MasterBarista & NFT

Smart Contract Audit Report
Prepared for LatteSwap

Date Issued:	Sep 17, 2021
Project ID:	AUDIT2021015
Version:	v1.0
Confidentiality Level:	Public

inspex

CYBERSECURITY PROFESSIONAL SERVICE

Report Information

Project ID	AUDIT2021015
Version	v1.0
Client	LatteSwap
Project	MasterBarista & NFT
Auditor(s)	Weerawat Pawanawiwat Peeraphut Punsuwan
Author	Weerawat Pawanawiwat
Reviewer	Suvicha Buakhom
Confidentiality Level	Public

Version History

Version	Date	Description	Author(s)
1.0	Sep 17, 2021	Full report	Weerawat Pawanawiwat

Contact Information

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

Table of Contents

1. Executive Summary

As requested by LatteSwap, Inspex team conducted an audit to verify the security posture of the MasterBarista & NFT smart contracts between Aug 24, 2021 and Aug 27, 2021. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of MasterBarista & NFT 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 6 high, 3 medium, 3 low, 2 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 MasterBarista & NFT smart contracts have high-level protections in place to be safe from most attacks.

This smart contract passes Inspex's security verification standard, and is trustworthy.

Approved by Inspex on Sep 17, 2021

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 & NFT smart contracts handle the distribution of NFTs and $LATTE. Users can gain $LATTE reward from yield farming on MasterBarista, and with Booster NFTs integrated, additional profit can be gained by staking the NFTs.

Scope Information:

Project Name	MasterBarista & NFT
Website	https://latteswap.com/
Smart Contract Type	Ethereum Smart Contract
Chain	Binance Smart Chain
Programming Language	Solidity

Audit Information:

Audit Method	Whitebox
Audit Date	Aug 24, 2021 - Aug 27, 2021
Reassessment Date	Sep 16, 2021

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

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

2.2. Scope

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

Initial Audit: (Commit: bed0ee0998a6181233f576c4061948856139e45f)

Contract	Location (URL)
MasterBarista	https://github.com/latteswap-official/latteswap-contract/blob/bed0ee0998/contracts/farm/MasterBarista.sol
Booster	https://github.com/latteswap-official/latteswap-contract/blob/bed0ee0998/contracts/nft/Booster.sol
BoosterConfig	https://github.com/latteswap-official/latteswap-contract/blob/bed0ee0998/contracts/nft/BoosterConfig.sol
LatteMarket	https://github.com/latteswap-official/latteswap-contract/blob/bed0ee0998/contracts/nft/LatteMarket.sol
LatteNFT	https://github.com/latteswap-official/latteswap-contract/blob/bed0ee0998/contracts/nft/LatteNFT.sol
OGNFT	https://github.com/latteswap-official/latteswap-contract/blob/bed0ee0998/contracts/nft/OGNFT.sol
OGOwnerToken	https://github.com/latteswap-official/latteswap-contract/blob/bed0ee0998/contracts/nft/OGOwnerToken.sol

Reassessment: (Commit: 08c8cd82586e165196bdc060dd80747befa9d578)

Contract	Location (URL)
MasterBarista	https://github.com/latteswap-official/latteswap-contract/blob/08c8cd8258/contracts/farm/MasterBarista.sol
Booster	https://github.com/latteswap-official/latteswap-contract/blob/08c8cd8258/contracts/nft/Booster.sol
BoosterConfig	https://github.com/latteswap-official/latteswap-contract/blob/08c8cd8258/contracts/nft/BoosterConfig.sol
LatteMarket	https://github.com/latteswap-official/latteswap-contract/blob/08c8cd8258/contracts/nft/LatteMarket.sol
LatteNFT	https://github.com/latteswap-official/latteswap-contract/blob/08c8cd8258/contracts/nft/LatteNFT.sol
OGNFT	https://github.com/latteswap-official/latteswap-contract/blob/08c8cd8258/contracts/nft/OGNFT.sol

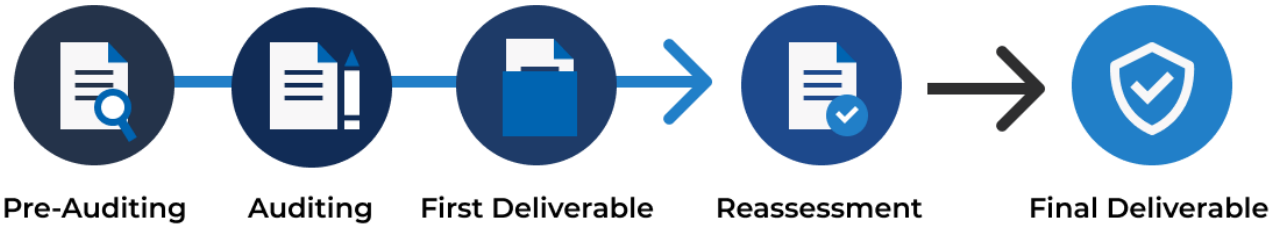
OGOwnerToken	https://github.com/latteswap-official/latteswap-contract/blob/08c8cd8258/contracts/nft/OGOwnerToken.sol
OGNFTOffering	https://github.com/latteswap-official/latteswap-contract/blob/08c8cd8258/contracts/nft/OGNFTOffering.sol
TripleSlopePriceModel	https://github.com/latteswap-official/latteswap-contract/blob/08c8cd8258/contracts/nft/og-price-models/TripleSlopePriceModel.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

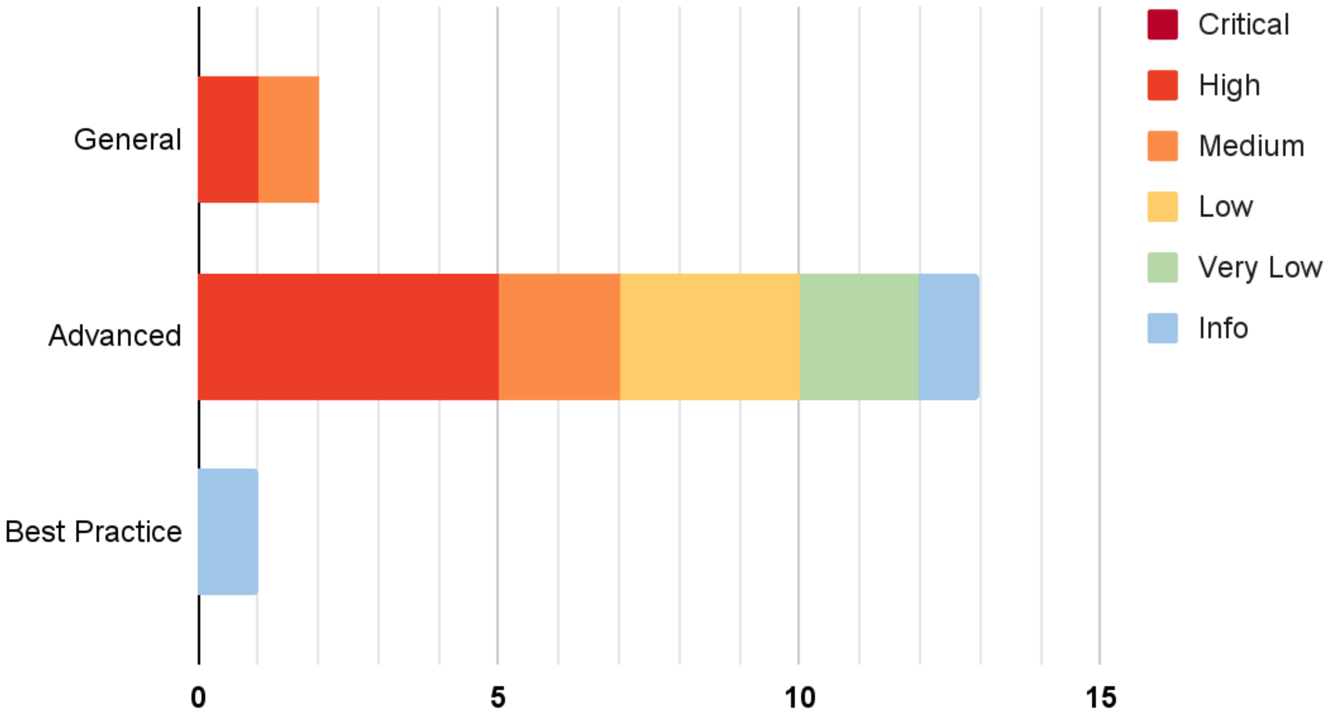
Both likelihood and impact can be categorized into three levels: **Low**, **Medium**, and **High**.

