DripBar, MasterBarista, LatteV2 & BeanBagV2

Smart Contract Audit Report
Prepared for LatteSwap

Date Issued:	Oct 11, 2021
Project ID:	AUDIT2021031
Version:	v1.0
Confidentiality Level:	Public

inspex

CYBERSECURITY PROFESSIONAL SERVICE

Report Information

Project ID	AUDIT2021031
Version	v1.0
Client	LatteSwap
Project	DripBar, MasterBarista, LatteV2 & BeanBagV2
Auditor(s)	Weerawat Pawanawiwat Patipon Suwanbol
Author	Patipon Suwanbol
Reviewer	Weerawat Pawanawiwat
Confidentiality Level	Public

Version History

Version	Date	Description	Author(s)
1.0	Oct 11, 2021	Full report	Patipon Suwanbol

Contact Information

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

Table of Contents

1. Executive Summary

As requested by LatteSwap, Inspex team conducted an audit to verify the security posture of the DripBar, MasterBarista, LatteV2 & BeanBagV2 smart contracts between Oct 1, 2021 and Oct 5, 2021. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of DripBar, MasterBarista, LatteV2 & BeanBagV2 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 high, 1 medium, 3 low, 1 very low, and 2 info-severity issues. With the project team's prompt response in resolving the issues found by Inspex, all issues were resolved or mitigated in the reassessment. Therefore, Inspex trusts that DripBar, MasterBarista, LatteV2 & BeanBagV2 smart contracts have high-level protections in place to be safe from most attacks.

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

LatteSwap is a decentralized exchange with integrated NFT functionalities operating on the Binance Smart Chain (BSC). It is a one-stop-shop for traders, yield farmers, and NFT collectors across the Blockchain ecosystem.

MasterBarista, LatteV2, and BeanBagV2 are the new version of the previous LatteSwap contracts which have been improved greatly, allowing LatteSwap to provide additional utilities for the users and further support integration with other platforms. With the new deployment of the DripBar contract, users can increase their earning from $LATTE by staking $BEAN to the DripBar to acquire new trending tokens specially selected by LatteSwap.

Scope Information:

Project Name	DripBar, MasterBarista, LatteV2 & BeanBagV2
Website	https://app.latteswap.com/
Smart Contract Type	Ethereum Smart Contract
Chain	Binance Smart Chain
Programming Language	Solidity

Audit Information:

Audit Method	Whitebox
Audit Date	Oct 1, 2021 - Oct 5, 2021
Reassessment Date	Oct 11, 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:

Contract	Location (URL)	Commit
DripBar	https://github.com/latteswap-official/latteswap-contract/tree/820704bf8a/contracts/farm/DripBar.sol	820704bf8aaeb14ab7d366ff3bf8c0b56918b6fa
MasterBarista	https://github.com/latteswap-official/latteswap-contract/tree/0fbab1e0ca/contracts/farm/MasterBarista.sol	0fbab1e0ca07bbe34dcd69988d2347166e4c690e
LATTEV2	https://github.com/latteswap-official/latteswap-contract/tree/0fbab1e0ca/contracts/farm/LATTEV2.sol	0fbab1e0ca07bbe34dcd69988d2347166e4c690e
BeanBagV2	https://github.com/latteswap-official/latteswap-contract/tree/0fbab1e0ca/contracts/farm/BeanBagV2.sol	0fbab1e0ca07bbe34dcd69988d2347166e4c690e

Reassessment: (Commit: 3cea9238c2a69e129562e6a7b3f45e89b6540c1e)

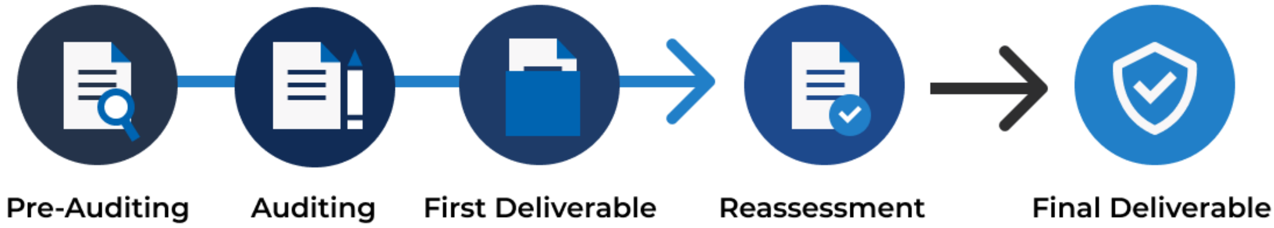
Contract	Location (URL)
DripBar	https://github.com/latteswap-official/latteswap-contract/tree/3cea9238c2/contracts/farm/DripBar.sol
MasterBarista	https://github.com/latteswap-official/latteswap-contract/tree/3cea9238c2/contracts/farm/MasterBarista.sol
LATTEV2	https://github.com/latteswap-official/latteswap-contract/tree/3cea9238c2/contracts/farm/LATTEV2.sol
BeanBagV2	https://github.com/latteswap-official/latteswap-contract/tree/3cea9238c2/contracts/farm/BeanBagV2.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
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

