Farm, Swap, Token & Vault

Smart Contract Audit Report
Prepared for LatteSwap

Date Issued: Sep 19, 2021
Project ID: AUDIT2021003
Version: v3.0
Confidentiality Level: Public

inspeX
CYBERSECURITY PROFESSIONAL SERVICE

Report Information

Project ID	AUDIT2021003
Version	v3.0
Client	LatteSwap
Project	Farm, Swap, Token & Vault
Auditor(s)	Weerawat Pawanawiwat Pongsakorn Sommalai Suvicha Buakhom
Author	Pongsakorn Sommalai
Reviewer	Weerawat Pawanawiwat
Confidentiality Level	Public

Version History

Version	Date	Description	Author(s)
3.0	Sep 19, 2021	Update project name	Weerawat Pawanawiwat
2.0	Sep 19, 2021	Update project introduction	Weerawat Pawanawiwat
1.0	Jun 21, 2021	Full report	Pongsakorn Sommalai

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 Farm, Swap, Token & Vault smart contracts between Jun 7, 2021 and Jun 10, 2021. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of Farm, Swap, Token & Vault 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 3 high, 2 medium, 3 low, 1 very low, and 1 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 Farm, Swap, Token & Vault 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 an Automated Market Maker (AMM) protocol that is forked from Uniswap V2 and launched on the Binance Smart Chain. On LatteSwap, users can perform ERC20 token swapping easily with the liquidity pool of the platform. Users can also provide liquidity to the pools and gain a part of the swapping fee and the platform's reward tokens. Moreover, the $LATTE auto compounding can also be performed in the LatteSwap platform.

Scope Information:

Project Name	Farm, Swap, Token & Vault
Website	https://www.latteswap.com/
Smart Contract Type	Ethereum Smart Contract
Programming Language	Solidity

Audit Information:

Audit Method	Whitebox
Audit Date	Jun 7, 2021 - Jun 10, 2021
Reassessment Date	Jun 21, 2021

2.2. Scope

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

Initial Audit:

Name	Location (URL)
LATTE.sol	https://github.com/latteswap-official/latteswap-contract/blob/9fac06bea550fef4b2a9f5d0e0154964ff477410/contracts/farm/LATTE.sol
LatteVault.sol	https://github.com/latteswap-official/latteswap-contract/blob/9fac06bea550fef4b2a9f5d0e0154964ff477410/contracts/farm/LatteVault.sol
MasterBarista.sol	https://github.com/latteswap-official/latteswap-contract/blob/9fac06bea550fef4b2a9f5d0e0154964ff477410/contracts/farm/MasterBarista.sol
LatteSwapFactory.sol	https://github.com/latteswap-official/latteswap-contract/blob/9fac06bea550fef4b2a9f5d0e0154964ff477410/contracts/swap/LatteSwapFactory.sol
LatteSwapLP.sol	https://github.com/latteswap-official/latteswap-contract/blob/9fac06bea550fef

	4b2a9f5d0e0154964ff477410/contracts/swap/LatteSwapLP.sol
LatteSwapPair.sol	https://github.com/latteswap-official/latteswap-contract/blob/9fac06bea550fef4b2a9f5d0e0154964ff477410/contracts/swap/LatteSwapPair.sol
LatteSwapRouter.sol	https://github.com/latteswap-official/latteswap-contract/blob/9fac06bea550fef4b2a9f5d0e0154964ff477410/contracts/swap/LatteSwapRouter.sol
LatteSwapLibrary.sol	https://github.com/latteswap-official/latteswap-contract/blob/9fac06bea550fef4b2a9f5d0e0154964ff477410/contracts/swap/libraries/LatteSwapLibrary.sol
LatteSwapMath.sol	https://github.com/latteswap-official/latteswap-contract/blob/9fac06bea550fef4b2a9f5d0e0154964ff477410/contracts/swap/libraries/LatteSwapMath.sol
LatteSwapSafeMath.sol	https://github.com/latteswap-official/latteswap-contract/blob/9fac06bea550fef4b2a9f5d0e0154964ff477410/contracts/swap/libraries/LatteSwapSafeMath.sol
TransferHelper.sol	https://github.com/latteswap-official/latteswap-contract/blob/9fac06bea550fef4b2a9f5d0e0154964ff477410/contracts/swap/libraries/TransferHelper.sol
UQ112x112.sol	https://github.com/latteswap-official/latteswap-contract/blob/9fac06bea550fef4b2a9f5d0e0154964ff477410/contracts/swap/libraries/UQ112x112.sol

Reassessment:

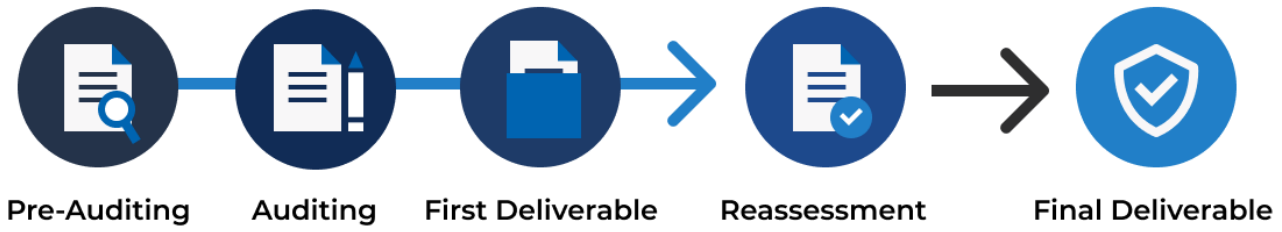
Name	Location (URL)
BeanBag.sol	https://github.com/latteswap-official/latteswap-contract/blob/b43fc4a6a283c2a6eff57a2289f2ed7548006cf6/contracts/farm/BeanBag.sol
LATTE.sol	https://github.com/latteswap-official/latteswap-contract/blob/b43fc4a6a283c2a6eff57a2289f2ed7548006cf6/contracts/farm/LATTE.sol
LatteVault.sol	https://github.com/latteswap-official/latteswap-contract/blob/b43fc4a6a283c2a6eff57a2289f2ed7548006cf6/contracts/farm/LatteVault.sol
MasterBarista.sol	https://github.com/latteswap-official/latteswap-contract/blob/b43fc4a6a283c2a6eff57a2289f2ed7548006cf6/contracts/farm/MasterBarista.sol
LinkList.sol	https://github.com/latteswap-official/latteswap-contract/blob/cf982b3a4ff8dc126bcd40ce8914baf81ca266f4/contracts/library/LinkList.sol
LatteSwapFactory.sol	https://github.com/latteswap-official/latteswap-contract/blob/b43fc4a6a283c2a6eff57a2289f2ed7548006cf6/contracts/swap/LatteSwapFactory.sol
LatteSwapLP.sol	https://github.com/latteswap-official/latteswap-contract/blob/b43fc4a6a283c2a6eff57a2289f2ed7548006cf6/contracts/swap/LatteSwapLP.sol
LatteSwapPair.sol	https://github.com/latteswap-official/latteswap-contract/blob/b43fc4a6a283c2

