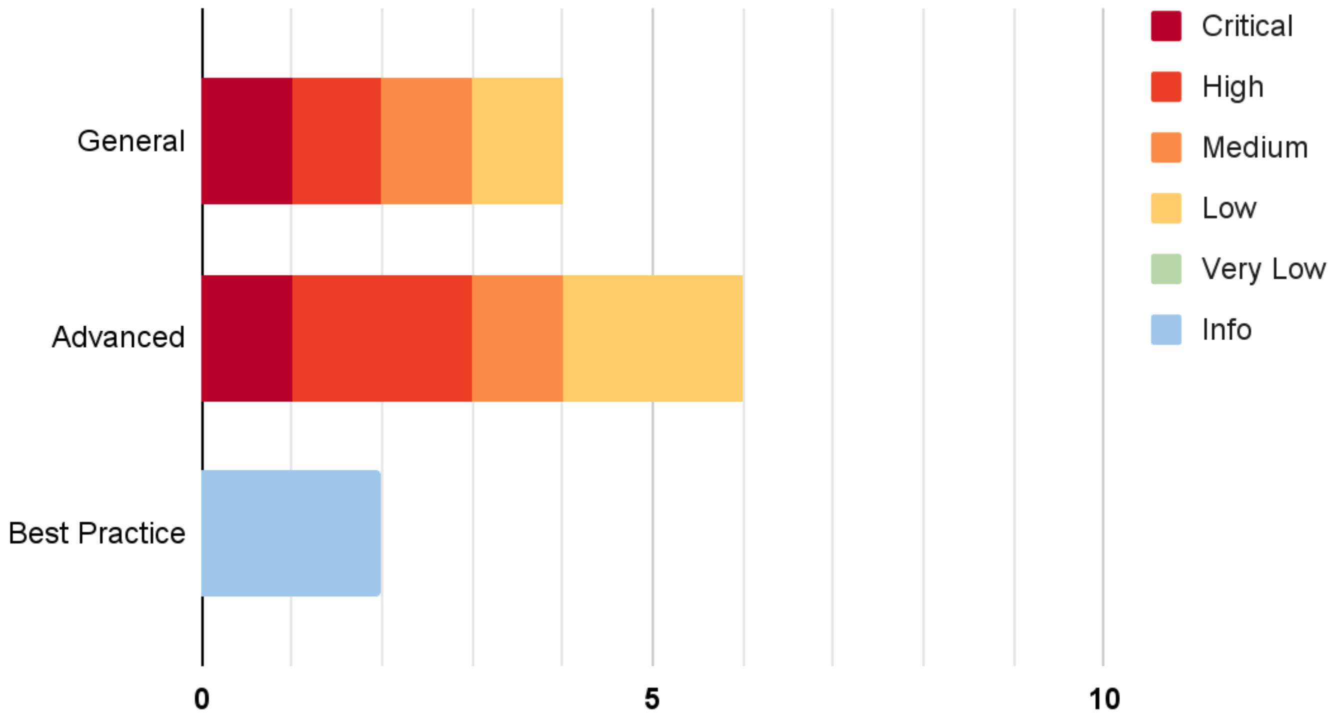
Both likelihood and impact can be categorized into three levels: **Low**, **Medium**, and **High**.

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 12 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	Unrestricted Minting of Reward Token	Advanced	Critical	Resolved
IDX-002	Token Manual Minting by Contract Owner	General	Critical	Resolved *
IDX-003	Improper Reward Calculation (Same Token)	Advanced	High	Resolved
IDX-004	Design Flaw in withdrawFee100 Pool	Advanced	High	Resolved
IDX-005	Centralized Control of State Variable	General	High	Resolved
IDX-006	Improper Reward Calculation (_withUpdate)	Advanced	Medium	Resolved
IDX-007	Unsafe Token Transfer	General	Medium	Resolved
IDX-008	Design Flaw in massUpdatePool() Function	General	Low	Acknowledged
IDX-009	Improper Condition Checking in emergencyWithdraw() Function	Advanced	Low	Resolved
IDX-010	Addition of Pool With Duplicated ibToken	Advanced	Low	Resolved
IDX-011	Improper Function Visibility	Best Practice	Info	Resolved *
IDX-012	Inexplicit Solidity Compiler Version	Best Practice	Info	Resolved

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

5. Detailed Findings Information

5.1. Unrestricted Minting of Reward Token

ID	IDX-001
Target	WSafeToken
Category	Advanced Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity: Critical** **Impact: High** Anyone can mint an unlimited amount of `feeToken` which could be used to stake for rewards as $COUPON tokens. **Likelihood: High** It is very likely that tokens will be minted since there is no cost and restriction.
Status	Resolved Foodcourt Finance team has resolved this issue as suggested in commit 9da4375e0fe1cf866fbc863ee375d018b288d5e8.

5.1.1. Description

The `distributeReward()` function is designed to compensate for the transfer fee of the deflationary token on wrapping. When a user wraps a deflationary token (`WSafeToken`), the `feeToken` will be minted for the same amount that is deducted in line 90 as a transfer fee.

WSafeToken.sol

```
77 function wrap(uint256 amount) public returns (uint256 totalReceived) {
78     uint256 balanceBefore = safeToken.balanceOf(address(this));
79     uint256 wrapRatio = getWrapRatio();
80     safeToken.transferFrom(msg.sender, address(this), amount);
81 
82     uint256 safeBalance = safeToken.balanceOf(address(this));
83 
84     totalReceived = safeBalance - balanceBefore;
85     totalReceived = totalReceived * 1e9 / wrapRatio;
86     uint256 wrapFee = totalReceived * wrapFeeRate / 1000;
87 
88     totalReceived -= wrapFee;
89 
90     distributeReward(amount);
91 
```

```
    _mint(msg.sender, totalReceived);
    _mint(devAddr, wrapFee);

    emit Wrap(msg.sender, amount, wrapFee, totalReceived);
}
```

However, the `distributeReward()` function has `public` visibility, so there is no access restriction. This allows anyone to execute this function directly without calling `wrap()` function, resulting in an unlimited mint amount of fee token being transferred to the function caller.