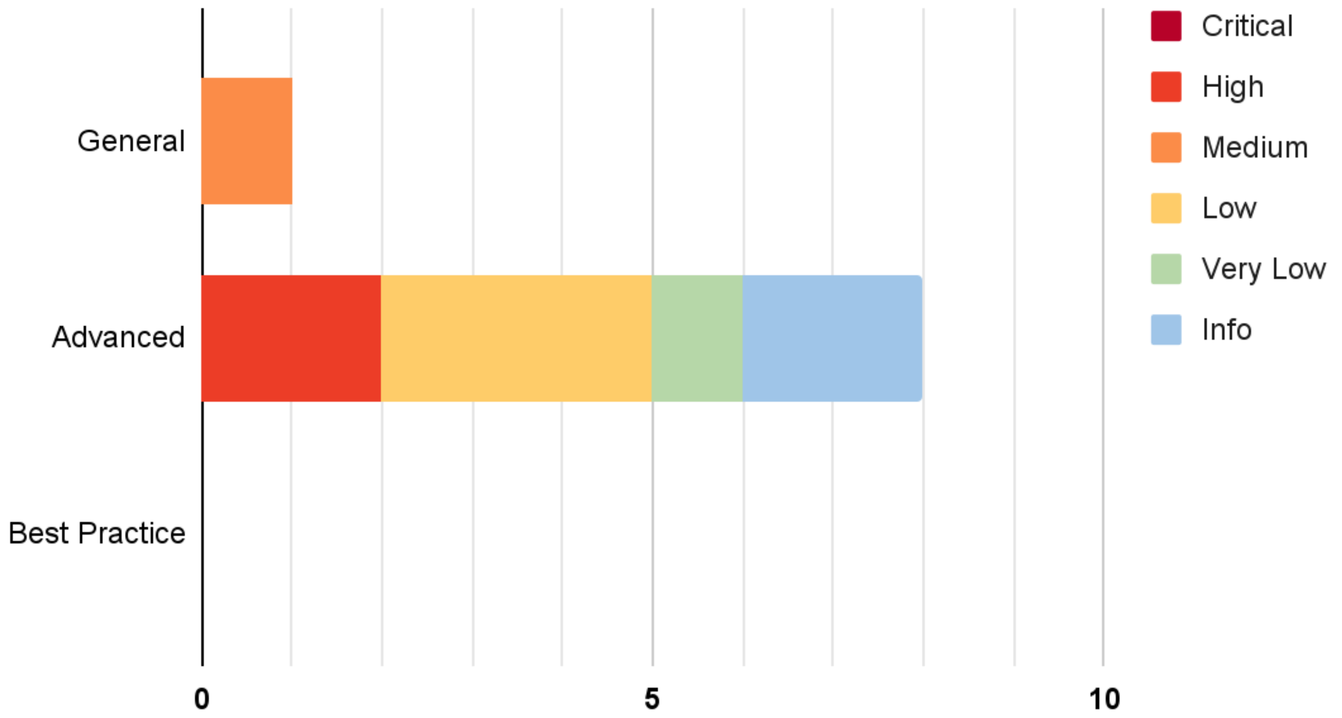
Both likelihood and impact can be categorized into three levels: **Low**, **Medium**, and **High**.

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



The statuses of the issues are defined as follows:

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

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

ID	Title	Category	Severity	Status
IDX-001	Use of Upgradable Contract Design	Advanced	High	Resolved *
IDX-002	Unrestricted Addition of $LATTE Minter	Advanced	High	Resolved *
IDX-003	Centralized Control of State Variable	General	Medium	Resolved *
IDX-004	Unchecked Repeated Migration	Advanced	Low	Resolved *
IDX-005	Denial of Service on Cap Exceeding	Advanced	Low	Resolved *
IDX-006	Unsupported Design for Deflationary Reward Token in DripBar	Advanced	Low	Resolved *
IDX-007	Insufficient Logging for Privileged Functions	Advanced	Very Low	Resolved
IDX-008	Unsupported Design for Deflationary Staking Token in DripBar	Advanced	Info	Resolved *
IDX-009	Unsupported Design for Deflationary Staking Token in MasterBarista	Advanced	Info	Resolved *

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

5. Detailed Findings Information

5.1 Use of Upgradable Contract Design

ID	IDX-001
Target	DripBar MasterBarista BeanBagV2
Category	Advanced Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity:** High **Impact:** High The logic of affected contracts can be arbitrarily changed. This allows the proxy owner to perform malicious actions e.g., stealing the user funds anytime they want. **Likelihood:** Medium This action can be performed by the proxy owner without any restriction.
Status	Resolved * LatteSwap team has implemented the timelock mechanism to the `MasterBarista` and `BeanBagV2` contracts. The users will be able to monitor the timelock for the upgrade of the contract and act accordingly if it is being misused. For the `MasterBarista` contract at `0xbCeE0d15a4402C9Cc894D52cc5E9982F60C463d6`, LatteSwap team has already transferred the contract ownership to the `Timelock` contract at `0x813879b5556b73c02a139e0340a33239c047957d`. However, at the time of the reassessment, the delay of the `Timelock` contract is 6 hours, which is insufficient according to Inspex's verification standard. LatteSwap team has confirmed that they will increase the delay of the `Timelock` contract to 24 hours. Hence, it is suggested that the users should verify that the delay of the `Timelock` contract is at least 24 hours before using it. For the `DripBar` contract, at the time of the reassessment, the contract has not been deployed yet, so the use of timelock is not confirmed. For the platform users, please verify that the timelock is properly deployed before using this platform. For the `BeanBagV2` contract, the contract owner is `ProxyAdmin` (`0x02af4337792a44afb4005d57c36f9c3bea6209bb`) whose owner is the `Timelock` contract (`0x813879b5556b73c02a139e0340a33239c047957d`). Since the delay of the `Timelock` contract at the time of the reassessment is 6 hours, which is insufficient according to Inspex's verification standard, LatteSwap team has confirmed that they will increase the delay of the `Timelock` contract to 24 hours. Hence, again, it is suggested that the users should verify that the delay of the `Timelock` contract is at least 24 hours before using it.

5.1.1 Description

Smart contracts are designed to be used as agreements that cannot be changed forever. When a smart contract is upgraded, the agreement can be changed from what was previously agreed upon.

As these smart contracts are upgradable, the logic of them can be modified by the owner anytime, making the smart contracts untrustworthy.

5.1.2 Remediation

Inspex suggests deploying the contracts without the proxy pattern or any solution that can make smart contracts upgradable.

However, if the upgradability 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 is notified of the potential changes being done on the smart contracts.