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 16 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	Improper Cap Deduction	Advanced	High	Resolved
IDX-002	Use of Upgradable Contract	Advanced	High	Resolved *
IDX-003	Unrestricted $LATTE Minting	Advanced	High	Resolved *
IDX-004	Unrestricted Boosted $LATTE Minting	Advanced	High	Resolved *
IDX-005	Improper NFT Burning	Advanced	High	Resolved
IDX-006	Division Before Multiplication	General	High	Resolved
IDX-007	Centralized Control of State Variable	General	Medium	Resolved *
IDX-008	Improper Token Burning	Advanced	Medium	Resolved
IDX-009	Unchecked Max Value	Advanced	Medium	Resolved
IDX-010	Auction Cancellation	Advanced	Low	Resolved
IDX-011	Improper Minimum allocBps Condition	Advanced	Low	Resolved
IDX-012	Improper Maximum accumAllocBps Condition	Advanced	Low	Resolved
IDX-013	Improper Selling and Auction Starting Condition Checking	Advanced	Very Low	Resolved
IDX-014	Insufficient Logging for Privileged Functions	Advanced	Very Low	Resolved
IDX-015	Improper Function Visibility	Best Practice	Info	Resolved
IDX-016	Potential Economic Attack	Advanced	Info	Resolved

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

5. Detailed Findings Information

5.1. Improper Cap Deduction

ID	IDX-001
Target	LatteMarket
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** High **Impact:** Medium More NFTs can be minted and bought than the cap limit and may cause reputation damage and financial damage if those NFTs can be used in the `Booster` contract. **Likelihood:** High The cap will be incorrectly deducted whenever NFTs are bought in bulk.
Status	Resolved LatteSwap team has resolved this issue as suggested in commit `ffcbc1c6e67e4ecaba407e994f3e5c7ea3970371` by deducting the cap with the amount of NFTs bought.

5.1.1. Description

In the `LatteMarket` contract, the `buyBatchNFT()` function can be used by the users to buy multiple NFTs of the same type in bulk. The `_buyNFTTo()` function is then called with the amount of NFT in the `_size` parameter in line 242.

LatteMarket.sol

```
217 /// @notice buyNFT based on its category id
218 /// @param _nftAddress - nft address
219 /// @param _categoryId - category id for each nft address
220 /// @param _size - amount to buy
221 /// @param _sig - signed signature using message sign
222 function buyBatchNFT(
223     address _nftAddress,
224     uint256 _categoryId,
225     uint256 _size,
226     bytes calldata _sig
227 )
228     external
229     payable
230     whenNotPaused
231     onlySupportedNFT(_nftAddress)
```

```
232     onlyNonBiddingNFT(_nftAddress, _categoryId)
233     permit(_sig)
234 {
235     LatteNFTMetadata memory metadata =
    latteNFTMetadata[_nftAddress][_categoryId];
236     /// re-use a storage usage by using the same metadata to validate
237     /// multiple modifiers can cause stack too deep exception
238     require(
239         block.number >= metadata.startBlock && block.number <=
    metadata.endBlock,
240         "LatteMarket::buyBatchNFT:: invalid block number"
241     );
242     _buyNFTTo(_nftAddress, _categoryId, _msgSender(), _size);
243 }
```

The `_buyNFTTo()` function calls the `_decreaseCap()` once in line 280 to deduct the total cap, preventing the NFT from being bought more than the limited amount.

LatteMarket.sol

```
273 /// @dev internal method for buyNFTTo to avoid stack-too-deep
274 function _buyNFTTo(
275     address _nftAddress,
276     uint256 _categoryId,
277     address _to,
278     uint256 _size
279 ) internal {
280     _decreaseCap(_nftAddress, _categoryId);
281     LatteNFTMetadata memory metadata =
    latteNFTMetadata[_nftAddress][_categoryId];
282     uint256 totalPrice = metadata.price.mul(_size);
283     uint256 feeAmount = totalPrice.mul(feePercentBps).div(1e4);
284     _safeWrap(metadata.quoteBep20, totalPrice);
285     if (feeAmount != 0) {
286         metadata.quoteBep20.safeTransfer(feeAddr, feeAmount);
287     }
```

However, the `_decreaseCap()` function deducts the total cap by just 1 in line 248, even when multiple NFTs are being bought.

LatteMarket.sol

```
245 /// @dev use to decrease a total cap by 1, will get reverted if no more to be
    decreased
246 function _decreaseCap(address _nftAddress, uint256 _categoryId) internal {
247     require(latteNFTMetadata[_nftAddress][_categoryId].cap > 0,
    "LatteMarket::_decreaseCap::maximum mint cap reached");
248     latteNFTMetadata[_nftAddress][_categoryId].cap =
latteNFTMetadata[_nftAddress][_categoryId].cap.sub(1);
}
```

This allows the NFT to be bought and minted more than the predefined cap, and may cause reputation damage to the platform, or financial damage if the NFT can be used in the `Booster` contract.

5.1.2. Remediation

Inspex suggests passing the number of NFTs bought to the `_decreaseCap()` function to deduct the cap correctly, for example:

LatteMarket.sol

```
/// @dev internal method for buyNFTTo to avoid stack-too-deep
function _buyNFTTo(
    address _nftAddress,
    uint256 _categoryId,
    address _to,
    uint256 _size
) internal {
    _decreaseCap(_nftAddress, _categoryId, _size);
    LatteNFTMetadata memory metadata =
latteNFTMetadata[_nftAddress][_categoryId];
    uint256 totalPrice = metadata.price.mul(_size);
    uint256 feeAmount = totalPrice.mul(feePercentBps).div(1e4);
    _safeWrap(metadata.quoteBep20, totalPrice);
    if (feeAmount != 0) {
        metadata.quoteBep20.safeTransfer(feeAddr, feeAmount);
    }
    metadata.quoteBep20.safeTransfer(tokenCategorySellers[_nftAddress]
[_categoryId], totalPrice.sub(feeAmount));
    ILatteNFT(_nftAddress).mintBatch(_to, _categoryId, "", _size);
    emit Trade(
        tokenCategorySellers[_nftAddress][_categoryId],
        _to,
        _nftAddress,
        _categoryId,
        totalPrice,
        feeAmount,
        _size
    );
}
```

In the `_decreaseCap()` function, the cap should be deducted by the value of the passed parameter.

LatteMarket.sol

```
/// @dev use to decrease a total cap by the number of NFT bought, will get reverted if no more to be decreased
function _decreaseCap(address _nftAddress, uint256 _categoryId, uint256 _size) internal {
    require(latteNFTMetadata[_nftAddress][_categoryId].cap >= _size, "LatteMarket::_decreaseCap::maximum mint cap reached");
    latteNFTMetadata[_nftAddress][_categoryId].cap = latteNFTMetadata[_nftAddress][_categoryId].cap.sub(_size);
}
```

5.2. Use of Upgradable Contract

ID	IDX-002
Target	MasterBarista Booster BoosterConfig LatteNFT LatteMarket OGNFT OGOwnerToken
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 confirmed that they will mitigate this issue by implementing the timelock mechanism when deploying the smart contracts to mainnet. The users will be able to monitor the timelock for the upgrade of the contract and act accordingly if it is being misused. At the time of the reassessment, the contracts are not 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.

5.2.1. 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 could be modified by the owner anytime, making the smart contracts untrustworthy.

5.2.2. 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 be notified of the potential changes being done on the smart contracts.

5.3. Unrestricted $LATTE Minting

ID	IDX-003
Target	MasterBarista
Category	Advanced Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity:** High **Impact:** High The contract owner can mint an unlimited amount of $LATTE. **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 confirmed that they will mitigate this issue by implementing the timelock mechanism when deploying the smart contracts to mainnet. The users will be able to monitor the timelock for the upgrade of the contract and act accordingly if it is being misused. At the time of the reassessment, the contracts are not 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.

5.3.1. Description

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

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
) external override onlyStakeTokenCallerContract(_stakeToken) {
    latte.mint(_to, _amount);
    latte.mint(devAddr, _amount.mul(devFeeBps).div(1e4));

    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

```solidity
/// @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 for adding any address to the `stakeTokenCallerContracts` list.

MasterBarista.sol

```solidity
/// @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.

5.3.2. 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 the `Booster` 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 `addStakeTokenCallerContract()` function. This allows the platform users to monitor the timelock and be notified of the potential changes being done on the smart contracts.

5.4. Unrestricted Boosted $LATTE Minting

ID	IDX-004
Target	BoosterConfig
Category	Advanced Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity: High** **Impact: High** The contract owner can set the `boostBps` and the `currentEnergy` states of any NFT to an arbitrary value, allowing the owner to gain profit from the boosted $LATTE reward. **Likelihood: Medium** Only the contract owner can perform this attack; however, there is no restriction preventing the owner from doing it.
Status	**Resolved *** LatteSwap team has confirmed that they will mitigate this issue by implementing the timelock mechanism when deploying the smart contracts to mainnet. The users will be able to monitor the timelock for the upgrade of the contract and act accordingly if it is being misused. At the time of the reassessment, the contracts are not 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.