WSafeToken.sol

```
function distributeReward(uint256 amount) public {
    uint256 totalReward = amount * safeFeeRate / 1000;
    feeToken.mint(msg.sender, totalReward);
    emit DistributeReward(msg.sender, amount, totalReward);
}
```

5.1.2. Remediation

Inspex suggests modifying the visibility of `distributeReward()` function from `public` to `internal` as shown below:

WSafeToken.sol

```
function distributeReward(uint256 amount) internal {
    uint256 totalReward = amount * safeFeeRate / 1000;
    feeToken.mint(msg.sender, totalReward);
    emit DistributeReward(msg.sender, amount, totalReward);
}
```

However, for the `WSafeToken` tokens already deployed, Inspex recommends setting the `safeFeeRate` to 0 to prevent this issue's impact by executing the `setSafeFeeRate()` function with 0 `safeFeeRate`.

5.2. Token Manual Minting by Contract Owner

ID	IDX-002
Target	CouponToken
Category	General Smart Contract Vulnerability
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	**Severity: Critical** **Impact: High** The contract owner can arbitrarily mint the affected tokens. **Likelihood: High** It is very likely that the contract owner can set his wallet address to be the `onlyMinter` and call the `mint()` function.
Status	Resolved * In the future, multiple `Cafeteria` contracts are expected to be used, and Northbridge will be added as a minter for $COUPON along with new rewarding mechanisms which require minting ability. Therefore, the Foodcourt Finance team has mitigated this issue as follows: 1. The minter timelock of the $COUPON contract has been set to 7 days. So, new contracts will be delayed for 7 days before being able to mint, and 7 days is long enough for the investors to decide whether they will accept the changes or not. Furthermore, the minter timelock cannot be set to a shorter duration than the previous, so there is no risk of the minter timelock being removed. 2. To prevent the bridge from over-minting, the Foodcourt Team will set a daily mint limit of $COUPON to fit the bridging demand as much as possible. The minter timelock of $COUPON has been set to 7 days in the following transaction: https://www.bscscan.com/tx/0x728ae85fb660d2b8f96065c63900941a444704403b48cc1a94f22c9a9798afb9

5.2.1. Description

In the `CouponToken` contract, the `mint()` function has `onlyMinter` as a modifier as shown below.

CouponToken.sol

```
function mint(address _to, uint256 _amount) public onlyMinter {
    increaseMint(_amount);
    _mint(_to, _amount);
}
```

The `onlyMinter` modifier checks that the caller is a minter by verifying the `_msgSender()` with `allowMinting` mapping.

Mintable.sol

```
modifier onlyMinter {
    require(allowMinting[_msgSender()].allowed, "not minter");
    require(block.timestamp >= allowMinting[_msgSender()].timelock, "mint locked");
    _;
}
```

The contract owner can set any address to be the minter by calling the `setAllowMinting()` function.

Mintable.sol

```
function setAllowMinting(address _address, bool _allowed) public onlyOwner {
    if (_allowed) {
        allowMinting[_address].allowed = true;
        allowMinting[_address].timelock = block.timestamp + minterTimelock;
    } else {
        allowMinting[_address].allowed = false;
        allowMinting[_address].timelock = 0;
    }

    emit AllowMinter(_msgSender(), _address, _allowed);
}
```

As a result, although the contract owner is currently not set to be the minter, the owner will still be able to set the owner's wallet address as the minter and call the `mint()` function in order to mint $COUPON.

5.2.2. Remediation

Inspex suggests performing the following actions:

- Remove the `Mintable` library from the token contract
- Set the `onlyOwner` as the modifier of `mint()` function
- Set the owner of the tokens to be the `Cafeteria` contract.

However, Foodcourt Finance has already deployed the $COUPON contracts to the BSC mainnet. To fix this issue, Inspex suggests doing the following actions:

- Set the $COUPON minter as `Cafeteria` contract only
- Renounce the ownership of CouponToken contract

5.3. Improper Reward Calculation (Same Token)

ID	IDX-003
Target	Cafeteria
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** High **Impact:** Medium The reward of the pool that has the same staking token as the reward token will be slightly lower than what it should be. **Likelihood:** High There is a pool that has the same staking token as the reward token deployed, so the miscalculation will happen everytime the reward is calculated.
Status	Resolved Foodcourt Finance team has resolved this issue in commit 9da4375e0fe1cf866fbc863ee375d018b288d5e8 by minting $COUPON reward to the `CouponReserver` contract instead.

5.3.1. Description

In the `Cafeteria` 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 the same as the reward token ($COUPON) or not.

Cafeteria.sol

```
function add(uint256 _allocPoint, IBEP20 _lpToken, uint16 _depositFeeBP, bool
_withdrawFee100, bool _withUpdate) public onlyOwner nonDuplicated(_lpToken) {
    require(_depositFeeBP <= 10000, "invalid deposit fee basis points");
    if (_withUpdate) {
        massUpdatePools();
    }
    uint256 lastRewardBlock = block.number > startBlock ? block.number :
startBlock;
    totalAllocPoint = totalAllocPoint.add(_allocPoint);
    poolExistence[_lpToken] = true;
    poolInfo.push(PoolInfo({
        lpToken: _lpToken,
        allocPoint: _allocPoint,
        lastRewardBlock: lastRewardBlock,
        accCouponPerShare: 0,
```

```
        depositFeeBP: _depositFeeBP,
        withdrawFee100: _withdrawFee100
    }));
}
```

When the `_lpToken` is the same token as $COUPON, reward calculation for that pool in the `updatePool()` function can be incorrect. This is because the current balance of the `_lpToken` in the contract is used in the calculation of the reward. Since the `_lpToken` is the same token as the reward, the reward minted to the contract will inflate the value of `lpSupply`, causing the reward of that pool to be less than what it should be.