	a6eff57a2289f2ed7548006cf6/contracts/swap/LatteSwapPair.sol
LatteSwapRouter.sol	https://github.com/latteswap-official/latteswap-contract/blob/b43fc4a6a283c2a6eff57a2289f2ed7548006cf6/contracts/swap/LatteSwapRouter.sol
LatteSwapLibrary.sol	https://github.com/latteswap-official/latteswap-contract/blob/b43fc4a6a283c2a6eff57a2289f2ed7548006cf6/contracts/swap/libraries/LatteSwapLibrary.sol
LatteSwapMath.sol	https://github.com/latteswap-official/latteswap-contract/blob/b43fc4a6a283c2a6eff57a2289f2ed7548006cf6/contracts/swap/libraries/LatteSwapMath.sol
LatteSwapSafeMath.sol	https://github.com/latteswap-official/latteswap-contract/blob/b43fc4a6a283c2a6eff57a2289f2ed7548006cf6/contracts/swap/libraries/LatteSwapSafeMath.sol
TransferHelper.sol	https://github.com/latteswap-official/latteswap-contract/blob/b43fc4a6a283c2a6eff57a2289f2ed7548006cf6/contracts/swap/libraries/TransferHelper.sol
UQ112x112.sol	https://github.com/latteswap-official/latteswap-contract/blob/b43fc4a6a283c2a6eff57a2289f2ed7548006cf6/contracts/swap/libraries/UQ112x112.sol

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

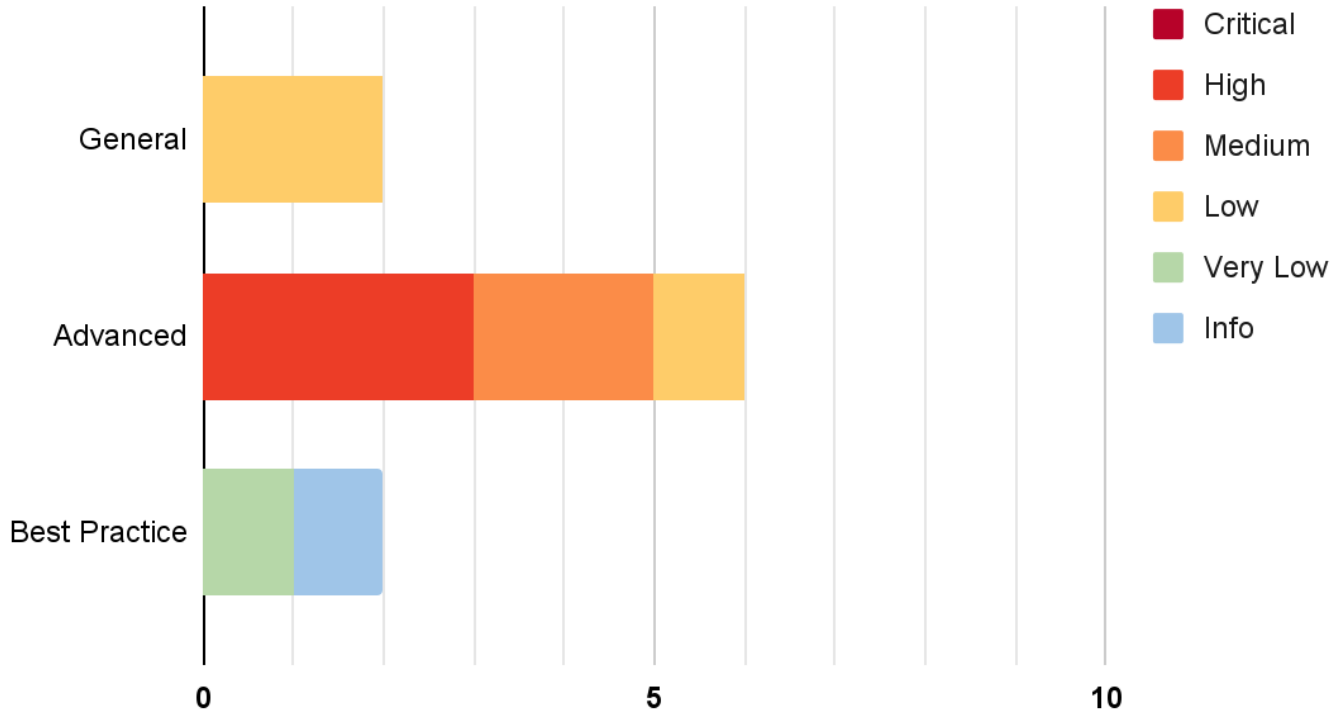
Both likelihood and impact can be categorized into three levels: **Low**, **Medium**, and **High**.

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**.

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

4. Summary of Findings

From the assessments, Inspex has found 10 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 complication.
Resolved *	The issue has been resolved with mitigations and clarifications.
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	Unclaimable Locked Rewards	Advanced	High	Resolved *
IDX-002	Improper Lock Amount Calculation	Advanced	High	Resolved
IDX-003	Improper Reward Calculation	Advanced	High	Resolved
IDX-004	Unchecked bonusLockUpBps Parameter Value	Advanced	Medium	Resolved
IDX-005	Improper Reward Distribution	Advanced	Medium	Resolved
IDX-006	Design Flaw in massUpdatePools() Function	General	Low	Resolved
IDX-007	Reduced Unlockable Amount for transferAll() Recipient	Advanced	Low	Resolved
IDX-008	Potential Centralized Control of State Variable	General	Low	Resolved *
IDX-009	Improper Update of State Variable	Best Practice	Very Low	Resolved
IDX-010	Improper Function Visibility	Best Practice	Info	Resolved

5. Detailed Findings Information

5.1. Unclaimable Locked Rewards

ID	IDX-001
Target	LatteVault.sol
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** High **Impact:** Medium A portion of the total $LATTE reward will be locked and unclaimable, causing the users to gain less $LATTE than expected. **Likelihood:** High All $LATTE locked during the bonus period is affected.
Status	Resolved * After discussing with the development team, `LatteVault` will only go live after the bonus period has ended. Without the bonus, there will be no locked $LATTE reward for the vault.