5.4.1. Description

The `BoosterConfig` contract is used to store and manage the states of the NFT for the calculation of the extra $LATTE reward in the `Booster` and `MasterBarista` contracts.

The `extraReward` is calculated in the `Booster` contract using the configs of the staked NFT from the `BoosterConfig` contract retrieved in line 349. It is calculated by multiplying the `boostBps` state with the original reward, and the reward is capped with the `currentEnergy` state in line 356. This means the higher these values are, the higher extra $LATTE reward that the NFT owner can gain.

Booster.sol

```
340  /// @dev a notifier function for letting some observer call when some
     conditions met
341  /// @dev currently, the caller will be a master barista calling before a latte
     lock
342  function masterBaristaCall(
343      address stakeToken,
344      address userAddr,
    uint256 unboostedReward
) external override inExec {
    NFTStakingInfo memory stakingNFT = userStakingNFT[stakeToken][userAddr];
    UserInfo storage user = userInfo[stakeToken][userAddr];
    (, uint256 currentEnergy, uint256 boostBps) = boosterConfig.energyInfo(
        stakingNFT.nftAddress,
        stakingNFT.nftTokenId
    );
    if (currentEnergy == 0) {
        return;
    }
    uint256 extraReward = MathUpgradeable.min(currentEnergy,
unboostedReward.mul(boostBps).div(1e4));
    totalAccumBoostedReward[stakeToken] =
totalAccumBoostedReward[stakeToken].add(extraReward);
    user.accumBoostedReward = user.accumBoostedReward.add(extraReward);
    uint256 newEnergy = currentEnergy.sub(extraReward);
    masterBarista.mintExtraReward(stakeToken, userAddr, extraReward);
    boosterConfig.updateCurrentEnergy(stakingNFT.nftAddress,
stakingNFT.nftTokenId, newEnergy);

    emit MasterBaristaCall(userAddr, extraReward, stakeToken, currentEnergy,
newEnergy);
}
```

The boostBps state of the NFTs can be set by using the `_setBoosterNFTEnergyInfo()` function which can be called by the contract owner using the `setBatchBoosterNFTEnergyInfo()` and the `setBoosterNFTEnergyInfo()` functions.

BoosterConfig.sol

```
/// @dev An internal function for setting booster NFT energy info
/// @param _param a BoosterNFTParams {nftAddress, nftTokenId, maxEnergy,
boostBps}
function _setBoosterNFTEnergyInfo(BoosterNFTParams calldata _param) internal {
    _boosterEnergyInfo[_param.nftAddress][_param.nftTokenId] =
BoosterEnergyInfo({
        maxEnergy: _param.maxEnergy,
        currentEnergy: _param.maxEnergy,
        boostBps: _param.boostBps,
        updatedAt: block.timestamp
    });

    emit SetBoosterNFTEnergyInfo(
        _param.nftAddress,
        _param.nftTokenId,
        _param.maxEnergy,
        _param.maxEnergy,
        _param.boostBps
    );
}
```

BoosterConfig.sol

```
/// @notice A function for setting booster NFT energy info as a batch
/// @param _params a list of BoosterNFTParams [{nftAddress, nftTokenId,
maxEnergy, boostBps}]
function setBatchBoosterNFTEnergyInfo(BoosterNFTParams[] calldata _params)
external onlyOwner {
    for (uint256 i = 0; i < _params.length; ++i) {
        _setBoosterNFTEnergyInfo(_params[i]);
    }
}

/// @notice A function for setting booster NFT energy info
/// @param _param a BoosterNFTParams {nftAddress, nftTokenId, maxEnergy,
boostBps}
function setBoosterNFTEnergyInfo(BoosterNFTParams calldata _param) external
onlyOwner {
    _setBoosterNFTEnergyInfo(_param);
}
```

It can also be set using the `_setCategoryNFTEnergyInfo()` function which can be called by the owner using the `setBatchCategoryNFTEnergyInfo()`, and the `setCategoryNFTEnergyInfo()` functions.

BoosterConfig.sol

```
/// @dev An internal function for setting category NFT energy info, used for
nft with non-preminted
/// @param _param a CategoryNFTParams {nftAddress, nftCategoryId, maxEnergy,
boostBps}
function _setCategoryNFTEnergyInfo(CategoryNFTParams calldata _param) internal
{
    _categoryEnergyInfo[_param.nftAddress][_param.nftCategoryId] =
CategoryEnergyInfo({
        maxEnergy: _param.maxEnergy,
        boostBps: _param.boostBps,
        updatedAt: block.timestamp
    });

    emit SetCategoryNFTEnergyInfo(_param.nftAddress, _param.nftCategoryId,
_param.maxEnergy, _param.boostBps);
}
```

BoosterConfig.sol

```
234 /// @notice A function for setting category NFT energy info as a batch, used for nft with non-preminted
235 /// @param _params a list of CategoryNFTParams [{nftAddress, nftTokenId, maxEnergy, boostBps}]
236 function setBatchCategoryNFTEnergyInfo(CategoryNFTParams[] calldata _params) external onlyOwner {
237     for (uint256 i = 0; i < _params.length; ++i) {
238         _setCategoryNFTEnergyInfo(_params[i]);
239     }
240 }
241
242 /// @notice A function for setting category NFT energy info, used for nft with non-preminted
243 /// @param _param a CategoryNFTParams {nftAddress, nftTokenId, maxEnergy, boostBps}
244 function setCategoryNFTEnergyInfo(CategoryNFTParams calldata _param) external onlyOwner {
245     _setCategoryNFTEnergyInfo(_param);
246 }
```

These functions allow the contract owner to set `boostBps` to any arbitrary number, including a massively high value.

Furthermore, the `currentEnergy` state of the NFTs can be updated by using the `updateCurrentEnergy()` function in line 173.

BoosterConfig.sol

```
152 /// @notice function for updating a curreny energy of the specified nft
153 /// @dev Only eligible caller can freely update an energy
154 /// @param _nftAddress a composite key for nft
155 /// @param _nftTokenId a composite key for nft
156 /// @param _updatedCurrentEnergy an updated curreny energy for the nft
157 function updateCurrentEnergy(
158     address _nftAddress,
159     uint256 _nftTokenId,
160     uint256 _updatedCurrentEnergy
161 ) external override onlyCaller {
162     require(_nftAddress != address(0), "BoosterConfig::updateCurrentEnergy::_nftAddress must not be address(0)");
163     BoosterEnergyInfo storage energy = _boosterEnergyInfo[_nftAddress][_nftTokenId];
164
165     if (energy.updatedAt == 0) {
166         uint256 categoryId = ILatteNFT(_nftAddress).latteNFTToCategory(_nftTokenId);
```

```
        CategoryEnergyInfo memory categoryEnergy = _categoryEnergyInfo[_nftAddress][categoryId];
        require(categoryEnergy.updatedAt != 0, "BoosterConfig::updateCurrentEnergy:: invalid nft to be updated");
        energy.maxEnergy = categoryEnergy.maxEnergy;
        energy.boostBps = categoryEnergy.boostBps;
    }

    energy.currentEnergy = _updatedCurrentEnergy;
    energy.updatedAt = block.timestamp;

    emit UpdateCurrentEnergy(_nftAddress, _nftTokenId, _updatedCurrentEnergy);
}
```

The caller of the `updateCurrentEnergy()` function is checked by the `onlyCaller` modifier, allowing only the addresses whitelisted in the `callerAllowance` mapping.

BoosterConfig.sol

```
/// @notice only eligible caller can continue the execution
modifier onlyCaller() {
    require(callerAllowance[msg.sender], "BoosterConfig::onlyCaller::only eligible caller");
    _;
}
```

The contract owner can use the `setCallerAllowance()` function to add any address to the `callerAllowance` mapping.

BoosterConfig.sol

```
/// @notice set caller allowance - only eligible caller can call a function
/// @dev only eligible callers can call this function
/// @param _caller a specified caller
/// @param _isAllowed a flag indicating the allowance of a specified token
function setCallerAllowance(address _caller, bool _isAllowed) external onlyOwner {
    require(_caller != address(0), "BoosterConfig::setCallerAllowance::_caller must not be address(0)");
    callerAllowance[_caller] = _isAllowed;

    emit SetCallerAllowance(_caller, _isAllowed);
}
```

This means that the contract owner can add the owner's wallet address to the list and freely use the `updateCurrentEnergy()` function to set the `currentEnergy` of any NFT to an arbitrary value.

With the control of both `boostBps` and `currentEnergy` NFT state variables, the contract owner can abuse this feature to mint a massive amount of $LATTE and gain the profit.

5.4.2. 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 the `Booster` 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 `onlyOwner` functions of `BoosterConfig` contract. This allows the platform users to monitor the timelock and be notified of the potential changes being done on the smart contracts.