Cafeteria.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 couponReward =
multiplier.mul(couponPerBlock).mul(pool.allocPoint).div(totalAllocPoint);
    coupon.mint(devaddr, couponReward.div(10));
    coupon.mint(address(this), couponReward);
    pool.accCouponPerShare =
pool.accCouponPerShare.add(couponReward.mul(1e12).div(lpSupply));
    pool.lastRewardBlock = block.number;
}
```

5.3.2. Remediation

Inspex suggests checking the value of the `_lpToken` in the `add()` function to prevent the pool with the same staking token as the reward token from being added, for example:

Cafeteria.sol

```
function add(uint256 _allocPoint, IBEP20 _lpToken, uint16 _depositFeeBP, bool
_withdrawFee100, bool _withUpdate) public onlyOwner nonDuplicated(_lpToken) {
    require(_depositFeeBP <= 10000, "invalid deposit fee basis points");
    require(_lpToken != coupon, '_lpToken is COUPON');
    if (_withUpdate) {
        massUpdatePools();
    }
    uint256 lastRewardBlock = block.number > startBlock ? block.number :
```

```
startBlock;
    totalAllocPoint = totalAllocPoint.add(_allocPoint);
    poolExistence[_lpToken] = true;
    poolInfo.push(PoolInfo({
        lpToken: _lpToken,
        allocPoint: _allocPoint,
        lastRewardBlock: lastRewardBlock,
        accCouponPerShare: 0,
        depositFeeBP: _depositFeeBP,
        withdrawFee100: _withdrawFee100
    }));
}
```

If the pool with the same staking token as the reward token is required, Inspex suggests minting the reward token to another contract to prevent the amount of the staked token from being mixed up with the reward token.

However, if the contract cannot be modified and redeployed, Inspex suggests implementing a `Shield` contract with all `onlyOwner` functions of the `Cafeteria` contract to handle the functions' logics. Additional checkings can be done before executing the actual `add()` function, for example:

Shield.sol

```
function add(uint256 _allocPoint, IBEP20 _lpToken, uint16 _depositFeeBP, bool
_withdrawFee100, bool _withUpdate) public onlyOwner {
    require(_lpToken != coupon, '_lpToken is COUPON');
    cafeteria.add(_allocPoint, _lpToken, _depositFeeBP, _withdrawFee100,
_withUpdate);
}
```

The value of `coupon` and `cafeteria` variables should be set accordingly.

The owner of the `Cafeteria` contract should then be set to the `Shield` contract using the `transferOwnership()` function.

For the pool already added, Inspex suggests performing the following actions:

- Deploy the wrapped token of $COUPON
- Set the `allocPoint` of the affected pool to 0
- Create a new pool with the deployed wrapped tokens
- Migrate all staked tokens from the affected pool to the new pool

Please note that the fixes for other issues are not yet applied in the examples above.

5.4. Design Flaw in withdrawFee100 Pool

ID	IDX-004
Target	Cafeteria
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** High **Impact:** Medium The product owner can use the destroyed tokens from the fee wallet to restake in available pools. **Likelihood:** High It is very likely that the product owner can access the fee wallet, and there is no restriction to prevent this attack from happening.
Status	Resolved Foodcourt Finance team has resolved this issue as suggested in commit 8c2107a33f453cc6876ea5d9929cbff68d2d45a9. Also, the `withdrawFee100` flag is renamed to `isRewardToken` to prevent misunderstanding, and only the pools of `RSafeToken` will have this flag set to true.

5.4.1. Description

In the `deposit()` function, when the user harvests the reward from `withdrawFee100` pools, all tokens staked will be transferred to the fee wallet as shown below in line 216.

Cafeteria.sol

```
191 function deposit(uint256 _pid, uint256 _amount) public nonReentrant {
192     PoolInfo storage pool = poolInfo[_pid];
193     UserInfo storage user = userInfo[_pid][msg.sender];
194     updatePool(_pid);
195 
196     uint256 pending =
user.amount.mul(pool.accCouponPerShare).div(1e12).sub(user.rewardDebt) +
user.lockedReward;
197     if (pending > 0) {
198         if (_amount == 0 || !pool.withdrawFee100) {
199             safeCouponTransfer(msg.sender, pending);
200             user.lockedReward = 0;
201         } else {
202             user.lockedReward = pending;
203         }
204     }
```

```
    if (_amount > 0) {
        pool.lpToken.safeTransferFrom(address(msg.sender), address(this),
_amount);
        if (pool.depositFeeBP > 0){
            uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
            pool.lpToken.safeTransfer(feeAddress, depositFee);
            user.amount = user.amount.add(_amount).sub(depositFee);
        } else {
            user.amount = user.amount.add(_amount);
        }
    } else if (pool.withdrawFee100){
        pool.lpToken.safeTransfer(feeAddress, user.amount);
        user.amount = 0;
    }
    user.rewardDebt = user.amount.mul(pool.accCouponPerShare).div(1e12);
    emit Deposit(msg.sender, _pid, _amount);
}
```

As a result, the product owner who can access the fee wallet can restake those tokens to the available pools.

5.4.2. Remediation

Inspex suggests burning the token from `withdrawFee100` pool transferred to the fee address, and modifying the contract to burn all tokens from the `withdrawFee100` pools by transferring those tokens to the dead address (`0x000000000000000000000000000000000000dEaD`) as shown in the following example:

Cafeteria.sol