5.1.1. Description

In the `MasterBarista` contract, a part of the $LATTE reward minted during the bonus period is locked for the user when harvesting.

MasterBarista.sol

```
348 function _harvest(UserInfo storage _user, PoolInfo storage _pool) internal {
349     require(_user.amount > 0, "LatteMasterBarista::_harvest::nothing to
    harvest");
350     uint256 pending =
    _user.amount.mul(_pool.accLattePerShare).div(1e12).sub(_user.rewardDebt);
351     require(pending <= latte.balanceOf(address(this)),
    "LatteMasterBarista::_harvest::wait what.. not enough LATTE");
352     uint256 bonus =
    _user.amount.mul(_pool.accLattePerShareTilBonusEnd).div(1e12).sub(_user.bonusDe
    bt);
353     safeLattetransfer(_msgSender(), pending);
354     latte.lock(_msgSender(), bonus.mul(bonusLockUpBps).div(10000));
355 }
```

After the locking period has ended, the user can claim $LATTE directly from the $LATTE contract.

LATTE.sol

```
348 function unlock() external {
349     require(_locks[msg.sender] > 0, "LATTE::unlock::no locked LATTE");
350
351     uint256 amount = canUnlockAmount(msg.sender);
352
353     _transfer(address(this), msg.sender, amount);
354     _locks[msg.sender] = _locks[msg.sender].sub(amount);
355     _lastUnlockBlock[msg.sender] = block.number;
356     _totalLock = _totalLock.sub(amount);
357 }
```

However, in the `LatteVault` contract, there is no function for the user to claim the locked portion of the reward. This causes the locked $LATTE owned by the `LatteVault` contract to be unclaimable forever.

5.1.2. Remediation

Inspex suggests adding the $LATTE unlocking capability to the `harvest()` function of the `LatteVault` contract. When the farmer bot calls the `harvest()` function, the unlockable $LATTE should be unlocked.

LatteVault.sol

```
348 function harvest() external onlyEOA onlyFarmer whenNotPaused nonReentrant {
349     IMasterBarista(masterBarista).harvest(0);
350
351     latte.unlock();
352
353     uint256 bal = available();
354     uint256 currentPerformanceFee = bal.mul(performanceFee).div(10000);
355     latte.safeTransfer(treasury, currentPerformanceFee);
356
357     uint256 currentCallFee = bal.mul(callFee).div(10000);
358     latte.safeTransfer(treasury, currentCallFee);
359
360     _earn();
361
362     lastHarvestedTime = block.timestamp;
363
364     emit Harvest(msg.sender, currentPerformanceFee, currentCallFee);
365 }
```

Please note that the data type of `latte` variable must be changed to the `LATTE` interface to call the `unlock()` function.

5.2. Improper Lock Amount Calculation

ID	IDX-002
Target	MasterBarista.sol
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** High **Impact:** Medium Less amount of $LATTE will be locked than intended in the developer's portion during the bonus period. The unfair distribution of $LATTE can lead to reputation damage. **Likelihood:** High The developer's portion of the $LATTE minted during the bonus period is always affected.
Status	Resolved The LatteSwap team has resolved this issue by fixing the calculation of locked reward.

5.2.1. Description

In the `updatePool()` function of `MasterBarista` contract, a part of the total $LATTE is minted as the developer's portion. If it is within the bonus period, some part of that portion will be locked from the developer's address.

MasterBarista.sol

```
257 function updatePool(uint256 _pid) public override {
258     PoolInfo storage pool = poolInfo[_pid];
259     if (block.number <= pool.lastRewardBlock) {
260         return;
261     }
262     uint256 totalStakeToken = IERC20(pool.stakeToken).balanceOf(address(this));
263     if (totalStakeToken == 0) {
264         pool.lastRewardBlock = block.number;
265         return;
266     }
267     uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
268     uint256 latteReward =
    multiplier.mul(lattePerBlock).mul(pool.allocPoint).div(totalAllocPoint);
269     latte.mint(devAddr, latteReward.mul(1500).div(10000));
270     latte.mint(address(this), latteReward);
271     pool.accLattePerShare =
    pool.accLattePerShare.add(latteReward.mul(1e12).div(totalStakeToken));
272
273     // Clear bonus & update accLattePerShareTilBonusEnd.
```

```
274     if (block.number <= bonusEndBlock) {
275         latte.lock(devAddr, latteReward.mul(bonusLockUpBps).div(100000));
276         pool.accLattePerShareTilBonusEnd = pool.accLattePerShare;
277     }
278     if(block.number > bonusEndBlock && pool.lastRewardBlock < bonusEndBlock) {
279         uint256 latteBonusPortion =
    bonusEndBlock.sub(pool.lastRewardBlock).mul(bonusMultiplier).mul(lattePerBlock)
    .mul(pool.allocPoint).div(totalAllocPoint);
280         latte.lock(devAddr, latteBonusPortion.mul(bonusLockUpBps).div(100000));
281         pool.accLattePerShareTilBonusEnd =
    pool.accLattePerShareTilBonusEnd.add(latteBonusPortion.mul(1e12).div(totalStake
    Token));
282     }
283
284     pool.lastRewardBlock = block.number;
285 }
```

However, when comparing to the users' locked portion in the `_harvest()` function, the percentage differs.