5.2 Unrestricted Addition of $LATTE Minter

ID	IDX-002
Target	MasterBarista
Category	Advanced Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity:** High **Impact:** High The contract owner can mint an arbitrary amount of $LATTE until the total supply cap (`cap`) is filled, which can then be dumped in the market to gain profit and lower the price of $LATTE, resulting in monetary loss for the token holders. **Likelihood:** Medium Only the contract owner can perform this attack; however, there is no restriction to prevent the owner from doing it.
Status	Resolved * LatteSwap team has already transferred the contract owner of the `MasterBarista` contract at `0xbCeE0d15a4402C9Cc894D52cc5E9982F60C463d6` to the `Timelock` contract at `0x813879b5556b73c02a139e0340a33239c047957d`. However, at the time of the reassessment, the delay of the `Timelock` contract is 6 hours, which is insufficient according to Inspex's verification standard. LatteSwap team has confirmed that they will increase the delay of the `Timelock` contract to 24 hours. Hence, it is suggested that the users should verify that the delay of the `Timelock` contract is at least 24 hours before using it.

5.2.1 Description

In the `MasterBarista` contract, the `mintExtraReward()` function can be called to mint extra $LATTE reward from the `Booster` contract.

MasterBarista.sol

```
/// @dev This is a function for mining an extra amount of latte, should be called only by stake token caller contract (boosting purposed)
/// @param _stakeToken a stake token address for validating a msg sender
/// @param _amount amount to be minted
function mintExtraReward(
    address _stakeToken,
    address _to,
    uint256 _amount,
    uint256 _lastRewardBlock
  ) external override onlyStakeTokenCallerContract(_stakeToken) {
```

```
    uint256 multiplierBps = _getBonusMultiplierProportionBps(_lastRewardBlock, block.number);
    uint256 toBeLockedNum = _amount.mul(multiplierBps).mul(bonusLockUpBps);
    _assignActiveToken();

    // mint & lock(if any) an extra reward
    activeLatte.mint(_to, _amount);
    activeLatte.lock(_to, toBeLockedNum.div(1e8));
    activeLatte.mint(devAddr, _amount.mul(devFeeBps).div(1e4));
    activeLatte.lock(devAddr, (toBeLockedNum.mul(devFeeBps)).div(1e12));

    emit MintExtraReward(_msgSender(), _stakeToken, _to, _amount);
}
```

The caller of the function is checked in the `onlyStakeTokenCallerContract` modifier, allowing only the addresses in `stakeTokenCallerContracts` list to call this function.

MasterBarista.sol

```
/// @dev only stake token caller contract can continue the execution (stakeTokenCaller must be a funder contract)
/// @param _stakeToken a stakeToken to be validated
modifier onlyStakeTokenCallerContract(address _stakeToken) {
    require(
      stakeTokenCallerContracts[_stakeToken].has(_msgSender()),
      "MasterBarista::onlyStakeTokenCallerContract: bad caller"
    );
    _;
}
```

However, the contract owner can use the `addStakeTokenCallerContract()` function to add any address to the `stakeTokenCallerContracts` list.

MasterBarista.sol

```
/// @notice Setter function for adding stake token contract caller
/// @param _stakeToken a pool for adding a corresponding stake token contract caller
/// @param _caller a stake token contract caller
function addStakeTokenCallerContract(address _stakeToken, address _caller) external onlyOwner {
    require(
      stakeTokenCallerAllowancePool[_stakeToken],
      "MasterBarista::addStakeTokenCallerContract: the pool doesn't allow a contract caller"
    );
    LinkList.List storage list = stakeTokenCallerContracts[_stakeToken];
    if (list.getNextOf(LinkList.start) == LinkList.empty) {
```

```
        list.init();
    }
    list.add(_caller);
    emit AddStakeTokenCallerContract(_stakeToken, _caller);
}
```

This means that the contract owner can add the owner's wallet address to the list and freely use the `mintExtraReward()` function to mint an arbitrary amount of $LATTE.

5.2.2 Remediation

In the ideal case, the contract owner should not be able to mint $LATTE freely.

Since with the current design, this functionality is needed for another contract to function properly, Inspex suggests mitigating this issue by implementing a timelock mechanism with a sufficient length of time to delay the use of the `addStakeTokenCallerContract()` function. This allows the platform users to monitor the timelock and be notified of the potential changes being done on the smart contracts.

5.3 Centralized Control of State Variable

ID	IDX-003
Target	DripBar MasterBarista LATTEV2 BeanBagV2
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 There is nothing to restrict the changes from being done; however, these actions can only be performed by the contract owner.
Status	Resolved * LatteSwap team has implemented the timelock mechanism to the `MasterBarista`, `LATTEV2`, and `BeanBagV2` contracts. The users will be able to monitor the timelock for the upgrade of the contract and act accordingly if it is being misused. For the `MasterBarista` contract at `0x813879B5556B73c02A139e0340A33239C047957D`, LatteSwap team has already transferred the contract owner to the `Timelock` contract at `0x813879b5556b73c02a139e0340a33239c047957d`. However, at the time of the reassessment, the delay of the `Timelock` contract is 6 hours, which is insufficient according to Inspex's verification standard. LatteSwap team has confirmed that they will increase the delay of the `Timelock` contract to 24 hours. Hence, it is suggested that the users should verify that the delay of the `Timelock` contract is at least 24 hours before using it. For the `LATTEV2` contract at `0xa269A9942086f5F87930499dC8317ccC9dF2b6CB` which inherits from the `AccessControl` contract, the current address in `admin_role` is LatteSwap: Deployer at `0xE626fC6D9f4F1FAA17a157FB854d27fC55327283`. It is suggested that the users should verify that the contract has already given the `admin_role` to the `Timelock` contract and removed the LatteSwap: Deployer from `admin_role` respectively. For the `DripBar` contract, at the time of the reassessment, the contract has not been deployed yet, so the use of timelock is not confirmed. For the platform users, please verify that the timelock is properly deployed before using this platform. For the `BeanBagV2` contract, the contract owner is `ProxyAdmin`

	(`0x02af4337792a44afb4005d57c36f9c3bea6209bb`) whose owner is the `Timelock` contract (`0x813879b5556b73c02a139e0340a33239c047957d`). Since the delay of the `Timelock` contract at the time of the reassessment is 6 hours, which is insufficient according to Inspex's verification standard, LatteSwap team has confirmed that they will increase the delay of the `Timelock` contract to 24 hours. Hence, again, it is suggested that the users should verify that the delay of the `Timelock` contract is at least 24 hours before using it.