```
function deposit(uint256 _pid, uint256 _amount) public nonReentrant {
    PoolInfo storage pool = poolInfo[_pid];
    UserInfo storage user = userInfo[_pid][msg.sender];
    updatePool(_pid);

    uint256 pending =
user.amount.mul(pool.accCouponPerShare).div(1e12).sub(user.rewardDebt) +
user.lockedReward;
    if (pending > 0) {
        if (_amount == 0 || !pool.withdrawFee100) {
            safeCouponTransfer(msg.sender, pending);
            user.lockedReward = 0;
        } else {
            user.lockedReward = pending;
        }
    }

    if (_amount > 0) {
```

```
        pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
        if (pool.depositFeeBP > 0){
            uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
            pool.lpToken.safeTransfer(feeAddress, depositFee);
            user.amount = user.amount.add(_amount).sub(depositFee);
        } else {
            user.amount = user.amount.add(_amount);
        }
    } else if (pool.withdrawFee100){
        pool.lpToken.safeTransfer("0x000000000000000000000000000000000000dEaD", user.amount);
        user.amount = 0;
    }
    user.rewardDebt = user.amount.mul(pool.accCouponPerShare).div(1e12);
    emit Deposit(msg.sender, _pid, _amount);
}
```

However, if the contract cannot be modified and redeployed, Inspex suggests setting the `feeAddress` to the dead address and setting the `depositFee` of all pools to 0.

5.5. Centralized Control of State Variable

ID	IDX-005
Target	Cafeteria SnackBar Mintable WSafeToken
Category	General Smart Contract Vulnerability
CWE	CWE-710: Improper Adherence to Coding Standard
Risk	**Severity:** High **Impact:** Medium The controlling authorities can change the critical state variables to gain additional profit. Thus, it is unfair to the other users. **Likelihood:** High There is nothing to restrict the changes from being done; however, the changes are limited by fixed values in the smart contracts.
Status	Resolved Foodcourt Finance team has resolved this issue by implementing 24-hour `Timelock` over the following contracts: - `Cafeteria` (1) Contract Address: 0xe43b7c5c4c2df51306cceb7cbc4b2fcc038874f1 Owner Address (`Timelock`): 0xbB3eB5B0c23030035c390B64cB32529FEe24921c - `Cafeteria` (2) Contract Address: 0xc0e2d1726e3465ea016ba2559b08664d905b0bd2 Owner Address (`Timelock`): 0x815E31e7d3D7348Af6F0Ea393DD2372270f22639 - `SnackBar` Contract Address: 0xEa15086a831a08262bAced9055E248dB7564B289 Owner Address (`Timelock`): 0x473a36afc9dd0c31687e754f6be39ba2d26c0af2 - `WMMP` (`WSafeToken`) Contract Address: 0x422d0A431D8fb752e3697e90BA04b3324Ea0Cb4a Owner Address (`Timelock`): 0x11f453367883aaF9fD53F9F7CBc3ECBB33265e9B - `WSafeMars` (`WSafeToken`) Contract Address: 0x40733aBc9Acb7d48Caa632ee83E4e7B3d0008d9D Owner Address (`Timelock`): 0x5A43f9F535b34D67f7bb528a3211436a7c96aD27 - `WSafeMoon` (`WSafeToken`) Contract Address: 0xa3863434a1Fc699185b3E6809a933056D1178366 Owner Address (`Timelock`): 0xb060146303f04B10E60e22E21525039e25381E59 For the `CouponToken` contract that inherits the `Mintable` contract, the minter timelock has been set to 7 days. Therefore, new contracts will be delayed for 7 days before being

	able to mint. The timelock has been set in the following transaction: https://www.bscscan.com/tx/0x728ae85fb660d2b8f96065c63900941a444704403b48cc1a94f22c9a9798afb9

5.5.1. Description

Critical state variables can be updated at 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, there is currently no constraint to prevent the authorities from modifying these variables without notifying the users.

The controllable privileged state update functions are as follows:

Target	Function	Modifier
Cafeteria.sol (L:106)	add()	onlyOwner
Cafeteria.sol (L:125)	set()	onlyOwner
Cafeteria.sol (L:281)	updateEmissionRate()	onlyOwner
SnackBar.sol (L:195)	emergencyRewardWithdraw()	onlyOwner
SnackBar.sol (L:205)	recoverWrongTokens()	onlyOwner
SnackBar.sol (L:218)	stopReward()	onlyOwner
SnackBar.sol (L:228)	updatePoolLimitPerUser()	onlyOwner
SnackBar.sol (L:245)	updateRewardPerBlock()	onlyOwner
SnackBar.sol (L:257)	updateStartAndEndBlocks()	onlyOwner
Mintable.sol (L:25)	setMinterTimelock()	onlyOwner
Mintable.sol (L:31)	setAllowMinting()	onlyOwner
Mintable.sol (L:43)	setDailyMintLimit()	onlyOwner
WSafeToken.sol (L:39)	setSafeFeeRate()	onlyOwner
WSafeToken.sol (L:47)	setWrapFeeRate()	onlyOwner
WSafeToken.sol (L:55)	setUnwrapFeeRate()	onlyOwner
Ownable.sol	renounceOwnership()	onlyOwner

Ownable.sol	transferOwnership()	onlyOwner

Please note that the `Ownable` contract is inherited from `OpenZeppelin` library and `contracts/libs/Ownable.sol`.

5.5.2. 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 reasonable amount of time

5.6. Improper Reward Calculation (_withUpdate)

ID	IDX-006
Target	Cafeteria
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** Medium The $COUPON reward miscalculation can lead to unfair $COUPON token distribution. **Likelihood:** Medium This issue happens whenever the `totalAllocPoint` is modified and the `_withUpdate` parameter is set to false.
Status	Resolved Foodcourt Finance team has resolved this issue as suggested in commit 11d4ec0a0755a0796308b73f39a77eb62f0b0ccc.

5.6.1. Description

The `totalAllocPoint` variable is used to determine the portion that each pool would get from the total rewards 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 rewards 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.

Cafeteria.sol