MasterBarista.sol

```
348 function _harvest(UserInfo storage _user, PoolInfo storage _pool) internal {
349     require(_user.amount > 0, "LatteMasterBarista::_harvest::nothing to
    harvest");
350     uint256 pending =
    _user.amount.mul(_pool.accLattePerShare).div(1e12).sub(_user.rewardDebt);
351     require(pending <= latte.balanceOf(address(this)),
    "LatteMasterBarista::_harvest::wait what.. not enough LATTE");
352     uint256 bonus =
    _user.amount.mul(_pool.accLattePerShareTilBonusEnd).div(1e12).sub(_user.bonusDe
    bt);
353     safeLattetransfer(_msgSender(), pending);
354     latte.lock(_msgSender(), bonus.mul(bonusLockUpBps).div(10000));
355 }
```

Assuming that:

- `lockedAmount` is the amount of $LATTE to be locked
- `reward` is the amount of $LATTE reward minted for the users during the bonus period
- `devReward` is the amount of $LATTE reward minted as the developer's portion during the bonus period
- `lockedPercentage` is the percentage of an amount to be locked = `bonusLockUpBps` / 10000

The developer's portion of the $LATTE reward is calculated as follows:

MasterBarista.sol

```
269 latte.mint(devAddr, latteReward.mul(1500).div(10000));
```

Which is equivalent to:

```
devReward = reward * 1500 / 10000
```

So the developer's part is equal to 15% of the $LATTE reward minted for the users.

For the users, the locked portion is calculated as follows:

MasterBarista.sol

```
354 latte.lock(_msgSender(), bonus.mul(bonusLockUpBps).div(10000));
```

Which is equivalent to:

```
lockedAmount = reward * bonusLockUpBps / 10000
lockedAmount = reward * lockedPercentage
```

But for the developer's portion, the locked portion is calculated as follows:

MasterBarista.sol

```
275 latte.lock(devAddr, latteReward.mul(bonusLockUpBps).div(100000));
```

Which is equivalent to:

```
lockedAmount = reward * bonusLockUpBps / 100000
lockedAmount = (reward / 10) * bonusLockUpBps / 10000
lockedAmount = (reward / 10) * lockedPercentage
```

This means that for the locked portion calculation, 10% is used as the developer's portion instead of 15%, causing the locked amount to be lower than what it should be.

5.2.2. Remediation

The correct locked amount should be calculated as follows:

```
lockedAmount = devReward * lockedPercentage
lockedAmount = (reward * 1500 / 10000) * (bonusLockUpBps / 10000)
lockedAmount = reward * bonusLockUpBps * 15 / 1000000
```

Therefore, Inspex suggests fixing the calculation of the developer's locked amount as shown in the following example:

MasterBarista.sol

```
257 function updatePool(uint256 _pid) public override {
258     PoolInfo storage pool = poolInfo[_pid];
259     if (block.number <= pool.lastRewardBlock) {
```

```
260             return;
261         }
262         uint256 totalStakeToken = IERC20(pool.stakeToken).balanceOf(address(this));
263         if (totalStakeToken == 0) {
264             pool.lastRewardBlock = block.number;
265             return;
266         }
267         uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
268         uint256 latteReward =
    multiplier.mul(lattePerBlock).mul(pool.allocPoint).div(totalAllocPoint);
269         latte.mint(devAddr, latteReward.mul(1500).div(10000));
270         latte.mint(address(this), latteReward);
271         pool.accLattePerShare =
    pool.accLattePerShare.add(latteReward.mul(1e12).div(totalStakeToken));
272
273         // Clear bonus & update accLattePerShareTilBonusEnd.
274         if (block.number <= bonusEndBlock) {
275             latte.lock(devAddr,
    latteReward.mul(bonusLockUpBps).mul(15).div(1000000));
276             pool.accLattePerShareTilBonusEnd = pool.accLattePerShare;
277         }
278         if(block.number > bonusEndBlock && pool.lastRewardBlock < bonusEndBlock) {
279             uint256 latteBonusPortion =
    bonusEndBlock.sub(pool.lastRewardBlock).mul(bonusMultiplier).mul(lattePerBlock)
    .mul(pool.allocPoint).div(totalAllocPoint);
280             latte.lock(devAddr,
    latteBonusPortion.mul(bonusLockUpBps).mul(15).div(1000000));
281             pool.accLattePerShareTilBonusEnd =
    pool.accLattePerShareTilBonusEnd.add(latteBonusPortion.mul(1e12).div(totalStake
    Token));
282         }
283
284         pool.lastRewardBlock = block.number;
285     }
```

5.3. Improper Reward Calculation

ID	IDX-003
Target	MasterBarista.sol
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 The pool that has the same staking token as the reward token is predefined in the smart contract as the first pool.
Status	Resolved The LatteSwap team has resolved this issue by creating the new contract (BeanBag) to store the $LATTE reward.

5.3.1. Description

In the `MasterBarista` contract, a new staking pool can be added using the `addPool()` function. The staking token for the new pool is defined using the `_stakeToken` variable; however, there is no additional checking whether the `_stakeToken` is the same as the reward token (`latte`) or not.

MasterBarista.sol

```
144 function addPool(
145       uint256 _allocPoint,
146       address _stakeToken
147 ) external override onlyOwner {
148       require(_stakeToken != address(0),
    "LatteMasterBarista::addPool::_stakeToken must not be address(0)");
149       require(!isDuplicatedPool(_stakeToken),
    "LatteMasterBarista::addPool::_stakeToken duplicated");
150
151       massUpdatePools();
152
153       uint256 lastRewardBlock = block.number > startBlock ? block.number :
    startBlock;
154       totalAllocPoint = totalAllocPoint.add(_allocPoint);
155       poolInfo.push(
156             PoolInfo({
157                   stakeToken: _stakeToken,
```

```
158                   allocPoint: _allocPoint,
159                   lastRewardBlock: lastRewardBlock,
160                   accLattePerShare: 0,
161                   accLattePerShareTilBonusEnd: 0
162             })
163       );
164
165       updatePool0alloc();
166
167       emit AddPool(_stakeToken, _allocPoint, totalAllocPoint);
168 }
```