5.3.1 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, there is 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
DripBar (L: 71)	DripBar	setRewardHolder()	onlyOwner
DripBar (L: 77)	DripBar	setRewardInfoLimit()	onlyOwner
DripBar (L: 83)	DripBar	addCampaignInfo()	onlyOwner
DripBar (L: 97)	DripBar	addRewardInfo()	onlyOwner
DripBar (L: 306)	DripBar	emergencyRewardWithdraw()	onlyOwner
MasterBarista (L: 179)	MasterBarista	setStakeTokenCallerAllowancePool()	onlyOwner
MasterBarista (L: 188)	MasterBarista	addStakeTokenCallerContract()	onlyOwner
MasterBarista (L: 204)	MasterBarista	removeStakeTokenCallerContract()	onlyOwner
MasterBarista (L: 224)	MasterBarista	setLattePerBlock()	onlyOwner
MasterBarista (L: 233)	MasterBarista	setPoolAllocBps()	onlyOwner
MasterBarista (L: 261)	MasterBarista	setBonus()	onlyOwner
MasterBarista (L: 282)	MasterBarista	addPool()	onlyOwner
MasterBarista (L: 310)	MasterBarista	setPool()	onlyOwner

MasterBarista (L: 332)	MasterBarista	removePool()	onlyOwner
MasterBarista (L: 840)	MasterBarista	migrate()	onlyOwner
@openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol (L: 60)	DripBar, MasterBarista, BeanBagV2	renounceOwnership()	onlyOwner
@openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol (L: 69)	DripBar, MasterBarista, BeanBagV2	transferOwnership()	onlyOwner
@openzeppelin/contracts/access/Ownable.sol (L: 54)	LATTEV2	renounceOwnership()	onlyOwner
@openzeppelin/contracts/access/Ownable.sol (L: 63)	LATTEV2	transferOwnership()	onlyOwner
@openzeppelin/contracts/access/AccessControl sol (L: 135)	LATTEV2	grantRole()	-
@openzeppelin/contracts/access/AccessControl.sol (L: 150)	LATTEV2	revokeRole()	-
@openzeppelin/contracts/access/AccessControl.sol (L: 170)	LATTEV2	renounceRole()	-

Please note that the `OwnableUpgradeable`, `Ownable`, and `AccessControl` contracts are inherited from OpenZeppelin's library by the affected contracts.

5.3.2 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, e.g., 24 hours

5.4 Unchecked Repeated Migration

ID	IDX-004
Target	MasterBarista
Category	Advanced Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity:** Low **Impact:** Medium The $LATTEV2 will not be migrated to the newer contract, causing the new BeanBagV2 to have insufficient reward token for the users. This causes monetary loss to the users and reputation damage to the platform. **Likelihood:** Low The migration is only done by the contract owner, and there is no benefit for the owner in executing the `migrate()` function multiple times, resulting in low motivation for this action.
Status	Resolved * LatteSwap team has confirmed that the `migrate()` function will be removed from the `MasterBarista` contract once the mitigation process of LATTEV2 is completed. This means that the `migrate()` function will be used only once.

5.4.1 Description

In the `MasterBarista` contract, the `migrate()` function is implemented to migrate the reward token from $LATTE to $LATTEV2 to prepare for new features. The $LATTE minted as the users' reward is transferred from the BeanBag contract to `MasterBarista` at line 849, and then the `redeem()` function of $LATTEV2 is called to convert $LATTE to $LATTEV2 at line 851. The newly acquired $LATTEV2 is then transferred to the new BeanBagV2 contract.

MasterBarista.sol

```
839 /// @notice migrate latteV1 -> latteV2 and beanV1 -> beanV2
840 function migrate(ILATTEV2 _latteV2, IBeanBag _beanV2) external onlyOwner {
841     massUpdatePools();
842     uint256 _amount = latte.balanceOf(address(bean));
843 
844     activeLatte = ILATTE(address(_latteV2));
845     activeBean = _beanV2;
846     latteV2 = _latteV2;
847     beanV2 = _beanV2;
848 
849     bean.safeLatteTransfer(address(this), _amount);
```

```
    latte.approve(address(_latteV2), uint256(-1));
    _latteV2.redeem(_amount);
    _latteV2.transfer(address(beanV2), _amount);
    latte.approve(address(_latteV2), 0);

    emit Migrate(_amount);
}
```

However, there is no checking whether the migration has been done or not. If the `migrate()` function is called multiple times, the address of `latteV2` and `beanV2` can be changed; however, the reward in the original `beanV2` address will not be transferred to the newer `beanV2`, since the transfer is done from `bean` only, not from `beanV2`.

This can cause the new `beanV2` address to have insufficient reward token for the users, and the users will be unable to claim their reward.