5.5. Improper NFT Burning

ID	IDX-005
Target	LatteNFT
Category	Advanced Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity:** High **Impact:** High The contract owner can burn any user's NFT at any time, which can cause financial damage to the user. **Likelihood:** Medium Only the contract owner can perform this attack; however, there is no restriction preventing the owner from doing it.
Status	Resolved LatteSwap team has resolved this issue as suggested in commit `b7cd5485c1778c921288ada671b93370601c331e` by removing the `burn()` function.

5.5.1. Description

In the `LatteNFT` contract, the `burn()` function can be used to burn any user's `LatteNFT`.

LatteNFT.sol

```
/**
 * @dev Burn a NFT token. Callable by owner only.
 */
function burn(uint256 _tokenId) external onlyMinter {
    uint256 categoryId = latteNFTToCategory[_tokenId];
    require(_categoryToLatteNFTList[categoryId].remove(_tokenId),
"LatteNFT::burn::tokenId not found");
    // Clear metadata (if any)
    if (bytes(_tokenURIs[_tokenId]).length != 0) {
        delete _tokenURIs[_tokenId];
    }
    _burn(_tokenId);
}
```

The function can be called by the addresses with the minter role.

LatteNFT.sol

```
/// @dev only the one having a MINTER_ROLE can continue an execution
modifier onlyMinter() {
    require(hasRole(MINTER_ROLE, _msgSender()), "LatteNFT::onlyMinter::only MINTER role");
    _;
}
```

Since the contract owner has the minter role and can grant the minter role to anyone, the contract owner can use the `burn()` function freely.

LatteNFT.sol

```
function initialize(string memory _baseURI) public initializer {
    ERC721Upgradeable.__ERC721_init("LATTE", "LATTE NFT");
    ERC721PausableUpgradeable.__ERC721Pausable_init();
    OwnableUpgradeable.__Ownable_init();
    AccessControlUpgradeable.__AccessControl_init();

    _setupRole(DEFAULT_ADMIN_ROLE, _msgSender());
    _setupRole(GOVERNANCE_ROLE, _msgSender());
    _setupRole(MINTER_ROLE, _msgSender());
    _setBaseURI(_baseURI);
}
```

The NFT can be bought and owned by the users, so it is unfair for the owner to be able to burn the NFT owned by other users.

5.5.2. Remediation

Inspex suggests removing the `burn()` function from the `LatteNFT` contract.

5.6. Division Before Multiplication

ID	IDX-006
Target	MasterBarista
Category	General Smart Contract Vulnerability
CWE	CWE-682: Incorrect Calculation
Risk	**Severity:** High **Impact:** Medium The allocation point of pools with allocBps will be lower than the expected amount, causing the users of these pools to gain less reward. **Likelihood:** High It is very likely for the rounding error to occur.
Status	Resolved LatteSwap team has resolved this issue as suggested in commit `baeb1694e5ee18fa6bbb69e32701bb434a74ad7e` by performing multiplication before division.

5.6.1. Description

In the `MasterBarista` contract, the `_updatePoolAlloc()` function is used to update the `allocPoint` of the pools with `allocBps`. However, `adjustedPoints` is calculated first by dividing `num` with `denom` in line 362. As `adjustedPoints` is used further in the calculation of `poolPoints`, dividing it first can cause significant miscalculations from the rounding error.

MasterBarista.sol

```
354  // @dev internal function for updating pool based on accumulated bps and points
355  function _updatePoolAlloc(uint256 _accumAllocBps, uint256
     _accumNonBpsPoolPoints) internal {
356      // n = kp/(1-k),
357      // where  k is accumAllocBps
358      // p is sum of points of other pools
359      address curr = pools.next[LinkList.start];
360      uint256 num = _accumNonBpsPoolPoints.mul(_accumAllocBps);
361      uint256 denom = uint256(10000).sub(_accumAllocBps);
362      uint256 adjustedPoints = num.div(denom);
363      uint256 poolPoints;
364      while (curr != LinkList.end) {
365          if (poolInfo[curr].allocBps == 0) {
366              curr = pools.getNextOf(curr);
367              continue;
368          }
```

```
        poolPoints = adjustedPoints.mul(poolInfo[curr].allocBps).div(_accumAllocBps);
        totalAllocPoint = totalAllocPoint.sub(poolInfo[curr].allocPoint).add(poolPoints);
        poolInfo[curr].allocPoint = poolPoints;
        emit PoolAllocChanged(curr, poolInfo[curr].allocBps, poolPoints);
        curr = pools.getNextOf(curr);
    }
}
```

To demonstrate the impact, please see the following examples, assuming that:

Case 1:

```
_accumAllocBps = 1000
_accumNonBpsPoolPoints = 10
```

The calculation of `adjustedPoints` will be as follows:

```
num = _accumNonBpsPoolPoints * _accumAllocBps = 10 * 1000 = 10000
denom = 10000 - _accumAllocBps = 10000 - 1000 = 9000
adjustedPoints = num / denom = 10000 / 9000 = 1.1111
```

Case 2:

```
_accumAllocBps = 4000
_accumNonBpsPoolPoints = 20
```

The calculation of `adjustedPoints` will be as follows:

```
num = _accumNonBpsPoolPoints * _accumAllocBps = 20 * 4000 = 8000
denom = 10000 - _accumAllocBps = 10000 - 4000 = 6000
adjustedPoints = num / denom = 8000 / 6000 = 1.3333
```

However, as decimal numbers are not fully supported by Solidity, `adjustedPoints` will be equal to 1 in both cases, and can cause significant deviations from the correct value.

5.6.2. Remediation

Inspex suggests calculating the `poolPoints` with `num` and `denom` variables instead of dividing first, for example:

MasterBarista.sol

```
// @dev internal function for updating pool based on accumulated bps and points
function _updatePoolAlloc(uint256 _accumAllocBps, uint256 _accumNonBpsPoolPoints) internal {
    // n = kp/(1-k),
    // where  k is accumAllocBps
    // p is sum of points of other pools
    address curr = pools.next[LinkList.start];
    uint256 num = _accumNonBpsPoolPoints.mul(_accumAllocBps);
    uint256 denom = uint256(10000).sub(_accumAllocBps);
    uint256 poolPoints;
    while (curr != LinkList.end) {
        if (poolInfo[curr].allocBps == 0) {
            curr = pools.getNextOf(curr);
            continue;
        }
        poolPoints = num.mul(poolInfo[curr].allocBps).div(_accumAllocBps.mul(denom));
        totalAllocPoint = totalAllocPoint.sub(poolInfo[curr].allocPoint).add(poolPoints);
        poolInfo[curr].allocPoint = poolPoints;
        emit PoolAllocChanged(curr, poolInfo[curr].allocBps, poolPoints);
        curr = pools.getNextOf(curr);
    }
}
```

5.7. Centralized Control of State Variable

ID	IDX-007
Target	MasterBarista BoosterConfig LatteMarket
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 potentially nothing to restrict the changes from being done by the owner; however, the changes are limited by fixed values in the smart contracts.
Status	Resolved * LatteSwap team has confirmed that they will implement the timelock mechanism when deploying the smart contracts to mainnet. The users will be able to monitor the timelock for the execution of critical functions and act accordingly if they are being misused. At the time of the reassessment, the contracts are not 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.

5.7.1. Description

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

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

The controllable privileged state update functions are as follows:

File	Contract	Function	Modifier
BoosterConfig.sol (L:183)	BoosterConfig	setStakeTokenAllowance()	onlyOwner
BoosterConfig.sol (L:194)	BoosterConfig	setCallerAllowance()	onlyOwner
BoosterConfig.sol (L:203)	BoosterConfig	setBatchBoosterNFTEnergyInfo()	onlyOwner

BoosterConfig.sol (L:211)	BoosterConfig	setBoosterNFTEnergyInfo()	onlyOwner
BoosterConfig.sol (L:236)	BoosterConfig	setBatchCategoryNFTEnergyInfo()	onlyOwner
BoosterConfig.sol (L:244)	BoosterConfig	setCategoryNFTEnergyInfo()	onlyOwner
BoosterConfig.sol (L:283)	BoosterConfig	setStakingTokenBoosterAllowance()	onlyOwner
MasterBarista.sol (L:162)	MasterBarista	setStakeTokenCallerAllowancePool()	onlyOwner
MasterBarista.sol (L:171)	MasterBarista	addStakeTokenCallerContract()	onlyOwner
MasterBarista.sol (L:187)	MasterBarista	removeStakeTokenCallerContract()	onlyOwner
MasterBarista.sol (L:207)	MasterBarista	setLattePerBlock()	onlyOwner
MasterBarista.sol (L:215)	MasterBarista	setPoolAllocBps()	onlyOwner
MasterBarista.sol (L:240)	MasterBarista	setBonus()	onlyOwner
MasterBarista.sol (L:261)	MasterBarista	addPool()	onlyOwner
MasterBarista.sol (L:289)	MasterBarista	setPool()	onlyOwner
MasterBarista.sol (L:311)	MasterBarista	removePool()	onlyOwner
LatteMarket.sol (L:169)	LatteMarket	setLatteNFTMetadata()	onlyOwner
LatteMarket.sol (L:476)	LatteMarket	setTransferFeeAddress()	onlyOwner
LatteMarket.sol (L:482)	LatteMarket	setFeePercent()	onlyOwner

5.7.2. 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

5.8. Improper Token Burning

ID	IDX-008
Target	MasterBarista
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** High The users will not be able to use the `emergencyWithdraw()` function through the funder contracts in an emergency situation. **Likelihood:** Low It is unlikely for the `emergencyWithdraw()` function to be used.
Status	Resolved LatteSwap team has resolved this issue as suggested in commit `92280788b76abd2a57a2a0cfafed16574f3bc1bd` by burning $BEAN from the beneficiary instead of the message sender.