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

MasterBarista.sol

```
257 function updatePool(uint256 _pid) public override {
258       PoolInfo storage pool = poolInfo[_pid];
259       if (block.number <= pool.lastRewardBlock) {
260             return;
261       }
262       uint256 totalStakeToken =
    IERC20(pool.stakeToken).balanceOf(address(this));
263       if (totalStakeToken == 0) {
264             pool.lastRewardBlock = block.number;
265             return;
266       }
267       uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
268       uint256 latteReward =
    multiplier.mul(lattePerBlock).mul(pool.allocPoint).div(totalAllocPoint);
269       latte.mint(devAddr, latteReward.mul(1500).div(10000));
270       latte.mint(address(this), latteReward);
271       pool.accLattePerShare =
    pool.accLattePerShare.add(latteReward.mul(1e12).div(totalStakeToken));
272
273       // Clear bonus & update accLattePerShareTilBonusEnd.
274       if (block.number <= bonusEndBlock) {
275             latte.lock(devAddr, latteReward.mul(bonusLockUpBps).div(100000));
276             pool.accLattePerShareTilBonusEnd = pool.accLattePerShare;
277       }
278       if(block.number > bonusEndBlock && pool.lastRewardBlock < bonusEndBlock)
    {
279             uint256 latteBonusPortion =
    bonusEndBlock.sub(pool.lastRewardBlock).mul(bonusMultiplier).mul(lattePerBlock)
```

```
.mul(pool.allocPoint).div(totalAllocPoint);
280             latte.lock(devAddr,
latteBonusPortion.mul(bonusLockUpBps).div(100000));
281             pool.accLattePerShareTilBonusEnd =
pool.accLattePerShareTilBonusEnd.add(latteBonusPortion.mul(1e12).div(totalStake
Token));
282         }
283
284         pool.lastRewardBlock = block.number;
285     }
```

5.3.2. Recommendation

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.

5.4. Unchecked bonusLockUpBps Parameter Value

ID	IDX-004
Target	MasterBarista.sol
Category	Advanced Smart Contract Vulnerability
CWE	CWE-20: Improper Input Validation
Risk	**Severity:** Medium **Impact:** High An improper setting of the bonusLockUpBps variable can cause the smart contract to be unusable for the users. **Likelihood:** Low The percentage of the locked portion should have been calculated before the bonus setting is applied, but it is still possible for an improper value to be set.
Status	Resolved The LatteSwap team has resolved this issue by adding the validation for the value of bonusLockUpBps.

5.4.1. Description

In the `MasterBarista` contract, the `bonusLockUpBps` variable is used for calculating the portion of the minted reward to be locked. The value is divided by 10000, meaning that if `bonusLockUpBps` is 10000, 100% of the reward will be locked.

MasterBarista.sol

```
348 function _harvest(UserInfo storage _user, PoolInfo storage _pool) internal {
349     require(_user.amount > 0, "LatteMasterBarista::_harvest::nothing to
    harvest");
350     uint256 pending =
    _user.amount.mul(_pool.accLattePerShare).div(1e12).sub(_user.rewardDebt);
351     require(pending <= latte.balanceOf(address(this)),
    "LatteMasterBarista::_harvest::wait what.. not enough LATTE");
352     uint256 bonus =
    _user.amount.mul(_pool.accLattePerShareTilBonusEnd).div(1e12).sub(_user.bonusDe
    bt);
353     safeLattetransfer(_msgSender(), pending);
354     latte.lock(_msgSender(), bonus.mul(bonusLockUpBps).div(10000));
355 }
```

However, in the `setBonus()` function, there is no validation for the value of `bonusLockUpBps`, allowing it to be set to a value over 10000. When this happens, the amount to be locked will be more than the actual amount sent to the user, causing the smart contract to be unusable for the user without sufficient $LATTE.

MasterBalista.sol

```
124 function setBonus(
125     uint256 _bonusMultiplier,
126     uint256 _bonusEndBlock,
127     uint256 _bonusLockUpBps
128     ) external onlyOwner {
129     require(_bonusEndBlock > block.number, "LatteMasterBarista::setBonus::bad
    bonusEndBlock");
130     require(_bonusMultiplier > 100, "LatteMasterBarista::setBonus::bad
    bonusMultiplier");
131
132     massUpdatePools();
133
134     bonusMultiplier = _bonusMultiplier;
135     bonusEndBlock = _bonusEndBlock;
136     bonusLockUpBps = _bonusLockUpBps;
137
138     emit BonusChanged(bonusMultiplier, bonusEndBlock, bonusLockUpBps);
139 }
```

5.4.2. Remediation

Inspex suggests adding validation for the value of `bonusLockUpBps`, preventing the lock amount to be over 100% of the reward given, for example:

MasterBalista.sol

```
124 function setBonus(
125     uint256 _bonusMultiplier,
126     uint256 _bonusEndBlock,
127     uint256 _bonusLockUpBps
128     ) external onlyOwner {
129     require(_bonusEndBlock > block.number, "LatteMasterBarista::setBonus::bad
    bonusEndBlock");
130     require(_bonusMultiplier > 100, "LatteMasterBarista::setBonus::bad
    bonusMultiplier");
131     require(_bonusLockUpBps <= 10000, "LatteMasterBarista::setBonus::bad
    bonusLockUpBps");
132
133     massUpdatePools();
134
135     bonusMultiplier = _bonusMultiplier;
136     bonusEndBlock = _bonusEndBlock;
137     bonusLockUpBps = _bonusLockUpBps;
```

```
138
139         emit BonusChanged(bonusMultiplier, bonusEndBlock, bonusLockUpBps);
140     }
```

5.5. Improper Reward Distribution

ID	IDX-005
Target	LatteVault.sol
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** High A part of the $LATTE reward could be claimed by a user without any prior token deposited, causing the other investors to gain less $LATTE reward. **Likelihood:** Low The chance for the procedure to be profitable is low due to the withdrawal fee and the frequent calling of `_earn()` from the farmer and other users.
Status	Resolved The LatteSwap team has resolved this issue by harvesting the reward before performing the deposit.

5.5.1. Description

In the `MasterBarista` contract, the `deposit()` function is used for staking $LATTE into the contract. Whenever this function is called, if the user already has $LATTE staked in the contract, the `_harvest()` function is also called.

MasterBarista.sol

```
290 function deposit(uint256 _pid, uint256 _amount) external override nonReentrant
    {
291     PoolInfo storage pool = poolInfo[_pid];
292     UserInfo storage user = userInfo[_pid][_msgSender()];
293     require(pool.stakeToken != address(0), "LatteMasterBarista::deposit::not
    accept deposit");
294     updatePool(_pid);
295     if (user.amount > 0) _harvest(user, pool);
296     IERC20(pool.stakeToken).safeTransferFrom(address(msg.sender),
    address(this), _amount);
297     user.amount = user.amount.add(_amount);
298     user.rewardDebt = user.amount.mul(pool.accLattePerShare).div(1e12);
299     user.bonusDebt =
    user.amount.mul(pool.accLattePerShareTilBonusEnd).div(1e12);
300     emit Deposit(msg.sender, _pid, _amount);
301 }
```

The `_harvest()` function transfers the pending $LATTE reward to the user's address. Even if the user deposits all of the currently owned $LATTE, some can be left inside the user's address due to the harvested amount returned. This causes no problem for the users interacting directly with this contract, however; it may cause some unfair reward distribution for the current implementation of the `LatteVault` contract.