```
function add(uint256 _allocPoint, IBEP20 _lpToken, uint16 _depositFeeBP, bool
_withdrawFee100, bool _withUpdate) public onlyOwner nonDuplicated(_lpToken) {
    require(_depositFeeBP <= 10000, "invalid deposit fee basis points");
    if (_withUpdate) {
        massUpdatePools();
    }
    uint256 lastRewardBlock = block.number > startBlock ? block.number :
startBlock;
    totalAllocPoint = totalAllocPoint.add(_allocPoint);
    poolExistence[_lpToken] = true;
    poolInfo.push(PoolInfo({
        lpToken: _lpToken,
        allocPoint: _allocPoint,
```

```
        lastRewardBlock: lastRewardBlock,
        accCouponPerShare: 0,
        depositFeeBP: _depositFeeBP,
        withdrawFee100: _withdrawFee100
    }));
}
```

Cafeteria.sol

```
function set(uint256 _pid, uint256 _allocPoint, uint16 _depositFeeBP, bool _withUpdate) public onlyOwner {
    require(_depositFeeBP <= 10000, "invalid deposit fee basis points");

    if (_withUpdate) {
        massUpdatePools();
    }

    totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint);
    poolInfo[_pid].allocPoint = _allocPoint;
    poolInfo[_pid].depositFeeBP = _depositFeeBP;

    // For emergency fix of 100% withdrawal fee
    if (_allocPoint == 0) {
        poolExistence[poolInfo[_pid].lpToken] = false;
    } else {
        poolExistence[poolInfo[_pid].lpToken] = true;
    }
}
```

For example:

Assuming that at block 8239999, `couponPerBlock` is set to 10 $COUPON per block, pool 0 `allocPoint` is set to 300, and `totalAllocPoint` is set to 9605.

Block	Action
8239999	All pools' rewards are updated
8249999	A new pool is added using the `add()` function, causing the `totalAllocPoint` to be changed from 9605 to 10000
8259999	The pools' rewards are updated once again.

From current logic, the total rewards allocated to the pool 0 during block 8239999 to block 8259999 is equal to 6,000.00 $COUPON calculated using the following equation:

```
pool 0 allocPoint / totalAllocPoint * couponPerBlock * totalRewardBlock
300 / 10,000 * 10 * 20000 = 6,000.00
```

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

- 0.3123 $COUPON per block, from block 8239999 to block 8249999, with a proportion of 300/9,605 = 3,123.37 $COUPON
- 0.3000 $COUPON per block, from block 8249999 to block 8259999, with a proportion of 300/10,000 = 3,000.00 $COUPON

The correct total $COUPON rewards is 6,123.37 $COUPON, which is different from the miscalculated reward by 123.37 $COUPON

5.6.2. Remediation

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

Cafeteria.sol

```
function add(uint256 _allocPoint, IBEP20 _lpToken, uint16 _depositFeeBP, bool
_withdrawFee100) public onlyOwner nonDuplicated(_lpToken) {
    require(_depositFeeBP <= 10000, "invalid deposit fee basis points");
    massUpdatePools();
    uint256 lastRewardBlock = block.number > startBlock ? block.number :
startBlock;
    totalAllocPoint = totalAllocPoint.add(_allocPoint);
    poolExistence[_lpToken] = true;
    poolInfo.push(PoolInfo({
        lpToken: _lpToken,
        allocPoint: _allocPoint,
        lastRewardBlock: lastRewardBlock,
        accCouponPerShare: 0,
        depositFeeBP: _depositFeeBP,
        withdrawFee100: _withdrawFee100
    }));
}
```

Cafeteria.sol

```
function set(uint256 _pid, uint256 _allocPoint, uint16 _depositFeeBP,) public
onlyOwner {
    require(_depositFeeBP <= 10000, "invalid deposit fee basis points");
    massUpdatePools();
    totalAllocPoint =
```

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

    // For emergency fix of 100% withdrawal fee
    if (_allocPoint == 0) {
        poolExistence[poolInfo[_pid].lpToken] = false;
    } else {
        poolExistence[poolInfo[_pid].lpToken] = true;
    }
}
```

However, if the contract cannot be modified and redeployed, Inspex suggests implementing a `Shield` contract with all `onlyOwner` functions of the `Cafeteria` contract to handle the functions' logics. The `Shield` contract can set the `_withUpdate` variable to `true` for `add()` and `set()` functions, for example:

Shield.sol

```
pragma solidity 0.6.12;

import "./libs/SafeMath.sol";
import "./libs/Ownable.sol";
import "./Cafeteria.sol";

contract Shield is Ownable {
    using SafeMath for uint256;

    Cafeteria public cafeteria;

    constructor(address _owner, Cafeteria _cafeteria) public {
        transferOwnership(_owner);
        cafeteria = _cafeteria;
    }

    function addPool(uint256 _allocPoint, address _lpToken, uint16 _depositFeeBP, bool _withdrawFee100) external onlyOwner {
        cafeteria.add(_allocPoint, _lpToken, _depositFeeBP, _withdrawFee100, true);
    }

    function setPool(uint256 _pid, uint256 _allocPoint, uint16 _depositFeeBP) external onlyOwner {
        cafeteria.set(_pid, _allocPoint, _depositFeeBP, true);
    }

    function updateEmissionRate(uint256 _couponPerBlock) external onlyOwner {
        cafeteria.updateEmissionRate(_couponPerBlock);
```

```
    }

    ...
```

The owner of the `Cafeteria` contract should then be set to the `Shield` contract using the `transferOwnership()` function.

Please note that the fixes for other issues are not yet applied in the examples above.

5.7. Unsafe Token Transfer

ID	IDX-007
Target	WSafeToken
Category	General Smart Contract Vulnerability
CWE	CWE-710: Improper Adherence to Coding Standard
Risk	**Severity:** Medium **Impact:** High The feeToken can be minted without transferring safeToken into the contract. **Likelihood:** Low Only improperly implemented tokens that do not revert the transaction with invalid transfer amount are affected.
Status	Resolved Foodcourt Finance team has resolved this issue as suggested in commit 9da4375e0fe1cf866fbc863ee375d018b288d5e8.