5.4.2 Remediation

Inspex suggests limiting the `migrate()` function to be usable for only one single time. For example:

MasterBarista.sol

```
/// @notice migrate latteV1 -> latteV2 and beanV1 -> beanV2
function migrate(ILATTEV2 _latteV2, IBeanBag _beanV2) external onlyOwner {
    require(address(latteV2) == address(0), "MasterBarista::migrate::already
migrated");
    massUpdatePools();
    uint256 _amount = latte.balanceOf(address(bean));

    activeLatte = ILATTE(address(_latteV2));
    activeBean = _beanV2;
    latteV2 = _latteV2;
    beanV2 = _beanV2;

    bean.safeLatteTransfer(address(this), _amount);
    latte.approve(address(_latteV2), uint256(-1));
    _latteV2.redeem(_amount);
    _latteV2.transfer(address(beanV2), _amount);
    latte.approve(address(_latteV2), 0);

    emit Migrate(_amount);
}
```

5.5 Denial of Service on Cap Exceeding

ID	IDX-005
Target	MasterBarista LATTEV2
Category	Advanced Smart Contract Vulnerability
CWE	CWE-755: Improper Handling of Exceptional Conditions
Risk	**Severity:** Low **Impact:** Medium Multiple functions of `MasterBarista` contract will be unusable from the failed token minting, disrupting the availability of the service. The users can withdraw their funds using the `emergencyWithdraw()` function, but the pending reward will be discarded. **Likelihood:** Low The original cap is the max value of `uint256`, which is unlikely to be reached; however, the cap can be adjusted by the governor, so it is possible for the cap to be filled.
Status	Resolved * LatteSwap team has confirmed that the reward distribution will be adjusted properly before $LATTE reaches the total supply cap.

5.5.1 Description

The `mint()` function of `LATTEV2` contract allows the minter to mint new $LATTEV2. The cap is checked at line 107 to make sure that the token minted will not exceed the cap.

LATTEV2.sol

```
103 /// @dev A function to mint LATTE. This will be called by a minter only.
104 /// @param _to The address of the account to get this newly mint LATTE
105 /// @param _amount The amount to be minted
106 function mint(address _to, uint256 _amount) external onlyMinter {
107     require(totalSupply().add(_amount) < cap, "LATTEV2::mint::cap exceeded");
108     _mint(_to, _amount);
109 }
```

The cap of $LATTEV2 is initially the maximum number of `uint256`, but it can be set to a lower value by the governor using the `setCap()` function.

LATTEV2.sol

```
/// @dev Set cap. Cap must lower than previous cap. Only Governor can adjust
/// @param _cap The new cap
function setCap(uint256 _cap) external onlyGovernor {
```

```
    require(_cap < cap, "LATTEV2::setCap::_cap must < cap");
    uint256 _prevCap = cap;
    cap = _cap;
    emit CapChanged(_prevCap, cap);
}
```

The `MasterBarista` contract is assigned to be the minter of $LATTEV2, and the `mint()` function will be called every time the reward of each pool is updated in the `updatePool()` function.

MasterBarista.sol

```
/// @dev Update reward variables of the given pool to be up-to-date.
/// @param _stakeToken The stake token address of the pool to be updated
function updatePool(address _stakeToken) public override {
    PoolInfo storage pool = poolInfo[_stakeToken];
    _assignActiveToken();
    if (block.number <= pool.lastRewardBlock) {
        return;
    }
    uint256 totalStakeToken = IERC20(_stakeToken).balanceOf(address(this));
    if (totalStakeToken == 0) {
        pool.lastRewardBlock = block.number;
        return;
    }
    uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
    uint256 latteReward =
multiplier.mul(lattePerBlock).mul(pool.allocPoint).div(totalAllocPoint);
    activeLatte.mint(devAddr, latteReward.mul(devFeeBps).div(10000));
    activeLatte.mint(address(activeBean), latteReward);
    pool.accLattePerShare =
pool.accLattePerShare.add(latteReward.mul(1e12).div(totalStakeToken));

    // Clear bonus & update accLattePerShareTilBonusEnd.
    if (block.number <= bonusEndBlock) {
        activeLatte.lock(devAddr,
latteReward.mul(bonusLockUpBps).mul(15).div(1000000));
        pool.accLattePerShareTilBonusEnd = pool.accLattePerShare;
    }
    if (block.number > bonusEndBlock && pool.lastRewardBlock < bonusEndBlock) {
        uint256 latteBonusPortion = bonusEndBlock
            .sub(pool.lastRewardBlock)
            .mul(bonusMultiplier)
            .mul(lattePerBlock)
            .mul(pool.allocPoint)
            .div(totalAllocPoint);
        activeLatte.lock(devAddr,
latteBonusPortion.mul(bonusLockUpBps).mul(15).div(1000000));
        pool.accLattePerShareTilBonusEnd =
```

```
pool.accLattePerShareTilBonusEnd.add(
            latteBonusPortion.mul(1e12).div(totalStakeToken)
        );
    }

    pool.lastRewardBlock = block.number;
}
```

If the cap is reduced to a low number, and the allocation point of each pool is not set to 0 before the cap is reached, the execution of `mint()` function in the `updatePool()` will be reverted, causing all functions in the `MasterBarista` that call `updatePool()` to be unusable.

5.5.2 Remediation

Inspex suggests modifying the `mint()` and `updatePool()` functions to handle the case when the minting cap is filled.

5.6 Unsupported Design for Deflationary Reward Token in DripBar

ID	IDX-006
Target	DripBar
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Low **Impact:** Medium Insufficient amount of reward token will be transferred to the `DripBar` contract on the addition of a new reward campaign, causing some users to be unable to claim their reward. This results in monetary loss for the users and reputation loss for the platform. **Likelihood:** Low The reward campaign can only be added by the contract owner, and it is not likely for the owner to start a campaign that rewards deflationary token unknowingly.
Status	Resolved * LatteSwap team has confirmed that the `DripBar` contract will not support the deflationary token as the reward token.

5.6.1 Description

The `DripBar` contract allows the users to do yield farming. The users can select the campaign to deposit the required token to acquire the reward token that they want, which is added through the `addRewardInfo()` function to the campaign.

However, an issue could arise when the reward token of the campaign is a deflationary token (the token that reduces the circulating supply itself when it is transferred).

This means the total reward amount of the campaign registered will be reduced due to the deflationary mechanism in the transfer process, but the `DripBar` contract recognizes it as the full amount as in line 107.