5.8.1. Description

In the `MasterBarista` contract, $BEAN is minted for the beneficiary (`_for`) of those who deposit $LATTE to the contract in line 563.

MasterBarista.sol

```
539 /// @dev Deposit LATTE to get even more LATTE.
540 /// @param _amount The amount to be deposited
541 function depositLatte(address _for, uint256 _amount)
542     external
543     override
544     onlyPermittedTokenFunder(_for, address(latte))
545     nonReentrant
546 {
547     PoolInfo storage pool = poolInfo[address(latte)];
548     UserInfo storage user = userInfo[address(latte)][_for];
549
550     if (user.fundedBy != address(0)) require(user.fundedBy == _msgSender(),
    "MasterBarista::depositLatte::bad sof");
551
552     updatePool(address(latte));
553
554     if (user.amount > 0) _harvest(_for, address(latte));
555     if (user.fundedBy == address(0)) user.fundedBy = _msgSender();
556     if (_amount > 0) {
557         IERC20(address(latte)).safeTransferFrom(address(_msgSender()), address(this), _amount);
558         user.amount = user.amount.add(_amount);
559     }
560     user.rewardDebt = user.amount.mul(pool.accLattePerShare).div(1e12);
561     user.bonusDebt = user.amount.mul(pool.accLattePerShareTilBonusEnd).div(1e12);
562
563     bean.mint(_for, _amount);
564
565     emit Deposit(_msgSender(), _for, address(latte), _amount);
566 }
```

However, in the `emergencyWithdraw()` function, $BEAN is burned from the `_msgSender()` address in line 670, not the beneficiary (`_for`).

```
658 /// @dev Withdraw without caring about rewards. EMERGENCY ONLY.
659 /// @param _for if the msg sender is a funder, can emergency withdraw a fundee
660 /// @param _stakeToken The pool's stake token
661 function emergencyWithdraw(address _for, address _stakeToken) external override nonReentrant {
662     UserInfo storage user = userInfo[_stakeToken][_for];
663     require(user.fundedBy == _msgSender(), "MasterBarista::emergencyWithdraw::only funder");
664     IERC20(_stakeToken).safeTransfer(address(_for), user.amount);
665
666     emit EmergencyWithdraw(_for, _stakeToken, user.amount);
667
668     // Burn BEAN if user emergencyWithdraw LATTE
669     if (_stakeToken == address(latte)) {
670         bean.burn(_msgSender(), user.amount);
671     }
672
673     // Reset user info
674     user.amount = 0;
675     user.rewardDebt = 0;
676     user.bonusDebt = 0;
677     user.fundedBy = address(0);
678 }
```

This causes the `emergencyWithdraw()` function to be unusable for the users who deposit $LATTE through funder contracts without transferring $BEAN to the funder contracts by themselves.

5.8.2. Remediation

Inspex suggests editing the `emergencyWithdraw()` function to burn $BEAN from the beneficiary (`_for`) instead of `_msgSender()`, for example:

```
/// @dev Withdraw without caring about rewards. EMERGENCY ONLY.
/// @param _for if the msg sender is a funder, can emergency withdraw a fundee
/// @param _stakeToken The pool's stake token
function emergencyWithdraw(address _for, address _stakeToken) external override
nonReentrant {
    UserInfo storage user = userInfo[_stakeToken][_for];
    require(user.fundedBy == _msgSender(),
"MasterBarista::emergencyWithdraw::only funder");
    IERC20(_stakeToken).safeTransfer(address(_for), user.amount);

    emit EmergencyWithdraw(_for, _stakeToken, user.amount);

    // Burn BEAN if user emergencyWithdraw LATTE
    if (_stakeToken == address(latte)) {
        bean.burn(_for, user.amount);
    }

    // Reset user info
    user.amount = 0;
    user.rewardDebt = 0;
    user.bonusDebt = 0;
    user.fundedBy = address(0);
}
```

5.9. Unchecked Max Value

ID	IDX-009
Target	BoosterConfig
Category	Advanced Smart Contract Vulnerability
CWE	CWE-20: Improper Input Validation
Risk	**Severity:** Medium **Impact:** Medium The energy level of the booster NFT can be set to the value exceeding the max energy defined, allowing the caller to use that NFT to gain unfair profit. **Likelihood:** Medium Only the addresses whitelisted by the contract owner can perform this attack; however, there is no restriction preventing them from doing it.
Status	Resolved LatteSwap team has resolved this issue in commit `b808074b17dfd30b1a4a58402e4c2a70e091a16f` by changing from updating an exact energy to deducting energy from the current energy.

5.9.1. Description

The `updateCurrentEnergy()` can be used to update the `currentEnergy` state of an NFT in line 173. The energy can be used to boost the farming reward in the `MasterBarista` contract via the `Booster` contract.

BoosterConfig.sol

```
152 /// @notice function for updating a curreny energy of the specified nft
153 /// @dev Only eligible caller can freely update an energy
154 /// @param _nftAddress a composite key for nft
155 /// @param _nftTokenId a composite key for nft
156 /// @param _updatedCurrentEnergy an updated curreny energy for the nft
157 function updateCurrentEnergy(
158     address _nftAddress,
159     uint256 _nftTokenId,
160     uint256 _updatedCurrentEnergy
161 ) external override onlyCaller {
162     require(_nftAddress != address(0),
    "BoosterConfig::updateCurrentEnergy::_nftAddress must not be address(0)");
163     BoosterEnergyInfo storage energy =
    _boosterEnergyInfo[_nftAddress][_nftTokenId];
164
165     if (energy.updatedAt == 0) {
166         uint256 categoryId =
ILatteNFT(_nftAddress).latteNFTToCategory(_nftTokenId);
        CategoryEnergyInfo memory categoryEnergy =
_categoryEnergyInfo[_nftAddress][categoryId];
        require(categoryEnergy.updatedAt != 0,
"BoosterConfig::updateCurrentEnergy:: invalid nft to be updated");
        energy.maxEnergy = categoryEnergy.maxEnergy;
        energy.boostBps = categoryEnergy.boostBps;
    }

    energy.currentEnergy = _updatedCurrentEnergy;
    energy.updatedAt = block.timestamp;

    emit UpdateCurrentEnergy(_nftAddress, _nftTokenId, _updatedCurrentEnergy);
}
```

However, even if there's the `maxEnergy` variable, it is not checked on the update, allowing the caller to set the energy to an arbitrary value and gain profit from that NFT.

5.9.2. Remediation

Inspex suggests checking the `maxEnergy` of the NFT on the update of energy, for example:

BoosterConfig.sol

```
/// @notice function for updating a curreny energy of the specified nft
/// @dev Only eligible caller can freely update an energy
/// @param _nftAddress a composite key for nft
/// @param _nftTokenId a composite key for nft
/// @param _updatedCurrentEnergy an updated curreny energy for the nft
function updateCurrentEnergy(
    address _nftAddress,
    uint256 _nftTokenId,
    uint256 _updatedCurrentEnergy
) external override onlyCaller {
    require(_nftAddress != address(0),
"BoosterConfig::updateCurrentEnergy::_nftAddress must not be address(0)");
    BoosterEnergyInfo storage energy =
_boosterEnergyInfo[_nftAddress][_nftTokenId];

    if (energy.updatedAt == 0) {
        uint256 categoryId =
ILatteNFT(_nftAddress).latteNFTToCategory(_nftTokenId);
        CategoryEnergyInfo memory categoryEnergy =
_categoryEnergyInfo[_nftAddress][categoryId];
        require(categoryEnergy.updatedAt != 0,
"BoosterConfig::updateCurrentEnergy:: invalid nft to be updated");
        energy.maxEnergy = categoryEnergy.maxEnergy;
        energy.boostBps = categoryEnergy.boostBps;
    }
    require(_updatedCurrentEnergy < energy.maxEnergy, "BoosterConfig::updateCurrentEnergy:: _updatedCurrentEnergy energy exceeds maxEnergy");
    energy.currentEnergy = _updatedCurrentEnergy;
    energy.updatedAt = block.timestamp;

    emit UpdateCurrentEnergy(_nftAddress, _nftTokenId, _updatedCurrentEnergy);
}
```

5.10. Auction Cancellation

ID	IDX-010
Target	LatteMarket
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Low **Impact:** Low The governance can cancel the auction even after the user has bid for the NFT, preventing the user from getting the NFT that the user should be eligible for, making it unfair for the bidding user and resulting in a loss of reputation for the platform. **Likelihood:** Medium Only the addresses with the governance role can perform this action; however, there is no restriction to prevent them from doing it.
Status	Resolved LatteSwap team has resolved this issue as suggested in commit `be2dfa60426f2d7eb55b4d7e002fa72aef90a6eb` by reverting the cancel transaction if there is an existing bidder.