5.7.1. Description

External `ERC20` tokens can be added to the contract as a `safeToken` using the `constructor()` function. `ERC20` tokens can be improperly implemented, allowing the execution of failed `transfer()` and `transferFrom()` functions without reverting when the invalid transfer amount occurs. This can cause significant damage to the smart contract if not enough tokens are available.

For example, in the `wrap()` function, if the user wraps a `safeToken` without having a sufficient amount, but there is no reverting in the `transferFrom()` function of `safeToken`, the `wrap()` transaction can be done successfully. This can cause a `feeToken` from `distributeReward()` function to be minted without transferring any token to the contract.

WSafeToken.sol

```
function wrap(uint256 amount) public returns (uint256 totalReceived) {
    uint256 balanceBefore = safeToken.balanceOf(address(this));
    uint256 wrapRatio = getWrapRatio();
    safeToken.transferFrom(msg.sender, address(this), amount);

    uint256 safeBalance = safeToken.balanceOf(address(this));

    totalReceived = safeBalance - balanceBefore;
    totalReceived = totalReceived * 1e9 / wrapRatio;
    uint256 wrapFee = totalReceived * wrapFeeRate / 1000;

```

```
    totalReceived -= wrapFee;

    distributeReward(amount);

    _mint(msg.sender, totalReceived);
    _mint(devAddr, wrapFee);

    emit Wrap(msg.sender, amount, wrapFee, totalReceived);
}
```

The `safeToken.transfer()` and `safeToken.transferFrom()` functions of `WSafeToken` are affected.

5.7.2. Remediation

Inspex suggests replacing the `transfer()` and `transferFrom()` functions of the untrusted tokens from `IERC20` with `safeTransfer()` and `safeTransferFrom()` functions from OpenZeppelin's `SafeERC20` contract, for example:

WSafeToken.sol

```
// SPDX-License-Identifier: BUSL-1.1
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/token/ERC20/SafeERC20.sol";
import "@openzeppelin/contracts/access/Ownable.sol";

interface IMintableERC20 is IERC20 {
    function mint(address _to, uint256 _amount) external;
}

// Wrapped Safe that can be traded without any fee
// TOEDIT: Name and Symbol
contract WSafeToken is ERC20('Wrapped SafeMoon', 'wSAFEMOON'), Ownable {
    using SafeERC20 for IERC20;
    IERC20 public immutable safeToken;
    IMintableERC20 public immutable feeToken;
    address public devAddr;
```

WSafeToken.sol

```
function wrap(uint256 amount) public returns (uint256 totalReceived) {
    uint256 balanceBefore = safeToken.balanceOf(address(this));
    uint256 wrapRatio = getWrapRatio();
    safeToken.safeTransferFrom(msg.sender, address(this), amount);

    uint256 safeBalance = safeToken.balanceOf(address(this));

```

```
    totalReceived = safeBalance - balanceBefore;
    totalReceived = totalReceived * 1e9 / wrapRatio;
    uint256 wrapFee = totalReceived * wrapFeeRate / 1000;

    totalReceived -= wrapFee;

    distributeReward(amount);

    _mint(msg.sender, totalReceived);
    _mint(devAddr, wrapFee);

    emit Wrap(msg.sender, amount, wrapFee, totalReceived);
}
```

WSafeToken.sol

```
function unwrap(uint256 amount) public {
    uint256 unwrapFee = amount * unwrapFeeRate / 1000;
    uint256 outputAmount = getWrapRatio() * (amount - unwrapFee) / 1e9;

    _burn(msg.sender, amount);
    _mint(devAddr, unwrapFee);

    safeToken.safeTransfer(msg.sender, outputAmount);

    emit Unwrap(msg.sender, amount, unwrapFee, outputAmount);
}
```

Please note that the `WSafeToken` contracts already deployed for $MMP, $SAFEMOON, and $SAFEMARS are not affected by this issue.

5.8. Design Flaw in massUpdatePool() Function

ID	IDX-008
Target	Cafeteria
Category	General Smart Contract Vulnerability
CWE	CWE-400: Uncontrolled Resource Consumption
Risk	**Severity:** Low **Impact:** Medium The `massUpdatePool()` 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 massUpdatePool() is eventually unusable.
Status	**Acknowledged** Foodcourt Finance team has acknowledged this issue. The function is used by the owner only, and the team will accept the high gas usage on the calling of this function. Moreover, if the pools can be removed, there could be complications with the frontend integration. Furthermore, Foodcourt Finance team will deploy new `Cafeteria` contracts for more pools.

5.8.1. Description

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

Cafeteria.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.

5.8.2. Remediation

Inspex suggests making the contract capable of removing unnecessary or ended pools to reduce the loop round in the `massUpdatePool()` function, for example:

```
require(_pid < poolInfo.length);
poolInfo[_pid] = poolInfo[poolInfo.length-1];
poolInfo.length--;
```

5.9. Improper Condition Checking in emergencyWithdraw() Function

ID	IDX-009
Target	Cafeteria
Category	Advanced Smart Contract Vulnerability
CWE	CWE-755: Improper Handling of Exceptional Conditions
Risk	**Severity:** Low **Impact:** Low Users can withdraw staked tokens from `withdrawFee100` pools. However, all rewards will be revoked. **Likelihood:** Medium It is likely that users will be able to execute the `emergencyWithdraw()` function. However, there is no benefit for the attacker, resulting in low motivation for the attack.
Status	Resolved Foodcourt Finance team has resolved this issue as suggested in commit 9da4375e0fe1cf866fbc863ee375d018b288d5e8.

5.9.1. Description

For the `withdrawFee100` pool, users cannot withdraw the staked tokens from pools as shown below in line 228.