MasterBarista.sol

```
348 function _harvest(UserInfo storage _user, PoolInfo storage _pool) internal {
349     require(_user.amount > 0, "LatteMasterBarista::_harvest::nothing to
    harvest");
350     uint256 pending =
    _user.amount.mul(_pool.accLattePerShare).div(1e12).sub(_user.rewardDebt);
351     require(pending <= latte.balanceOf(address(this)),
    "LatteMasterBarista::_harvest::wait what.. not enough LATTE");
352     uint256 bonus =
    _user.amount.mul(_pool.accLattePerShareTilBonusEnd).div(1e12).sub(_user.bonusDe
    bt);
353     safeLattetransfer(_msgSender(), pending);
354     latte.lock(_msgSender(), bonus.mul(bonusLockUpBps).div(10000));
355 }
```

The `LatteVault` contract is a vault for the users to collectively stake and compound $LATTE reward in the `MasterBarista` contract. Users can use the `deposit()` function to deposit $LATTE into the `LatteVault` contract, which then calls the `_earn()` function.

LatteVault.sol

```
107 function deposit(uint256 _amount) external whenNotPaused nonReentrant onlyEOA {
108     require(_amount > 0, "LatteVault::deposit::nothing to deposit");
109
110     uint256 pool = balanceOf();
111     latte.safeTransferFrom(msg.sender, address(this), _amount);
112     uint256 currentShares = 0;
113     if (totalShares != 0) {
114         currentShares = (_amount.mul(totalShares)).div(pool);
115     } else {
116         currentShares = _amount;
117     }
118     UserInfo storage user = userInfo[msg.sender];
119
120     user.shares = user.shares.add(currentShares);
121     user.lastDepositedTime = block.timestamp;
122
123     totalShares = totalShares.add(currentShares);
124
125     user.latteAtLastUserAction = user.shares.mul(balanceOf()).div(totalShares);
126     user.lastUserActionTime = block.timestamp;
127
```

```
128     _earn();
129
130     emit Deposit(msg.sender, _amount, currentShares, block.timestamp);
131 }
```

The `_earn()` function checks the current $LATTE balance in the contract, then calls the `MasterBarista`'s `deposit()` function to stake $LATTE.

LatteVault.sol

```
347 function _earn() internal {
348     uint256 bal = available();
349     if (bal > 0) {
350         IMasterBarista(masterBarista).deposit(0, bal);
351     }
352 }
```

As shown previously, the `MasterBarista`'s `deposit()` function harvests and returns the pending $LATTE reward to the user, this means that the $LATTE reward of the vault since the last call of`_earn()` stays inside the `LatteVault` contract until the next call.

The `withdraw()` function can be used to withdraw the $LATTE staked together with the user's share of the staking reward. In this function, the `balanceOf()` function is used as one of the factors to calculate the withdrawable amount.

LatteVault.sol

```
288 function withdraw(uint256 _shares) public onlyEOA nonReentrant {
289     UserInfo storage user = userInfo[msg.sender];
290     require(_shares > 0, "LatteVault::withdraw::nothing to withdraw");
291     require(_shares <= user.shares, "LatteVault::withdraw::withdraw amount
    exceeds balance");
292
293     uint256 currentAmount = (balanceOf().mul(_shares)).div(totalShares);
294     user.shares = user.shares.sub(_shares);
295     totalShares = totalShares.sub(_shares);
296
297     uint256 bal = available();
298     if (bal < currentAmount) {
299         uint256 balWithdraw = currentAmount.sub(bal);
300         IMasterBarista(masterBarista).withdraw(0, balWithdraw);
301         uint256 balAfter = available();
302         uint256 diff = balAfter.sub(bal);
303         if (diff < balWithdraw) {
304             currentAmount = bal.add(diff);
305         }
306     }
```

```
307
308     if (block.timestamp < user.lastDepositedTime.add(withdrawFeePeriod)) {
309         uint256 currentWithdrawFee = currentAmount.mul(withdrawFee).div(10000);
310         latte.safeTransfer(treasury, currentWithdrawFee);
311         currentAmount = currentAmount.sub(currentWithdrawFee);
312     }
313
314     if (user.shares > 0) {
315         user.latteAtLastUserAction =
    user.shares.mul(balanceOf()).div(totalShares);
316     } else {
317         user.latteAtLastUserAction = 0;
318     }
319
320     user.lastUserActionTime = block.timestamp;
321
322     latte.safeTransfer(msg.sender, currentAmount);
323
324     emit Withdraw(msg.sender, currentAmount, _shares);
325 }
```

The `balanceOf()` function returns the sum of the $LATTE staked in `MasterBarista` contract by the vault contract and the current balance of $LATTE in the vault contract. This means that $LATTE harvested from the last call of `_earn()` is included in the withdrawal amount calculation.

LatteVault.sol

```
339 function balanceOf() public view returns (uint256) {
340     (uint256 amount, ,) = IMasterBarista(masterBarista).userInfo(0,
    address(this));
341     return latte.balanceOf(address(this)).add(amount);
342 }
```

This opens up a room for anyone to quickly call `deposit()` then `withdraw()` whenever the vault's pending reward is high. Calling `deposit()` will harvest the pending $LATTE reward to the vault contract, and calling `withdraw()` will allow the user to gain a portion of the $LATTE reward without having to leave one's $LATTE staked.