5.10.1. Description

In the `LatteMarket` contract, the `readyToStartAuction()` function can be used by the governance to start an auction, allowing the users to bid on the NFT during the auction time (from `_startBlock` to `_endBlock`).

LatteMarket.sol

```
/// @notice this needs to be called when the seller want to start AUCTION the token
/// @param _nftAddress - nft address
/// @param _categoryId - category id for each nft address
/// @param _price - starting price of a token
/// @param _cap - total cap for this nft address with a category id
/// @param _startBlock - starting block for a sale
/// @param _endBlock - end block for a sale
function readyToStartAuction(
    address _nftAddress,
    uint256 _categoryId,
    uint256 _price,
    uint256 _cap,
    uint256 _startBlock,
    uint256 _endBlock,
    IERC20Upgradeable _quoteToken
) external whenNotPaused onlySupportedNFT(_nftAddress) onlyGovernance {
    latteNFTMetadata[_nftAddress][_categoryId].isBidding = true;
    _readyToSellNFTTo(
        _nftAddress,
        _categoryId,
        _price,
        address(_msgSender()),
        _cap,
        _startBlock,
        _endBlock,
        _quoteToken
    );
}
```

The auction can be cancelled using the `cancelBiddingNFT()` function. However, there is no checking whether anyone has bid for the NFT or not. This means that there is a potential winner eligible as a "to-be" owner of the NFT being auctioned, and cancelling it means that the user will not get that NFT as intended.

LatteMarket.sol

```
/// @notice cancel a bidding token, similar to cancel sell, with
functionalities to return bidding amount back to the user
/// @param _nftAddress - nft address
/// @param _categoryId - category id for each nft address
function cancelBiddingNFT(address _nftAddress, uint256 _categoryId)
    external
    whenNotPaused
    onlySupportedNFT(_nftAddress)
    onlyGovernance
    onlyBiddingNFT(_nftAddress, _categoryId)
{
    BidEntry memory toBeReturned = tokenBid[_nftAddress][_categoryId];
    IERC20Upgradeable returnedQuoteBep20 =
latteNFTMetadata[_nftAddress][_categoryId].quoteBep20;
    _delBidByCompositeId(_nftAddress, _categoryId);
    _cancelSellNFT(_nftAddress, _categoryId);
    if (toBeReturned.bidder != address(0)) {
        _safeUnwrap(returnedQuoteBep20, toBeReturned.bidder,
toBeReturned.price);
    }
    emit CancelBidNFT(toBeReturned.bidder, _nftAddress, _categoryId);
}
```

5.10.2. Remediation

Inspex suggests checking the bidder of the auction to prevent the cancellation of an auction with a bidder, for example:

LatteMarket.sol

```
/// @notice cancel a bidding token, similar to cancel sell, with functionalities to return bidding amount back to the user
/// @param _nftAddress - nft address
/// @param _categoryId - category id for each nft address
function cancelBiddingNFT(address _nftAddress, uint256 _categoryId)
    external
    whenNotPaused
    onlySupportedNFT(_nftAddress)
    onlyGovernance
    onlyBiddingNFT(_nftAddress, _categoryId)
{
    BidEntry memory bidEntry = tokenBid[_nftAddress][_categoryId];
    require(bidEntry.bidder == address(0), "LatteMarket::cancelBiddingNFT::auction already has a bidder");
    IERC20Upgradeable returnedQuoteBep20 = latteNFTMetadata[_nftAddress][_categoryId].quoteBep20;
    _delBidByCompositeId(_nftAddress, _categoryId);
    _cancelSellNFT(_nftAddress, _categoryId);
    emit CancelBidNFT(_nftAddress, _categoryId);
}
```

5.11. Improper Minimum allocBps Condition

ID	IDX-011
Target	MasterBarista
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Low **Impact:** Medium A pool with `allocBps` set to a value more than zero cannot be set back to zero, preventing the pool reward from being paused without removing the pool. **Likelihood:** Low It is unlikely for the allocation point of each pool to be changed.
Status	Resolved LatteSwap team has resolved this issue as suggested in commit `298edf9c4f5f27c89889049bca1fcab8ac166b8a` by removing the minimum `_allocBps` condition and reset the `allocPoint` when the bps is set to 0.

5.11.1. Description

In the `MasterBarista` contract, there are two ways to assign the allocation points for each pool: setting a static `allocPoint` and setting a fixed bps using `allocBps`.

MasterBarista.sol

```
// Info of each pool.
struct PoolInfo {
    uint256 allocPoint; // How many allocation points assigned to this pool.
    uint256 lastRewardBlock; // Last block number that LATTE distribution occurs.
    uint256 accLattePerShare; // Accumulated LATTE per share, times 1e12. See below.
    uint256 accLattePerShareTilBonusEnd; // Accumated LATTE per share until Bonus End.
    uint256 allocBps; // Pool allocation in BPS, if it's not a fixed bps pool, leave it 0
}
```

Whenever the total allocation point is updated, the `updatePoolsAlloc()` is called to get the sum of the `allocPoint` and `allocBps`. The `allocPoint` of the pools with more than zero `allocBps` will be ignored in line 338-342. After getting the accumulated values, the `_updatePoolAlloc()` function is called.

MasterBarista.sol

```
/// @dev Update pools' alloc point
function updatePoolsAlloc() internal {
    address curr = pools.next[LinkList.start];
    uint256 points = 0;
    uint256 accumAllocBps = 0;
    while (curr != LinkList.end) {
        if (poolInfo[curr].allocBps > 0) {
            accumAllocBps = accumAllocBps.add(poolInfo[curr].allocBps);
            curr = pools.getNextOf(curr);
            continue;
        }

        points = points.add(poolInfo[curr].allocPoint);
        curr = pools.getNextOf(curr);
    }

    // re-adjust an allocpoints for those pool having an allocBps
    if (points != 0) {
        _updatePoolAlloc(accumAllocBps, points);
    }
}
```

The `_updatePoolAlloc()` function updates the `allocPoint` of the pools by calculating from their `allocBps`, resulting in a static reward allocation for these pools.

MasterBarista.sol

```
// @dev internal function for updating pool based on accumulated bps and points
function _updatePoolAlloc(uint256 _accumAllocBps, uint256 _accumNonBpsPoolPoints) internal {
    // n = kp/(1-k),
    // where  k is accumAllocBps
    // p is sum of points of other pools
    address curr = pools.next[LinkList.start];
    uint256 num = _accumNonBpsPoolPoints.mul(_accumAllocBps);
    uint256 denom = uint256(10000).sub(_accumAllocBps);
    uint256 adjustedPoints = num.div(denom);
    uint256 poolPoints;
    while (curr != LinkList.end) {
        if (poolInfo[curr].allocBps == 0) {
            curr = pools.getNextOf(curr);
            continue;
        }
        poolPoints = adjustedPoints.mul(poolInfo[curr].allocBps).div(_accumAllocBps);
        totalAllocPoint =
    totalAllocPoint.sub(poolInfo[curr].allocPoint).add(poolPoints);
371         poolInfo[curr].allocPoint = poolPoints;
372         emit PoolAllocChanged(curr, poolInfo[curr].allocBps, poolPoints);
373         curr = pools.getNextOf(curr);
374     }
375 }
```

The `setPoolAllocBps()` function can be used to set the `allocBps` of a pool. However, the condition at line 221 checks that the new `_allocBps` must be more than 1000, and therefore, the pools with `allocBps` cannot have it set back to 0. This prevents the contract owner from pausing the reward of this pool without removing the pool.