Cafeteria.sol

```
224 function withdraw(uint256 _pid, uint256 _amount) public nonReentrant {
225     PoolInfo storage pool = poolInfo[_pid];
226     UserInfo storage user = userInfo[_pid][msg.sender];
227     require(user.amount >= _amount, "invalid amount");
228     require(!pool.withdrawFee100, "harvest only");
229     updatePool(_pid);
230     uint256 pending =
    user.amount.mul(pool.accCouponPerShare).div(1e12).sub(user.rewardDebt) +
    user.lockedReward;
231     if (pending > 0) {
232         safeCouponTransfer(msg.sender, pending);
233         user.lockedReward = 0;
234     }
235     if (_amount > 0) {
236         user.amount = user.amount.sub(_amount);
237         pool.lpToken.safeTransfer(address(msg.sender), _amount);
238     }
239     user.rewardDebt = user.amount.mul(pool.accCouponPerShare).div(1e12);
```

```
    emit Withdraw(msg.sender, _pid, _amount);
}
```

By executing `emergencyWithdraw()` function, the `withdrawFee100` condition check is missing. Thus, users are still able to withdraw their tokens staked. Nevertheless, all rewards are revoked.

Cafeteria.sol

```
function emergencyWithdraw(uint256 _pid) public nonReentrant {
    PoolInfo storage pool = poolInfo[_pid];
    UserInfo storage user = userInfo[_pid][msg.sender];
    uint256 amount = user.amount;
    user.amount = 0;
    user.rewardDebt = 0;
    user.lockedReward = 0;
    pool.lpToken.safeTransfer(address(msg.sender), amount);
    emit EmergencyWithdraw(msg.sender, _pid, amount);
}
```

As a result, by performing this attack, the reward tokens will be left in the contract, and this can also affect the Improper Reward Calculation (Same Token) issue.

5.9.2. Remediation

Inspex suggests checking the `withdrawFee100` pool attribute before allowing users to withdraw their tokens staked as shown in the following example:

Cafeteria.sol

```
function emergencyWithdraw(uint256 _pid) public nonReentrant {
    PoolInfo storage pool = poolInfo[_pid];
    require(!pool.withdrawFee100, "harvest only");
    UserInfo storage user = userInfo[_pid][msg.sender];
    uint256 amount = user.amount;
    user.amount = 0;
    user.rewardDebt = 0;
    user.lockedReward = 0;
    pool.lpToken.safeTransfer(address(msg.sender), amount);
    emit EmergencyWithdraw(msg.sender, _pid, amount);
}
```

5.10. Addition of Pool With Duplicated ibToken

ID	IDX-010
Target	Cafeteria
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Low **Impact:** Medium Adding pools with duplicated ibToken can cause the reward calculation of those pools to be incorrect. **Likelihood:** Low It is unlikely that pools with duplicated token will be added unintentionally. Also, there is no benefit for doing so, resulting in low motivation for the attack.
Status	Resolved Foodcourt finance team has resolved this issue in commit 9da4375e0fe1cf866fbc863ee375d018b288d5e8 by preventing the `poolExistence` map from being set back to false.

5.10.1. Description

The `Cafeteria` contract is used to distribute $COUPON to the users who are staking specific tokens in each pool. New pools can be added by the owner of the contract using the `add()` function.

Cafeteria.sol

```
function add(uint256 _allocPoint, IBEP20 _lpToken, uint16 _depositFeeBP, bool
_withdrawFee100, bool _withUpdate) public onlyOwner nonDuplicated(_lpToken) {
    require(_depositFeeBP <= 10000, "invalid deposit fee basis points");
    if (_withUpdate) {
        massUpdatePools();
    }
    uint256 lastRewardBlock = block.number > startBlock ? block.number :
startBlock;
    totalAllocPoint = totalAllocPoint.add(_allocPoint);
    poolExistence[_lpToken] = true;
    poolInfo.push(PoolInfo({
        lpToken: _lpToken,
        allocPoint: _allocPoint,
        lastRewardBlock: lastRewardBlock,
        accCouponPerShare: 0,
        depositFeeBP: _depositFeeBP,
        withdrawFee100: _withdrawFee100
```

```
    }));
}
```

The add() function has the nonDuplicated modifier to prevent duplicated lpToken from being added. This function checks the boolean value from poolExistence mapping.

Cafeteria.sol

```
modifier nonDuplicated(IBEP20 _lpToken) {
    require(poolExistence[_lpToken] == false, "duplicated");
    _;
}
```

The value of an lpToken entry in poolExistence can be set to false at line 138 in the set() function by setting the allocPoint of the pool to 0.

Cafeteria.sol

```
125 function set(uint256 _pid, uint256 _allocPoint, uint16 _depositFeeBP, bool
    _withUpdate) public onlyOwner {
126     require(_depositFeeBP <= 10000, "invalid deposit fee basis points");
127 
128     if (_withUpdate) {
129         massUpdatePools();
130     }
131 
132     totalAllocPoint =
    totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint);
133     poolInfo[_pid].allocPoint = _allocPoint;
134     poolInfo[_pid].depositFeeBP = _depositFeeBP;
135 
136     // For emergency fix of 100% withdrawal fee
137     if (_allocPoint == 0) {
138         poolExistence[poolInfo[_pid].lpToken] = false;
139     } else {
140         poolExistence[poolInfo[_pid].lpToken] = true;
141     }
142 }
```

Therefore, if the allocPoint of any pool is set to 0, a new pool with the same lpToken can be added. This can cause a miscalculation if the lpToken of the old pool is not completely withdrawn, since the balance of the lpToken is used to calculate the amount of reward per share in the updatePool() function at line 177 and 186.