DripBar.sol

```
96  // @notice if the new reward info is added, the reward & its end block will be extended by the newly pushed reward info.
97  function addRewardInfo(uint256 _campaignID, uint256 _endBlock, uint256 _rewardPerBlock) external onlyOwner {
98      RewardInfo[] storage rewardInfo = campaignRewardInfo[_campaignID];
99      CampaignInfo storage campaign = campaignInfo[_campaignID];
100     require(rewardInfo.length < rewardInfoLimit, "DripBar::addRewardInfo::reward info length exceeds the limit");
101     require(rewardInfo.length == 0 || rewardInfo[rewardInfo.length - 1].endBlock >= block.number, "DripBar::addRewardInfo::reward period ended");
```

```
    require(rewardInfo.length == 0 || rewardInfo[rewardInfo.length - 1].endBlock < _endBlock, "DripBar::addRewardInfo::bad new endblock");
    uint256 startBlock = rewardInfo.length == 0 ?  campaign.startBlock : rewardInfo[rewardInfo.length - 1].endBlock;
    uint256 blockRange = _endBlock.sub(startBlock);
    uint256 totalRewards = _rewardPerBlock.mul(blockRange);
    campaign.rewardToken.safeTransferFrom(rewardHolder, address(this), totalRewards);
    campaign.totalRewards = campaign.totalRewards.add(totalRewards);
    rewardInfo.push(RewardInfo({
        endBlock: _endBlock,
        rewardPerBlock: _rewardPerBlock
    }));
    emit AddRewardInfo(_campaignID, rewardInfo.length-1, _endBlock, _rewardPerBlock);
}
```

As a result, the users will be unable to harvest the reward when the reward token amount is insufficient.

5.6.2 Remediation

Inspex suggests modifying the logic of the `addRewardInfo()` function to validate the amount of the token received.

5.7 Insufficient Logging for Privileged Function

ID	IDX-007
Target	DripBar
Category	Advanced Smart Contract Vulnerability
CWE	CWE-778: Insufficient Logging
Risk	**Severity:** Very Low **Impact:** Low Privileged function execution cannot be monitored easily by the users. **Likelihood:** Low It is not likely that the execution of the privileged function will be a malicious action.
Status	Resolved LatteSwap team has resolved this issue as suggested in commit `3cea9238c2a69e129562e6a7b3f45e89b6540c1e` by emitting an event in the privileged function.

5.7.1 Description

Privileged function that is executable by the controlling parties is not logged properly by emitting an event. Without an event, it is not easy for the public to monitor the execution of the privileged function, allowing the controlling parties to perform actions that cause big impacts to the platform.

The owner can withdraw the pool reward through the `emergencyRewardWithdraw()` function, which affects the user's reward, and no event is emitted.

DripBar.sol

```
// @notice Withdraw reward. EMERGENCY ONLY.
function emergencyRewardWithdraw(uint256 _campaignID, uint256 _amount, address
_beneficiary) external onlyOwner nonReentrant {
    CampaignInfo storage campaign = campaignInfo[_campaignID];
    uint256 currentStakingPendingReward = _pendingReward(_campaignID,
campaign.totalStaked, 0);
    require(currentStakingPendingReward.add(_amount) <= campaign.totalRewards,
"DripBar::emergencyRewardWithdraw::not enough reward token");
    campaign.totalRewards = campaign.totalRewards.sub(_amount);
    campaign.rewardToken.safeTransfer(_beneficiary, _amount);
}
```

5.7.2 Remediation

Inspex suggests emitting event for the `emergencyRewardWithdraw()` function, for example:

DripBar.sol

```
// @notice Withdraw reward. EMERGENCY ONLY.
event EmergencyRewardWithdraw(uint256 _campaignID, uint256 _amount, address
_beneficiary);
function emergencyRewardWithdraw(uint256 _campaignID, uint256 _amount, address
_beneficiary) external onlyOwner nonReentrant {
    CampaignInfo storage campaign = campaignInfo[_campaignID];
    uint256 currentStakingPendingReward = _pendingReward(_campaignID,
campaign.totalStaked, 0);
    require(currentStakingPendingReward.add(_amount) <= campaign.totalRewards,
"DripBar::emergencyRewardWithdraw::not enough reward token");
    campaign.totalRewards = campaign.totalRewards.sub(_amount);
    campaign.rewardToken.safeTransfer(_beneficiary, _amount);
    emit EmergencyRewardWithdraw(_campaignID, _amount, _beneficiary);
}
```

5.8 Unsupported Design for Deflationary Staking Token in DripBar

ID	IDX-008
Target	DripBar
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved * LatteSwap team has confirmed that the `DripBar` contract will not support the deflationary token as the staking token.

5.8.1 Description

In the `DripBar` contract, the users can stake their tokens to earn reward tokens. The staking token of each campaign can be a simple token or LP token depending on the campaigns that are specified by the contract owner.

However, an issue could arise when the campaign uses a deflationary token (the token that reduces the circulating supply itself when it is transferred) as the staking token for a campaign.

This means the amount of token that user stakes will be reduced due to the deflationary mechanism in the transfer process, but the `DripBar` contract recognizes it as the full amount as in line 254.

DripBar.sol

```
241 // @notice Stake Staking tokens to DripBar
242 function deposit(uint256 _campaignID, uint256 _amount) external nonReentrant {
243     CampaignInfo storage campaign = campaignInfo[_campaignID];
244     UserInfo storage user = userInfo[_campaignID][msg.sender];
245     _updateCampaign(_campaignID);
246     if (user.amount > 0) {
247         uint256 pending =
user.amount.mul(campaign.accRewardPerShare).div(1e12).sub(user.rewardDebt);
248         if (pending > 0) {
249             campaign.rewardToken.safeTransfer(address(msg.sender), pending);
250         }
251     }
252     if (_amount > 0) {
253         campaign.stakingToken.safeTransferFrom(address(msg.sender),
address(this), _amount);
```

```
        user.amount = user.amount.add(_amount);
        campaign.totalStaked = campaign.totalStaked.add(_amount);
    }
    user.rewardDebt = user.amount.mul(campaign.accRewardPerShare).div(1e12);
    emit Deposit(msg.sender, _amount, _campaignID);
}
```

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 campaign in the `DripBar` contract which uses a deflationary token ($TOKEN) as the staking token. $TOKEN has a 10% burn rate when the token is transferred.