MasterBarista.sol

```
212 /// @dev Set a specified pool's alloc BPS
213 /// @param _allocBps The new alloc Bps
214 /// @param _stakeToken pid
215 function setPoolAllocBps(address _stakeToken, uint256 _allocBps) external
    onlyOwner {
216     require(
217         _stakeToken != address(0) && _stakeToken != address(1),
218         "MasterBarista::setPoolAllocBps::_stakeToken must not be address(0) or
    address(1)"
        );
219     require(pools.has(_stakeToken), "MasterBarista::setPoolAllocBps::pool
220 hasn't been set");
        require(_allocBps > 1000, "MasterBarista::setPoolallocBps::_allocBps must >
221 1000");
222     address curr = pools.next[LinkList.start];
223     uint256 accumAllocBps = 0;
224     while (curr != LinkList.end) {
225         if (poolInfo[curr].allocBps > 0) {
226             accumAllocBps = accumAllocBps.add(poolInfo[curr].allocBps);
227         }
228         curr = pools.getNextOf(curr);
229     }
        require(accumAllocBps.add(_allocBps) < 10000,
230 "MasterBarista::setPoolallocBps::accumAllocBps must < 10000");
231     massUpdatePools();
232     poolInfo[_stakeToken].allocBps = _allocBps;
233     updatePoolsAlloc();
234 }
```

5.11.2. Remediation

Inspex suggests removing the `_allocBps > 1000` assertion and reset the `allocPoint` of the pool whenever the `allocBps` is set to zero, for example:

MasterBarista.sol

```
/// @dev Set a specified pool's alloc BPS
/// @param _allocBps The new alloc Bps
/// @param _stakeToken pid
function setPoolAllocBps(address _stakeToken, uint256 _allocBps) external onlyOwner {
    require(
        _stakeToken != address(0) && _stakeToken != address(1),
        "MasterBarista::setPoolAllocBps::_stakeToken must not be address(0) or address(1)"
    );
    require(pools.has(_stakeToken), "MasterBarista::setPoolAllocBps::pool hasn't been set");
    address curr = pools.next[LinkList.start];
    uint256 accumAllocBps = 0;
    while (curr != LinkList.end) {
        if (poolInfo[curr].allocBps > 0) {
            accumAllocBps = accumAllocBps.add(poolInfo[curr].allocBps);
        }
        curr = pools.getNextOf(curr);
    }
    require(accumAllocBps.add(_allocBps) < 10000, "MasterBarista::setPoolallocBps::accumAllocBps must < 10000");
    if (_allocBps == 0) {
      totalAllocPoint = totalAllocPoint.sub(poolInfo[_stakeToken].allocPoint);
      poolInfo[_stakeToken].allocPoint = 0;
    }
    massUpdatePools();
    poolInfo[_stakeToken].allocBps = _allocBps;
    updatePoolsAlloc();
}
```

Please note that the remediations for other issues are not yet applied to the example above.

5.12. Improper Maximum accumAllocBps Condition

ID	IDX-012
Target	MasterBarista
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Low **Impact:** Medium The accumulated `allocBps` is incorrectly validated and may prevent the pool allocation from being updated properly. **Likelihood:** Low It is unlikely for the allocation point of each pool to be changed.
Status	Resolved LatteSwap team has resolved this issue as suggested in commit `30d9335c10a8c9a2f12db9fd6a18bf0ed6ba0f2a` by excluding the previous value of `allocBps` from the calculation.

5.12.1. Description

In the `MasterBarista` contract, the `setPoolAllocBps()` function can be used to set the `allocBps` of a pool. However, the condition at line 230 checks that the existing accumulated `allocBps` added with the new `_allocBps` must not exceed 10000 without considering the previous value of the pool's `allocBps`.

MasterBarista.sol

```
212 /// @dev Set a specified pool's alloc BPS
213 /// @param _allocBps The new alloc Bps
214 /// @param _stakeToken pid
215 function setPoolAllocBps(address _stakeToken, uint256 _allocBps) external onlyOwner {
216     require(
217         _stakeToken != address(0) && _stakeToken != address(1),
218         "MasterBarista::setPoolAllocBps::_stakeToken must not be address(0) or address(1)"
219     );
220     require(pools.has(_stakeToken), "MasterBarista::setPoolAllocBps::pool hasn't been set");
221     require(_allocBps > 1000, "MasterBarista::setPoolallocBps::_allocBps must > 1000");
222     address curr = pools.next[LinkList.start];
223     uint256 accumAllocBps = 0;
224     while (curr != LinkList.end) {
```

```
225         if (poolInfo[curr].allocBps > 0) {
226             accumAllocBps = accumAllocBps.add(poolInfo[curr].allocBps);
227         }
228         curr = pools.getNextOf(curr);
229     }
230     require(accumAllocBps.add(_allocBps) < 10000,
"MasterBarista::setPoolallocBps::accumAllocBps must < 10000");
231     massUpdatePools();
232     poolInfo[_stakeToken].allocBps = _allocBps;
233     updatePoolsAlloc();
234 }
```

To demonstrate the impact, please see the following example case, assuming that:

```
Pool A allocBps = 2500
Pool B allocBps = 2500
Pool C allocBps = 2500
Pool D allocBps = 2000
```

Updating Pool D's `allocBps` to 2300 should be allowed, as the total `allocBps` will be equal to 2500 + 2500 + 2500 + 2300 = 9800, which does not exceed 10000.

However, this case will be reverted in the `setPoolAllocBps()` function since `accumAllocBps` will be equal to 2500 + 2500 + 2500 + 2000 = 9500, and `accumAllocBps.add(_allocBps)` in line 230 will be equal to 9500 + 2300 = 11800, which exceeds 10000.

5.12.2. Remediation

Inspex suggests excluding the previous value of `allocBps` from the calculation, for example:

MasterBarista.sol

```
212 /// @dev Set a specified pool's alloc BPS
213 /// @param _allocBps The new alloc Bps
214 /// @param _stakeToken pid
215 function setPoolAllocBps(address _stakeToken, uint256 _allocBps) external
onlyOwner {
216     require(
217         _stakeToken != address(0) && _stakeToken != address(1),
218         "MasterBarista::setPoolAllocBps::_stakeToken must not be address(0) or
address(1)"
219     );
220     require(pools.has(_stakeToken), "MasterBarista::setPoolAllocBps::pool
hasn't been set");
221     require(_allocBps > 1000, "MasterBarista::setPoolallocBps::_allocBps must >
1000");
222     address curr = pools.next[LinkList.start];
```

```
    uint256 accumAllocBps = 0;
    while (curr != LinkList.end) {
        if (curr != _stakeToken) {
            accumAllocBps = accumAllocBps.add(poolInfo[curr].allocBps);
        }
        curr = pools.getNextOf(curr);
    }
    require(accumAllocBps.add(_allocBps) < 10000,
"MasterBarista::setPoolallocBps::accumAllocBps must < 10000");
    massUpdatePools();
    poolInfo[_stakeToken].allocBps = _allocBps;
    updatePoolsAlloc();
}
```

Please note that the remediations for other issues are not yet applied to the example above.

5.13. Improper Selling and Auction Starting Condition Checking

ID	IDX-013
Target	LatteMarket
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Very Low **Impact:** Low A sale and auction of an NFT can be replaced by a new entry, overwriting the sale or auction metadata of the original entry. This can result in unfair sales or auctions for `LatteMarket` contract's users. **Likelihood:** Low It is unlikely for the governance to set a sale or auction for an NFT with an existing entry.
Status	**Resolved** LatteSwap team has resolved this issue as suggested in commit `b508eba707236fb367692a3a6af642d8c5132beb` by checking the existence of the original sale entry before starting a new one.

5.13.1. Description

The `LatteMarket` contract can be used to set up NFT sales and auctions with the `readyToSellNFT()`, `readyToSellNFTTo()`, and `readyToStartAuction()` functions. All of these functions call `_readyToSellNFTTo()` function.

LatteMarket.sol

```
/// @notice this needs to be called when the seller want to SELL the token
/// @param _nftAddress - nft address
/// @param _categoryId - category id for each nft address
/// @param _price - price of a token
/// @param _cap - total cap for this nft address with a category id
/// @param _startBlock - starting block for a sale
/// @param _endBlock - end block for a sale
function readyToSellNFT(
    address _nftAddress,
    uint256 _categoryId,
    uint256 _price,
    uint256 _cap,
    uint256 _startBlock,
    uint256 _endBlock,
    IERC20Upgradeable _quoteToken
) external whenNotPaused onlySupportedNFT(_nftAddress)
```

```
onlyNonBiddingNFT(_nftAddress, _categoryId) onlyGovernance {
    _readyToSellNFTTo(
        _nftAddress,
        _categoryId,
        _price,
        address(_msgSender()),
        _cap,
        _startBlock,
        _endBlock,
        _quoteToken
    );
}
```

LatteMarket.sol

```
/// @notice this needs to be called when the seller want to start AUCTION the token
/// @param _nftAddress - nft address
/// @param _categoryId - category id for each nft address
/// @param _price - starting price of a token
/// @param _to - whom this token is selling to
/// @param _cap - total cap for this nft address with a category id
/// @param _startBlock - starting block for a sale
/// @param _endBlock - end block for a sale
function readyToSellNFTTo(
    address _nftAddress,
    uint256 _categoryId,
    uint256 _price,
    address _to,
    uint256 _cap,
    uint256 _startBlock,
    uint256 _endBlock,
    IERC20Upgradeable _quoteToken
) external whenNotPaused onlySupportedNFT(_nftAddress) onlyNonBiddingNFT(_nftAddress, _categoryId) onlyGovernance {
    _readyToSellNFTTo(_nftAddress, _categoryId, _price, _to, _cap, _startBlock, _endBlock, _quoteToken);
}
```

