Security Audit Report for LatteSwap Smart Contract

Date: Dec 22, 2021

Version: 1.1

Contact: contact@blocksecteam.com

Contents

Report Manifest

Item	Description
Client	LatteSwap
Target	LatteSwap Smart Contract

Version History

Version	Date	Description
1.0	Dec 21, 2021	First Release
1.1	Dec 22, 2021	Second Release

About BlockSec The BlockSec Team focuses on the security of the blockchain ecosystem, and collaborates with leading DeFi projects to secure their products. The team is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and released detailed analysis reports of high-impact security incidents. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The auditing process is iterative. Specifically, we will audit the commits that fix the founding issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following.

Contract Name	Stage	Commit SHA
LatteSwap	Initial	`e8abea5a782fa5ceec81493aea40a3fc00a6e5bc`
LatteSwap	Updated	`115b30d32dc67ab6ce5b70e87e89dc639a1e4543`
LatteSwap	Final	`a2ff99b2454d5308f85cd197502f0a032145b415`

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report do not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team).

We also manually analyze possible attack scenarios with independent auditors to cross-check the result.

- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

- Reentrancy
- DoS
- Access control
- Data handling and data Flow
- Exception handling
- Untrusted external call and control flow
- Initialization consistency
- Events operation
- Error-prone randomness
- Improper use of the proxy system

1.3.2 DeFi Security

- Semantic consistency
- Functionality consistency
- Access control
- Business logic
- Token operation
- Emergency mechanism
- Oracle security
- Whitelist and blacklist
- Economic impact
- Batch transfer

1.3.3 NFT Security

- Duplicated item
- Verification of the token receiver
- Off-chain metadata security

1.3.4 Additional Recommendation

- Gas optimization
- Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [1] and Common Weakness Enumeration [2]. Accordingly, the severity measured in this report are classified into four categories: **High**, **Medium**, **Low** and **Undetermined**.

[1] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[2] https://cwe.mitre.org/

Chapter 2 Findings

In total, we find nine potential issues in the smart contract. We also have eight recommendations, as follows:

- High Risk: 6
- Medium Risk: 2
- Low Risk: 1
- Recommendations: 8

ID	Severity	Description	Category	Status
1	High	Unchecked Lengths for Array Parameters	Software Security	Confirmed & fixed
2	Medium	Improper Access Control for Governance	Software Security	Confirmed & fixed
3	Low	Unverified Token Address	Software Security	Not an issue
4	High	Non-updated State Variable for Reward Calculation	DeFi Security	Confirmed & fixed
5	High	Incorrect Accounting for User Deposits	DeFi Security	Confirmed & fixed
6	High	Incorrect Accounting for User Borrows and Repays	DeFi Security	Confirmed & fixed
7	Medium	Potential Reward Losses For Strategy Updates	DeFi Security	Not an issue
8	High	Accounting Errors for Collaterals	DeFi Security	Confirmed & fixed
9	High	Incorrect Accounting For the Repay Process	DeFi Security	Confirmed & fixed
10	-	Extraneous Field in Strategy Data Structure	Additional Recommendations	Confirmed & fixed
11	-	Unclear Revert Messages	Additional Recommendations	Confirmed & fixed
12	-	Unused Internal `_balanceOf` Function	Additional Recommendations	Confirmed & fixed
13	-	Unused `receive()` Function	Additional Recommendations	Confirmed & fixed
14	-	Unchecked Existence of State Variables	Additional Recommendations	Confirmed & fixed
15	-	Unchecked Function Parameters	Additional Recommendations	Confirmed & fixed
16	-	Unchecked Callee Address	Additional Recommendations	Confirmed & fixed
17	-	Do Not Use Elastic Supply Tokens	Additional Recommendations	Not an issue

The details are provided in the following sections.

2.1 Software Security

2.1.1 Unchecked Lengths for Array Parameters

Status Confirmed and fixed.