5.5.2. Remediation

Inspex suggests adding a `MasterBarista.harvest()` function call in the `deposit()` function before checking the balance of the pool. This call will harvest the pending reward before calculating the user's shares, then the second call of `_earn()` at the end will stake all $LATTE, leaving no token left in the contract. The example fix is shown in the source code below:

LatteVault.sol

```
107 function deposit(uint256 _amount) external whenNotPaused nonReentrant onlyEOA {
108     require(_amount > 0, "LatteVault::deposit::nothing to deposit");
109 
110     IMasterBarista(masterBarista).harvest(0);
111 
112     uint256 pool = balanceOf();
113     latte.safeTransferFrom(msg.sender, address(this), _amount);
114     uint256 currentShares = 0;
115     if (totalShares != 0) {
116         currentShares = (_amount.mul(totalShares)).div(pool);
117     } else {
118         currentShares = _amount;
119     }
120     UserInfo storage user = userInfo[msg.sender];
121 
122     user.shares = user.shares.add(currentShares);
123     user.lastDepositedTime = block.timestamp;
124 
125     totalShares = totalShares.add(currentShares);
126 
127     user.latteAtLastUserAction = user.shares.mul(balanceOf()).div(totalShares);
128     user.lastUserActionTime = block.timestamp;
129 
130     _earn();
131 
132     emit Deposit(msg.sender, _amount, currentShares, block.timestamp);
133 }
```

5.6. Design Flaw in massUpdatePools() Function

ID	IDX-006
Target	MasterBarista.sol
Category	General Smart Contract Vulnerability
CWE	CWE-400: Uncontrolled Resource Consumption
Risk	**Severity:** Low **Impact:** Medium The `massUpdatePools()` function will eventually be unusable due to excessive gas usage. **Likelihood:** Low It is very unlikely that the poolInfo size will be raised until the gas usage of `massUpdatePools()` exceeds the maximum gas limit.
Status	Resolved The LatteSwap team has resolved this issue by changing the data structure, storing the list of pools to a linked list and adding `removePool()` function.

5.6.1. Description

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

MasterBarista.sol

```
248 function massUpdatePools() public {
249     uint256 length = poolInfo.length;
250     for (uint256 pid = 0; pid < length; ++pid) {
251         updatePool(pid);
252     }
253 }
```

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 the 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 may eventually be unusable due to excessive gas usage.

5.6.2. Remediation

Inspex suggests making the contract capable of removing unnecessary/ended pools from `poolInfo` to reduce the loop rounds in the `massUpdatePools()` function, for example:

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

5.7. Reduced Unlockable Amount for transferAll() Recipient

ID	IDX-007
Target	LATTE.sol
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Low **Impact:** Medium The victim's unlockable amount before the `endReleaseBlock` is reduced, allowing the attacker to gain an early advantage of $LATTE unlock amount over the victim. **Likelihood:** Low This attack requires multiple prior steps and cannot be done to multiple addresses at once.
Status	Resolved The LatteSwap team has resolved this issue. The new calculation of `_lastUnlockBlock` value has been implemented.

5.7.1. Description

The $LATTE has a locking mechanic that releases an increasing amount of the locked tokens over time. The unlockable amount is calculated in the `canUnlockAmount()` function. The higher `_lastUnlockBlock` grows, the lower $LATTE can be unlocked.

LATTE.sol

```
142 function canUnlockAmount(address _account) public view returns (uint256) {
143     // When block number less than startReleaseBlock, no LATTEs can be unlocked
144     if (block.number < startReleaseBlock) {
145         return 0;
146     }
147     // When block number more than endReleaseBlock, all locked LATTEs can be
    unlocked
148     else if (block.number >= endReleaseBlock) {
149         return _locks[_account];
150     }
151     // When block number is more than startReleaseBlock but less than
    endReleaseBlock,
152     // some LATTEs can be released
153     else
154     {
155         uint256 releasedBlock = block.number.sub(_lastUnlockBlock[_account]);
```

```
156             uint256 blockLeft = endReleaseBlock.sub(_lastUnlockBlock[_account]);
157             return _locks[_account].mul(releasedBlock).div(blockLeft);
158         }
159     }
```

The locked amount can be transferred to another address using the `transferAll()` function. This function transfers the whole $LATTE balance of the current address to another, including the locked balance. Also, if the value of `_lastUnlockBlock` of the recipient is lower than the sender's, it is also updated.

LATTE.sol

```
175 function transferAll(address _to) external {
176     require(msg.sender != _to, "LATTE::transferAll::no self-transferAll");
177 
178     _locks[_to] = _locks[_to].add(_locks[msg.sender]);
179 
180     if (_lastUnlockBlock[_to] < startReleaseBlock) {
181         _lastUnlockBlock[_to] = startReleaseBlock;
182     }
183 
184     if (_lastUnlockBlock[_to] < _lastUnlockBlock[msg.sender]) {
185         _lastUnlockBlock[_to] = _lastUnlockBlock[msg.sender];
186     }
187 
188     _locks[msg.sender] = 0;
189     _lastUnlockBlock[msg.sender] = 0;
190 
191     _transfer(msg.sender, _to, balanceOf(msg.sender));
192 }
```

With this, an attacker can use an address with a tiny amount staked in `MasterBarista` to call `MasterBarista.harvest()` and `MasterBarista.unlock()`, then use `transferAll()` function to update the victim's `_lastUnlockBlock`, reducing the victim's unlockable amount.

5.7.2. Remediation

Inspex suggests modifying the logic of the `transferAll()` function by calculating a new `_lastUnlockBlock` value that keeps the integrity of the unlockable amount.

To make the `transferAll()` function fair, the new unlockable amount of the recipient must be equal to the sum of the unlockable amount of the sender and recipient. This can be done by setting the `_lastUnlockBlock[to]` to a new value (`newLastUnlockBlock`) that makes the following equation true:

```
(canUnlockAmount(msg.sender) + canUnlockAmount(_to)) / (_locks[msg.sender] +
_locks[_to]) = (block.number - newLastUnlockBlock) / (endReleaseBlock -
newLastUnlockBlock)
```

To solve for `newLastUnlockBlock`, the equation can be rewritten as follows:

```
numerator = (block.number * _locks[msg.sender]) + (block.number * _locks[_to]) -
(endReleaseBlock * canUnlockAmount(msg.sender)) - (endReleaseBlock *
canUnlockAmount(_to))
denominator = _locks[msg.sender] + _locks[_to] - canUnlockAmount(msg.sender) -
canUnlockAmount(_to)
newLastUnlockBlock = numerator / denominator
```

As a result, the fix can be seen in the following example:

LATTE.sol

```
175 function transferAll(address _to) external {
176     require(msg.sender != _to, "LATTE::transferAll::no self-transferAll");
177
178     _locks[_to] = _locks[_to].add(_locks[msg.sender]);
179
180     if (_lastUnlockBlock[_to] < startReleaseBlock) {
181       _lastUnlockBlock[_to] = startReleaseBlock;
182     }
183
184     else if (block.number < endReleaseBlock) {
185         uint256 fromUnlocked = canUnlockAmount(msg.sender);
186         uint256 toUnlocked = canUnlockAmount(_to);
187         uint256 numerator =
    block.number.mul(_locks[msg.sender]).add(block.number.mul(_locks[_to])).sub(end
    ReleaseBlock.mul(fromUnlocked))).sub(endReleaseBlock.mul(toUnlocked));
188         uint256 denominator =
    _locks[msg.sender].add(_locks[_to]).sub(fromUnlocked).sub(toUnlocked);
189         _lastUnlockBlock[_to] = numerator.div(denominator);
190     }
191
192     _locks[msg.sender] = 0;
193     _lastUnlockBlock[msg.sender] = 0;
194
195     _transfer(msg.sender, _to, balanceOf(msg.sender));
196 }
```

5.8. Potential Centralized Control of State Variable

ID	IDX-008
Target	MasterBarista.sol LatteVault.sol
Category	General Smart Contract Vulnerability
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	**Severity:** Low **Impact:** Low 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, the changes are limited by fixed values in the smart contracts.
Status	Resolved * The LatteSwap team has planned to deploy the contract with Timelock.

5.8.1. 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 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
MasterBarista.sol (L:105)	setLattePerBlock()	onlyOwner
MasterBarista.sol (L:112)	setPool0allocBps()	onlyOwner
MasterBarista.sol (L:124)	setBonus()	onlyOwner
MasterBarista.sol (L:144)	addPool()	onlyOwner
MasterBarista.sol (L:173)	setPool()	onlyOwner
LatteVault.sol (L:165)	setAdmin()	onlyOwner

LatteVault.sol (L:174)	setTreasury()	onlyOwner
LatteVault.sol (L:183)	setPerformanceFee()	onlyAdmin
LatteVault.sol (L:192)	setCallFee()	onlyAdmin
LatteVault.sol (L:201)	setWithdrawFee()	onlyAdmin
LatteVault.sol (L:210)	setWithdrawFeePeriod()	onlyAdmin

5.8.2. 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

5.9. Improper Update of State Variable

ID	IDX-009
Target	LatteVault.sol
Category	Smart Contract Best Practice
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Very Low **Impact:** Low The miscalculation of `user.latteAtLastUserAction` can lead to reputation damage when this value is used in the public website. **Likelihood:** Low It is very unlikely that this miscalculation can cause issues in the `LatteVault` contract.
Status	Resolved The LatteSwap team has resolved this issue by reordering the function logic.

5.9.1. Description

In the `withdraw()` function, the value of `user.latteAtLastUserAction` is calculated from the `balanceOf()` function.

LatteVault.sol

```
314 if (user.shares > 0) {
315     user.latteAtLastUserAction = user.shares.mul(balanceOf()).div(totalShares);
316 } else {
317     user.latteAtLastUserAction = 0;
318 }
319
320 user.lastUserActionTime = block.timestamp;
321
322 latte.safeTransfer(msg.sender, currentAmount);
```

The `balanceOf()` function calculates the current balance of $LATTE owned by this contract by adding the staked $LATTE to the current $LATTE balance.

LatteVault.sol

```
339 function balanceOf() public view returns (uint256) {
340     (uint256 amount, ,) = IMasterBarista(masterBarista).userInfo(0,
    address(this));
341     return latte.balanceOf(address(this)).add(amount);
342 }
```

As shown in the following source code, $LATTE is transferred after the value of `user.latteAtLastUserAction` is calculated.

LatteVault.sol

```
314 if (user.shares > 0) {
315     user.latteAtLastUserAction = user.shares.mul(balanceOf()).div(totalShares);
316 } else {
317     user.latteAtLastUserAction = 0;
318 }
319
320 user.lastUserActionTime = block.timestamp;
321
322 latte.safeTransfer(msg.sender, currentAmount);
```

As a result, the value of `user.latteAtLastUserAction` is higher than it should be. However, the value is not being used in any of the in-scope contracts. Therefore, there is currently no security impact.

5.9.2. Remediation

Inspex suggests transferring the $LATTE before calculating the `user.latteAtLastUserAction` as in the following example:

LatteVault.sol

```
314 user.lastUserActionTime = block.timestamp;
315
316 latte.safeTransfer(msg.sender, currentAmount);
317
318 if (user.shares > 0) {
319     user.latteAtLastUserAction = user.shares.mul(balanceOf()).div(totalShares);
320 } else {
321     user.latteAtLastUserAction = 0;
322 }
```

5.10. Improper Function Visibility

ID	IDX-010
Target	LATTE.sol MasterBarista.sol LatteSwapRouter.sol
Category	Smart Contract Best Practice
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	**Severity:** Info **Impact:** None **Likelihood:** None
Status	Resolved The LatteSwap team has resolved this issue by changing all affected functions' visibility to `external`.

5.10.1. 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 `setCap()` function of the `LATTE` contract is set to public and it is never called from any internal function.

LATTE.sol

```
58 function setCap(uint256 _cap) public onlyGovernor {
59     require(_cap < cap, "LATTE::setCap::_cap must < cap");
60     uint256 prevCap = cap;
61     cap = _cap;
62     emit CapChanged(prevCap, cap);
63 }
```

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

Target	Function
LATTE.sol (L:58)	setCap()
LATTE.sol (L:67)	setGovernor()

LATTE.sol (L:77)	mint()
MasterBarista.sol (L:98)	setDev()
LatteSwapRouter.sol (L:419)	quote()
LatteSwapRouter.sol (L:423)	getAmountOut()
LatteSwapRouter.sol (L:433)	getAmountIn()
LatteSwapRouter.sol (L:443)	getAmountsOut()
LatteSwapRouter.sol (L:453)	getAmountsIn()

5.10.2. Remediation

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

LATTE.sol

```
58 function setCap(uint256 _cap) external onlyGovernor {
59     require(_cap < cap, "LATTE::setCap::_cap must < cap");
60     uint256 prevCap = cap;
61     cap = _cap;
62     emit CapChanged(prevCap, cap);
63 }
```

6. Appendix

6.1. About Inspex

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

Follow Us On:

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

6.2. References

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

inspeX

CYBERSECURITY PROFESSIONAL SERVICE