LatteMarket.sol

```
/// @notice this needs to be called when the seller want to start AUCTION the token
/// @param _nftAddress - nft address
/// @param _categoryId - category id for each nft address
/// @param _price - starting price of a token
/// @param _cap - total cap for this nft address with a category id
/// @param _startBlock - starting block for a sale
/// @param _endBlock - end block for a sale
function readyToStartAuction(
    address _nftAddress,
    uint256 _categoryId,
    uint256 _price,
    uint256 _cap,
    uint256 _startBlock,
    uint256 _endBlock,
    IERC20Upgradeable _quoteToken
) external whenNotPaused onlySupportedNFT(_nftAddress) onlyGovernance {
    latteNFTMetadata[_nftAddress][_categoryId].isBidding = true;
    _readyToSellNFTTo(
        _nftAddress,
        _categoryId,
        _price,
        address(_msgSender()),
        _cap,
        _startBlock,
        _endBlock,
        _quoteToken
    );
}
```

The `_readyToSellNFTTo()` function sets the seller address and calls the `_setLatteNFTMetadata()` and `_setCurrentPrice()` functions.

LatteMarket.sol

```
/// @dev an internal function for readyToSellNFTTo
function _readyToSellNFTTo(
    address _nftAddress,
    uint256 _categoryId,
    uint256 _price,
    address _to,
    uint256 _cap,
    uint256 _startBlock,
    uint256 _endBlock,
    IERC20Upgradeable _quoteToken
) internal {
    tokenCategorySellers[_nftAddress][_categoryId] = _to;
    _setLatteNFTMetadata(
        LatteNFTMetadataParam({
            cap: _cap,
            startBlock: _startBlock,
            endBlock: _endBlock,
            nftAddress: _nftAddress,
            nftCategoryId: _categoryId
        })
    );
    _setCurrentPrice(_nftAddress, _categoryId, _price, _quoteToken);
}
```

The `_setLatteNFTMetadata()` function sets the metadata of the NFT sale or auction entry.

LatteMarket.sol

```
function _setLatteNFTMetadata(LatteNFTMetadataParam memory _param) internal {
    require(
        _param.startBlock > block.number && _param.endBlock >
_param.startBlock,
        "LatteMarket::_setLatteNFTMetadata::invalid start or end block"
    );
    LatteNFTMetadata storage metadata =
latteNFTMetadata[_param.nftAddress][_param.nftCategoryId];
    metadata.cap = _param.cap;
    metadata.startBlock = _param.startBlock;
    metadata.endBlock = _param.endBlock;

    emit SetLatteNFTMetadata(_param.nftAddress, _param.nftCategoryId,
_param.cap, _param.startBlock, _param.endBlock);
}
```

The `_setCurrentPrice()` function sets the price and the type of token required to buy or bid for the NFT.

LatteMarket.sol

```
function _setCurrentPrice(
    address _nftAddress,
    uint256 _categoryId,
    uint256 _price,
    IERC20Upgradeable _quoteToken
) internal {
    require(address(_quoteToken) != address(0),
"LatteMarket::_setCurrentPrice::invalid quote token");
    latteNFTMetadata[_nftAddress][_categoryId].price = _price;
    latteNFTMetadata[_nftAddress][_categoryId].quoteBep20 = _quoteToken;
    emit Ask(_msgSender(), _nftAddress, _categoryId, _price, _quoteToken);
}
```

However, there is no checking whether there is an existing sale or auction or not, allowing these functions to be called multiple times for the same NFT, overwriting the metadata for the original entry, leading to an unfair changing of the entry.

5.13.2. Remediation

Inspex suggests checking the existence of the `latteNFTMetadata` for the NFT selling and auction entry before starting a new one, for example:

LatteMarket.sol

```
/// @dev an internal function for readyToSellNFTTo
function _readyToSellNFTTo(
    address _nftAddress,
    uint256 _categoryId,
    uint256 _price,
    address _to,
    uint256 _cap,
    uint256 _startBlock,
    uint256 _endBlock,
    IERC20Upgradeable _quoteToken
) internal {
    require(latteNFTMetadata[_nftAddress][_categoryId].startBlock == 0, "LatteMarket::_readyToSellNFTTo::duplicated entry");
    tokenCategorySellers[_nftAddress][_categoryId] = _to;
    _setLatteNFTMetadata(
        LatteNFTMetadataParam({
            cap: _cap,
            startBlock: _startBlock,
            endBlock: _endBlock,
            nftAddress: _nftAddress,
            nftCategoryId: _categoryId
        })
    );
    _setCurrentPrice(_nftAddress, _categoryId, _price, _quoteToken);n);
}
```

5.14. Insufficient Logging for Privileged Functions

ID	IDX-014
Target	MasterBarista OGOwnerToken
Category	Advanced Smart Contract Vulnerability
CWE	CWE-778: Insufficient Logging
Risk	**Severity:** Very Low **Impact:** Low Privileged functions' executions cannot be monitored easily by the users. **Likelihood:** Low It is not likely that the execution of the privileged functions will be a malicious action.
Status	Resolved LatteSwap team has resolved this issue as suggested in commit `c9f459bf9252cc1a601887eed0a65e61c2f68476` and `ed2f62884d147dbeac379b48da73c5e3e69cb94a` by emitting events for privileged functions.

5.14.1. Description

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

For example, the owner can set the amount of $LATTE per block by executing `setLattePerBlock()` function in the `MasterBarista` contract, and no event is emitted.

The privileged functions without sufficient logging are as follows:

File	Contract	Function
MasterBarista.sol (L:207)	MasterBarista	setLattePerBlock()
OGOwnerToken.sol (L:31)	OGOwnerToken	setOkHolders()

5.14.2. Remediation

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

MasterBarista.sol

```
207 /// @dev Set LATTE per block.
```

```
/// @param _lattePerBlock The new emission rate for LATTE
function setLattePerBlock(uint256 _lattePerBlock) external onlyOwner {
    massUpdatePools();
    emit SetLatterPerBlock(lattePerBlock, _lattePerBlock);
    lattePerBlock = _lattePerBlock;
}
```

5.15. Improper Function Visibility

ID	IDX-015
Target	LatteNFT OGNFT OGOwnerToken
Category	Smart Contract Best Practice
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved LatteSwap team has resolved this issue as suggested in commit `5878d859010ec695c6764824c58ed92a16960ee0` by changing the visibility of the affected functions.

5.15.1. Description

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

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

OGNFT.sol

```
function initialize(
    string memory _baseURI,
    IERC20Upgradeable _latte,
    IMasterBarista _masterBarista
) public initializer {
    LatteNFT.initialize(_baseURI);

    masterBarista = _masterBarista;
    latte = _latte;
}
```

Public

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

File	Contract	Function
LatteNFT.sol (L:314)	LatteNFT	pause()
LatteNFT.sol (L:320)	LatteNFT	unpause()
OGNFT.sol (L:22)	OGNFT	initialize()
OGOwnerToken.sol (L:31)	OGOwnerToken	setOkHolders()
OGOwnerToken.sol (L:37)	OGOwnerToken	mint()
OGOwnerToken.sol (L:42)	OGOwnerToken	burn()

5.15.2. Remediation

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

OGNFT.sol

```
function initialize(
    string memory _baseURI,
    IERC20Upgradeable _latte,
    IMasterBarista _masterBarista
) external initializer {
    LatteNFT.initialize(_baseURI);

    masterBarista = _masterBarista;
    latte = _latte;
}
```

5.16. Potential Economic Attack

ID	IDX-016
Target	LatteMarket
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved LatteSwap team has resolved this issue in commit `89ff4b55b92beafd202a4290d8fc958d0ba7cf13` by creating a new `OGNFTOffering` contract with a buy amount limit and a dynamic price calculation using triple slope price model. The team has also confirmed that the contract will be owned by a timelock contract.

5.16.1. Description

The `LatteMarket` can be under a potential economic attack by the attackers with a large amount of funds. The attackers can potentially buy all NFTs available from the `LatteMarket` contract and control the market price of the NFTs since the attacker owns the whole supply of those specific kinds of NFT.

5.16.2. Remediation

Inspex suggests preparing measures for this scenario. For example, limiting the number of NFT bought for each user, or setting the price dynamically for the number of the NFT bought.

6. Appendix

6.1. About Inspex

CYBERSECURITY
PROFESSIONAL
SERVICE

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

Follow Us On:

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

6.2. References

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

inspex

CYBERSECURITY PROFESSIONAL SERVICE