Description In the following code, array parameters are not checked whether they share the same length.

```
394function kill(
395    address[] calldata _users,
396    uint256[] calldata _maxDebtShares,
397    address _to,
398    IFlashLiquidateStrategy _flashLiquidateStrategy
399) public nonReentrant accrue {
400    // 1. Load required config
401    uint256 _liquidationPenalty = marketConfig.liquidationPenalty(address(this));
402    uint256 _liquidationTreasuryBps = marketConfig.liquidationTreasuryBps(address(this));
403    require(_liquidationPenalty <= 19000 && _liquidationPenalty >= 10000, "bad liquidation penalty
           ");
404    require(_liquidationTreasuryBps <= 2000 && _liquidationTreasuryBps >= 500, "bad liquidation
           treasury bps");
405    require(marketConfig.treasury() != address(0), "bad treasury");
```

Listing 2.1: FlatMarket.sol

Impact Unknown.

Suggestion Check lengths for array parameters.

2.1.2 Improper Access Control for Governance

Status Confirmed and fixed.

Description In both LatteSwapYieldStrategy and PCSYieldStrategy, the access control of `exit()`, `pause()`, `unpause()` are the same as `deposit()`, `withdraw()`, i.e. `onlyGovernance()`. In other words, any account with the governance role can also call `deposit()`, `withdraw()` and `exit()`. This is a violation of the access control policy that each role should be able to fulfill only one functionality.

```
189function pause() external onlyGovernance whenNotPaused {
190    _pause();
191    emit LogPause();
192}
```

Listing 2.2: PCSYieldStrategy.sol

```
 98function deposit(bytes calldata _data) external override onlyGovernance whenNotPaused {
 99    (uint256 _amount, address _sender, , uint256 _stake) = abi.decode(_data, (uint256, address,
           uint256, uint256));
100    // turns amount with n decimal into WAD
101    uint256 _share = (_amount * to18ConversionFactor).wdiv(WadRayMath.WAD); // [wad] convert
           amount of staking token with vary decimal points
```

```
102    // Overflow check for int256(wad) cast below
103    // Also enforces a non-zero wad
104    require(int256(_share) > 0, "PCSYieldStrategy::deposit:: share overflow");
```

Listing 2.3: PCSYieldStrategy.sol

Impact Unknown.

Suggestion Create a new role for `deposit()`, `withdraw()` and `exit()` actions

2.1.3 Unverified Token Address

Status Not an issue.

Description LatteSwapYieldStrategy does NOT check whether the `_stakingToken` is correct for `_latteBooster` as in PCSYieldStrategy.

```
61 function initialize(IBooster _latteBooster, IERC20Upgradeable _stakingToken) external initializer
       {
62     OwnableUpgradeable.__Ownable_init();
63     PausableUpgradeable.__Pausable_init();
64     AccessControlUpgradeable.__AccessControl_init();
65
66     latteBooster = _latteBooster;
67     stakingToken = _stakingToken;
68     masterBarista = IMasterBarista(IBooster(latteBooster).masterBarista());
69     rewardToken = IERC20Upgradeable(masterBarista.activeLatte());
70     decimals = IToken(address(_stakingToken)).decimals();
71     to18ConversionFactor = 10**(18 - decimals);
72
73     _setupRole(DEFAULT_ADMIN_ROLE, _msgSender());
74     _setupRole(GOVERNANCE_ROLE, _msgSender());
75 }
```

Listing 2.4: LatteSwapYieldStrategy.sol

Impact Unknown.

Suggestion Check corresponding token addresses in `initialize()`.

Feedback from the Developer This booster is basically a wrapped MasterChef, thus I dońt think validation is needed.

2.2 DeFi Security

2.2.1 Non-updated State Variable for Reward Calculation

Status Confirmed and fixed.