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

Holder	Balance
User A	100

Total $TOKEN of the campaign in the `DripBar` contract: 90

User B stakes 100 $TOKEN to the campaign in the `Dripbar` contract. The `DripBar` contract will receive 90 $TOKEN since $TOKEN has 10% deduction from the deflationary mechanism, in this case 10 $TOKEN.

Holder	Balance
User A	100
User B	100

Total $TOKEN of the campaign in the `DripBar` contract: 180

User B then withdraws 100 $TOKEN from the `DripBar` contract by executing the `withdraw()` function. The `DripBar` contract will validate whether the withdrawn `amount` exceeds the `user.amount` of the campaign or not.

DripBar.sol

```
// @notice Withdraw Staking tokens from STAKING.
function withdraw(uint256 _campaignID, uint256 _amount) external nonReentrant {
    _withdraw(_campaignID, _amount);
}
```

The `withdraw()` function calls the internal function, the `_withdraw()` function.

Dripbar.sol

```
// @notice internal method for withdraw (withdraw and harvest method depend on this method)
function _withdraw(uint256 _campaignID, uint256 _amount) internal {
    CampaignInfo storage campaign = campaignInfo[_campaignID];
    UserInfo storage user = userInfo[_campaignID][msg.sender];
    require(user.amount >= _amount, "DripBar::withdraw::bad withdraw amount");
    _updateCampaign(_campaignID);
    uint256 pending = user.amount.mul(campaign.accRewardPerShare).div(1e12).sub(user.rewardDebt);
    if (pending > 0) {
        campaign.rewardToken.safeTransfer(address(msg.sender), pending);
    }
    if (_amount > 0) {
        user.amount = user.amount.sub(_amount);
        campaign.stakingToken.safeTransfer(address(msg.sender), _amount);
        campaign.totalStaked = campaign.totalStaked.sub(_amount);
    }
    user.rewardDebt = user.amount.mul(campaign.accRewardPerShare).div(1e12);

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

Since User B staked 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 ***`DripBar`*** *contract: 80*

As a result, if User A decides to withdraw 100 $TOKEN, or even 90 $TOKEN, this transaction will be reverted since the balance in the contract is insufficient.

Scenario 2: Reward Calculation Exploit

Assuming that there is a campaign in the `DripBar` contract which uses a deflationary token ($TOKEN) as the staking token. $TOKEN has a 10% burn rate when the token is transferred.

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

User A stakes 100 $TOKEN to the contract, and the `DripBar` 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 `DripBar` contract will then calculate the reward from the `updateCampaign()` function, which will call the internal function (`_updateCampaign()`). During the calculation as in line 229, the reward is affected by the total amount of $TOKEN (`campaign.totalStaked`).

DripBar.sol

```
198 // @notice Update reward variables of the given campaign to be up-to-date.
199 function _updateCampaign(uint256 _campaignID) internal {
200     CampaignInfo storage campaign = campaignInfo[_campaignID];
201     RewardInfo[] memory rewardInfo = campaignRewardInfo[_campaignID];
202     if (block.number <= campaign.lastRewardBlock) {
203         return;
204     }
205     if (campaign.totalStaked == 0) {
206         // if there is no total supply, return and use the campaign's start block as the last reward block
207         // so that ALL reward will be distributed.
208         // however, if the first deposit is out of reward period, last reward block will be its block number
209         // in order to keep the multiplier = 0
210         if (block.number > _endBlockOf(_campaignID, block.number)) {
211             campaign.lastRewardBlock = block.number;
212         }
213         return;
214     }
215     // @dev for each reward info
216     for (uint256 i = 0; i < rewardInfo.length; ++i) {
217         // @dev get multiplier based on current Block and rewardInfo's end block
218         // multiplier will be a range of either (current block - campaign.lastRewardBlock)
219         // or (reward info's endblock - campaign.lastRewardBlock) or 0
220         uint256 multiplier = getMultiplier(campaign.lastRewardBlock, block.number, rewardInfo[i].endBlock);
221         if (multiplier == 0) continue;
222         // @dev if currentBlock exceed end block, use end block as the last reward block
223         // so that for the next iteration, previous endBlock will be used as the last reward block
224         if (block.number > rewardInfo[i].endBlock) {
225             campaign.lastRewardBlock = rewardInfo[i].endBlock;
226         } else {
227             campaign.lastRewardBlock = block.number;
228         }
229         campaign.accRewardPerShare =
```

```
campaign.accRewardPerShare.add(multiplier.mul(rewardInfo[i].rewardPerBlock).mul
(1e12).div(campaign.totalStaked));
    }
}
```

Since the `DripBar` contract registers token 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. This means the value of `campaign.accRewardPerShare` can be increased dramatically by manipulating the total amount of $TOKEN (`campaign.totalStaked`) to be as low as possible.

Hence, 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 `campaign.accRewardPerShare` state to be overly inflated, so the users can claim an exceedingly large amount of reward from the contract.

However, since the staking token of all campaigns is designed to be $BEANV2, which is not a deflationary token, there is no impact for this issue.

5.8.2 Remediation

Inspex suggests modifying the logic of the `deposit()` function to validate the amount of the token received from the user instead of using the value of `amount` parameter directly, for example:

DripBar.sol