Cafeteria.sol

```
172 function updatePool(uint256 _pid) public {
173     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 couponReward =
multiplier.mul(couponPerBlock).mul(pool.allocPoint).div(totalAllocPoint);
    coupon.mint(devaddr, couponReward.div(10));
    coupon.mint(address(this), couponReward);
    pool.accCouponPerShare =
pool.accCouponPerShare.add(couponReward.mul(1e12).div(lpSupply));
    pool.lastRewardBlock = block.number;
}
```

5.10.2. Remediation

Inspex suggests checking the balance of the `lpToken` in the pool before setting the value in `poolExistence` mapping to false in `set()` function as follows:

Cafeteria.sol

```
function set(uint256 _pid, uint256 _allocPoint, uint16 _depositFeeBP, bool
_withUpdate) public onlyOwner {
    require(_depositFeeBP <= 10000, "invalid deposit fee basis points");

    if (_withUpdate) {
        massUpdatePools();
    }

    totalAllocPoint =
totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint);
    poolInfo[_pid].allocPoint = _allocPoint;
    poolInfo[_pid].depositFeeBP = _depositFeeBP;

    // For emergency fix of 100% withdrawal fee
    if (_allocPoint == 0) {
        if (poolInfo[_pid].lpToken.balanceOf(address(this)) == 0) {
            poolExistence[poolInfo[_pid].lpToken] = false;
        }
    } else {
        poolExistence[poolInfo[_pid].lpToken] = true;
    }
}
```

And it is also recommended to check the existence of a pool before allowing a deposit, for example:

Cafeteria.sol

```
function deposit(uint256 _pid, uint256 _amount) public nonReentrant {
    PoolInfo storage pool = poolInfo[_pid];
    require(poolExistence[pool.lpToken], "Pool does not exist");
    UserInfo storage user = userInfo[_pid][msg.sender];
    updatePool(_pid);

    uint256 pending =
user.amount.mul(pool.accCouponPerShare).div(1e12).sub(user.rewardDebt) +
user.lockedReward;
    if (pending > 0) {
        if (_amount == 0 || !pool.withdrawFee100) {
            safeCouponTransfer(msg.sender, pending);
            user.lockedReward = 0;
        } else {
            user.lockedReward = pending;
        }
    }
```

5.11. Improper Function Visibility

ID	IDX-011
Target	Cafeteria CouponToken FoodcourtRouter Mintable WSafeToken RSafeToken
Category	Smart Contract Best Practice
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	**Severity:** Info **Impact:** None **Likelihood:** None
Status	Resolved * Foodcourt finance team has resolved this issue as suggested in commit 9da4375e0fe1cf866fbc863ee375d018b288d5e8, except for the `FoodcourtRouter` contract to make it similar to `PancakeRouter` as much as possible.

5.11.1. 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 `add()` function of the Cafeteria is set to public and it is never called from any internal function.

Cafeteria.sol

```
// Add a new lp to the pool. Can only be called by the owner.
function add(uint256 _allocPoint, IBEP20 _lpToken, uint16 _depositFeeBP, bool
_withdrawFee100, bool _withUpdate) public onlyOwner nonDuplicated(_lpToken) {
```

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

Target	Function
Cafeteria.sol (L: 106)	add()
Cafeteria.sol (L: 125)	set()

Cafeteria.sol (L: 191)	deposit()
Cafeteria.sol (L: 224)	withdraw()
Cafeteria.sol (L: 244)	emergencyWithdraw()
Cafeteria.sol (L: 268)	dev()
Cafeteria.sol (L: 274)	setFeeAddress()
Cafeteria.sol (L: 281)	updateEmissionRate()
CouponToken.sol (L: 16)	mint()
FoodcourtRouter.sol (L: 790)	quote()
FoodcourtRouter.sol (L: 794)	getAmountOut()
FoodcourtRouter.sol (L: 804)	getAmountIn()
FoodcourtRouter.sol (L: 814)	getAmountsOut()
FoodcourtRouter.sol (L: 824)	getAmountsIn()
Mintable.sol (L:25)	setMinterTimelock()
Mintable.sol (L:31)	setAllowMinting()
Mintable.sol (L:43)	setDailyMintLimit()
WSafeToken.sol (L: 39)	setSafeFeeRate()
WSafeToken.sol (L: 47)	setWrapFeeRate()
WSafeToken.sol (L: 55)	setUnwrapFeeRate()
WSafeToken.sol (L: 63)	setDev()
WSafeToken.sol (L: 77)	wrap()
WSafeToken.sol (L: 99)	unwrap()
RSafeToken.sol (L: 11)	getRealTotalSupply()
RSafeToken.sol (L: 16)	mint()

5.11.2. Remediation

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

Cafeteria.sol

```
// Add a new lp to the pool. Can only be called by the owner.
function add(uint256 _allocPoint, IBEP20 _lpToken, uint16 _depositFeeBP, bool _withdrawFee100, bool _withUpdate) external onlyOwner nonDuplicated(_lpToken) {
```

5.12. Inexplicit Solidity Compiler Version

ID	IDX-012
Target	CouponToken RSafeToken WSafeToken
Category	Smart Contract Best Practice
CWE	CWE-1104: Use of Unmaintained Third Party Components
Risk	**Severity:** Info **Impact:** None **Likelihood:** None
Status	Resolved Foodcourt finance team has resolved this issue as suggested in commit 9da4375e0fe1cf866fbc863ee375d018b288d5e8.

5.12.1. Description

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

CouponToken.sol

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;
```

The following table contains all targets which the inexplicit compiler version is declared.

Target	Version
CouponToken.sol	^0.8.0
RSafeToken.sol	^0.8.0
WSafeToken.sol	^0.8.0

5.12.2. Remediation

Inspex suggests fixing the solidity compiler of `CouponToken`, `RSafeToken`, and `WSafeToken` to the latest stable version. During the audit activity, the latest stable version of Solidity compiler in major 0.8 is v0.8.6.

CouponToken.sol

```
// SPDX-License-Identifier: MIT

pragma solidity 0.8.6;
```

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