Description The `transfer()` function (in Clerk.sol) will update the balance for `_from` and `_to` along with `harvest(_from)` and `harvest(_to)`. As a result, `_from` and `_to` will receive the latest yield before transferring the balance, which can affect the yield. However, the rewardDebts inside the yield strategy will not be updated for the new balances, which affect the reward debt calculation as the balance changes.

```
252function transfer(
253    IERC20Upgradeable _token,
254    address _from,
255    address _to,
256    uint256 _share
257    ) public override allowed(_from) {
258    require(_to != address(0), "Clerk::transfer:: to not set"); // To avoid a bad UI from burning
           funds
259
260    // Harvest reward (if any) for _from and _to
261    _harvest(_from, _token);
262    _harvest(_to, _token);
263
264    balanceOf[_token][_from] = balanceOf[_token][_from] - _share;
265    balanceOf[_token][_to] = balanceOf[_token][_to] + _share;
266
267    emit LogTransfer(_token, _from, _to, _share);
268}
```

Listing 2.5: Clerk.sol

```
348function _harvest(address _sender, IERC20Upgradeable _token) internal {
349    StrategyData memory _data = strategyData[_token];
350    IStrategy _strategy = strategy[_token];
351    if (address(_strategy) == address(0)) return;
352    int256 _balanceChange = _strategy.harvest(
353        abi.encode(_data.balance, _sender, _totals[_token].share, balanceOf[_token][_sender])
354    );
355
356    if (_balanceChange == 0) {
357        return;
358    }
359
360    uint256 _totalAmount = _totals[_token].amount;
361
362    // if there is a balance from harvest, add it to amount, thus making 1 share = 1 +-
           balanceChange amount
363    if (_balanceChange > 0) {
364        uint256 _add = uint256(_balanceChange);
365        _totalAmount = _totalAmount + _add;
366        _totals[_token].amount = _totalAmount.toUint128();
367        emit LogStrategyProfit(_token, _add);
368    } else if (_balanceChange < 0) {
369        uint256 _sub = uint256(-_balanceChange);
370        _totalAmount = _totalAmount - _sub;
371        _totals[_token].amount = _totalAmount.toUint128();
372        _data.balance = _data.balance - _sub.toUint128();
373        emit LogStrategyLoss(_token, _sub);
374    }
375
376    strategyData[_token] = _data;
377}
```

Listing 2.6: Clerk.sol

Impact The reward calculation logic is incorrect due to non-updated state variable.

Suggestion Update rewardDebt in the strategy contracts based on the new balance after transferring.

2.2.2 Incorrect Accounting for User Deposits

Status Confirmed and fixed.

Description When adding/removing collaterals, FlatMarket will transfer all the user deposits to itself, and the market will be the sole owner of all collaterals. This will affect the harvest scheme, as the reward from the strategy is calculated from the user balance in Clerk. If the owner of a collateral is the market, then the user would not receive any yield from the collateral.

```
154 function _addCollateral(address _to, uint256 _share) internal {
155     userCollateralShare[_to] = userCollateralShare[_to] + _share;
156     uint256 _oldTotalCollateralShare = totalCollateralShare;
157     totalCollateralShare = _oldTotalCollateralShare + _share;
158
159     _addTokens(collateral, _share);
160
161     emit LogAddCollateral(msg.sender, _to, _share);
162 }
```

Listing 2.7: FlatMarket.sol

```
176 function _addTokens(IERC20Upgradeable _token, uint256 _share) internal {
177     clerk.transfer(_token, msg.sender, address(this), _share);
178 }
```

Listing 2.8: FlatMarket.sol

Impact Users would not receive any rewards from their collaterals.

Suggestion Fix the accounting issues in FlatMarket contract.

2.2.3 Incorrect Accounting for User Borrows and Repays

Status Confirmed and fixed.

Description There is a double accounting error in some functions in the FlatMarket contract. For example, for `borrowAndWithdraw()`:

- `balanceOf[FLAT][_to]` increased in `_borrow()`.
- FLAT transferred to `_to` in `_vaultWithdraw`.

  After this call, `_to` can call `Clerk.withdraw()` again to spend this balance.

```
229 function borrowAndWithdraw(
230     address _to,
231     uint256 _borrowAmount,
232     uint256 _minPrice,
233     uint256 _maxPrice
234 )
```

```
235     external
236     nonReentrant
237     accrue
238     updateCollateralPriceWithSlippageCheck(_minPrice, _maxPrice)
239     checkSafe
240     returns (uint256 _debtShare, uint256 _share)
241 {
242     // 1. Borrow FLAT
243     (_debtShare, _share) = _borrow(_to, _borrowAmount);
244
245     // 2. Withdraw FLAT from Clerk to "_to"
246     _vaultWithdraw(flat, _to, _borrowAmount, 0);
247 }
```

Listing 2.9: FlatMarket.sol

The similar problem also exists in the depositAndBorrow(), depositRepayAndWithdraw() functions.