```
// @notice Stake Staking tokens to DripBar
function deposit(uint256 _campaignID, uint256 _amount) external nonReentrant {
    CampaignInfo storage campaign = campaignInfo[_campaignID];
    UserInfo storage user = userInfo[_campaignID][msg.sender];
    _updateCampaign(_campaignID);
    uint256 receivedBalance;
    if (user.amount > 0) {
        uint256 pending =
user.amount.mul(campaign.accRewardPerShare).div(1e12).sub(user.rewardDebt);
        if (pending > 0) {
            campaign.rewardToken.safeTransfer(address(msg.sender), pending);
        }
    }
    if (_amount > 0) {
        uint256 balanceBefore = campaign.stakingToken.balanceOf(address(this));
        campaign.stakingToken.safeTransferFrom(address(msg.sender),
address(this), _amount);
        receivedBalance =
campaign.stakingToken.balanceOf(address(this)).sub(balanceBefore);
        user.amount = user.amount.add(receivedBalance);
        campaign.totalStaked = campaign.totalStaked.add(receivedBalance);
    }
    user.rewardDebt = user.amount.mul(campaign.accRewardPerShare).div(1e12);
```

```
    emit Deposit(msg.sender, receivedBalance, _campaignID);
}
```

5.9 Unsupported Design for Deflationary Staking Token in MasterBarista

ID	IDX-009
Target	MasterBarista
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved * LatteSwap team has confirmed that the `MasterBarista` contract will not support the deflationary token as the staking token.

5.9.1 Description

In the `MasterBarista` contract, the users can deposit their tokens to the reward token as $LATTE. The depositing token of each pool could be a simple token or LP token specified by the contract owner.

However, an issue could arise when the pool uses a deflationary token (the token that reduces the circulating supply itself when it is transferred) as the depositing token for a pool.

This means the amount of token that user deposits will be reduced due to the deflationary mechanism in transfer process, but the `MasterBarista` contract recognize it as the full amount as in line 518.

MasterBarista.sol

```
490 /// @dev Deposit token to get LATTE.
491 /// @param _stakeToken The stake token to be deposited
492 /// @param _amount The amount to be deposited
493 function deposit(
494     address _for,
495     address _stakeToken,
496     uint256 _amount
497   ) external override onlyPermittedTokenFunder(_for, _stakeToken) nonReentrant
{
498     _assignActiveToken();
499     require(
500       _stakeToken != address(0) && _stakeToken != address(1),
501       "MasterBarista::setPool::_stakeToken must not be address(0) or
address(1)"
502     );
503     require(_stakeToken != address(latte), "MasterBarista::deposit::use
```

```
depositLatte instead");
    require(pools.has(_stakeToken), "MasterBarista::deposit::no pool");

    PoolInfo storage pool = poolInfo[_stakeToken];
    UserInfo storage user = userInfo[_stakeToken][_for];

    if (user.fundedBy != address(0)) require(user.fundedBy == _msgSender(),
"MasterBarista::deposit::bad sof");

    uint256 lastRewardBlock = pool.lastRewardBlock;
    updatePool(_stakeToken);

    if (user.amount > 0) _harvest(_for, _stakeToken, lastRewardBlock);
    if (user.fundedBy == address(0)) user.fundedBy = _msgSender();
    if (_amount > 0) {
      IERC20(_stakeToken).safeTransferFrom(address(_msgSender()),
address(this), _amount);
      user.amount = user.amount.add(_amount);
    }

    user.rewardDebt = user.amount.mul(pool.accLattePerShare).div(1e12);
    user.bonusDebt =
user.amount.mul(pool.accLattePerShareTilBonusEnd).div(1e12);

    emit Deposit(_msgSender(), _for, _stakeToken, _amount);
}
```

The failure of recognizing the token amount could lead to the scenarios that are similarly described in **IDX-007 Unsupported Design for Deflationary Token Staking in DripBar**.

However, since deflationary tokens are not planned to be used as the staking token, there is no impact for this issue.

5.9.2 Remediation

Inspex suggests modifying the logic of the `deposit()` function to validate the amount of the token received from the user instead of using the value of `amount` parameter directly, for example:

MasterBarista.sol

```
/// @dev Deposit token to get LATTE.
/// @param _stakeToken The stake token to be deposited
/// @param _amount The amount to be deposited
function deposit(
    address _for,
    address _stakeToken,
    uint256 _amount
  ) external override onlyPermittedTokenFunder(_for, _stakeToken) nonReentrant
```

```
{
    _assignActiveToken();
    require(
      _stakeToken != address(0) && _stakeToken != address(1),
      "MasterBarista::setPool::_stakeToken must not be address(0) or
address(1)"
    );
    require(_stakeToken != address(latte), "MasterBarista::deposit::use
depositLatte instead");
    require(pools.has(_stakeToken), "MasterBarista::deposit::no pool");

    PoolInfo storage pool = poolInfo[_stakeToken];
    UserInfo storage user = userInfo[_stakeToken][_for];
    uint256 receivedBalance;

    if (user.fundedBy != address(0)) require(user.fundedBy == _msgSender(),
"MasterBarista::deposit::bad sof");

    uint256 lastRewardBlock = pool.lastRewardBlock;
    updatePool(_stakeToken);

    if (user.amount > 0) _harvest(_for, _stakeToken, lastRewardBlock);
    if (user.fundedBy == address(0)) user.fundedBy = _msgSender();
    if (_amount > 0) {
      uint256 balanceBefore = IERC20(_stakeToken).balanceOf(address(this));
      IERC20(_stakeToken).safeTransferFrom(address(_msgSender()),
address(this), _amount);
      receivedBalance =
IERC20(_stakeToken).balanceOf(address(this)).sub(balanceBefore);
      user.amount = user.amount.add(receivedBalance);
    }

    user.rewardDebt = user.amount.mul(pool.accLattePerShare).div(1e12);
    user.bonusDebt =
user.amount.mul(pool.accLattePerShareTilBonusEnd).div(1e12);

    emit Deposit(_msgSender(), _for, _stakeToken, receivedBalance);
}
```

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]

CYBERSECURITY PROFESSIONAL SERVICE