Impact It makes the FlatMarket and Clerk contracts have incorrect accounting for users.

Suggestion Fix the accounting errors in the mentioned functions.

2.2.4 Potential Reward Losses For Strategy Updates

Status Not an issue.

Description The rewardDebt of users implemented in strategy may cause unfairness when setting the new strategy from the old one. When calling the `Clerk.setStrategy()`, the unclaimed rewards of the users will be lost. In summary, it means that users must call (or others call for them) the `harvest()` before it can be safely removed and the `setStrategy()` being called.

Impact Users may suffer losses when the `Clerk.setStrategy()` is called.

Suggestion Request the users to claim rewards before setting new strategies.

Feedback from the Developer It is an intended design to exit once setting up a new strategy, we cannot programmatically force all users to harvest before setting a new strategy. However, if this case happened, we would definitely announce the strategy migration to let the users get their fair amount of rewards.

2.2.5 Accounting Errors for Collaterals

Status Confirmed and fixed.

Description In the `Updated` version (commit hash `115b30d3`) of the contract, there is a double-collateral issue in FlatMarket and Clerk. Specifically:

1. Market #1 and #2 have the same collateral.
2. User A calls `depositAndAddCollateral()` in Market #1. In `_addCollateral()`, Market #1 will call `Clerk.transfer(collateral, msg.sender, _to)` (in which msg.sender and _to are both User A).
3. User A then invokes `addCollateral()` in Market #2. In this function, the require passes because there is no `userCollateralShare` for user A, and `Clerk.balanceOf(collateral, A)` is non-zero.

After the above three steps, user A has successfully borrowed twice using only one collateral.

```
229 function _addCollateral(address _to, uint256 _share) internal {
230     require(
```

```
231       clerk.balanceOf(collateral, msg.sender) - userCollateralShare[msg.sender] >= _share,
232       "not enough balance to add collateral"
233     );
234
235     userCollateralShare[_to] = userCollateralShare[_to] + _share;
236     uint256 _oldTotalCollateralShare = totalCollateralShare;
237     totalCollateralShare = _oldTotalCollateralShare + _share;
238
239     _addTokens(collateral, _to, _share);
240
241     emit LogAddCollateral(msg.sender, _to, _share);
242 }
```

Listing 2.10: FlatMarket.sol

Impact The Clerk is vulnerable and may suffer losses.

Suggestion Allow only one FlatMarket for a collateral token.

2.2.6 Incorrect Accounting For the Repay Process

Status Confirmed and fixed.

Description In the `depositRepayAndWithdraw()` function, the caller needs to repay debts for the `_for` address. However, the debt value calculation is based on the position of the `_to` address.

```
229 function depositRepayAndWithdraw(
230     address _for,
231     address _to,
232     uint256 _maxDebtReturn,
233     uint256 _collateralAmount,
234     uint256 _minPrice,
235     uint256 _maxPrice
236 ) external nonReentrant accrue updateCollateralPriceWithSlippageCheck(_minPrice, _maxPrice)
        checkSafe {
237     // 1. Find out how much debt to repaid
238     uint256 _debtValue = MathUpgradeable.min(_maxDebtReturn, debtShareToValue(userDebtShare[_to]))
            ;
239
240     // 2. Deposit FLAT to Vault for preparing to settle the debt
241     _vaultDeposit(flat, msg.sender, _debtValue, 0);
242
243     // 3. Repay the debt
244     _repay(_for, _debtValue);
245
246     // 4. Remove collateral from FlatMarket to "_to"
247     uint256 _collateralShare = clerk.toShare(collateral, _collateralAmount, false);
248     _removeCollateral(msg.sender, _collateralShare);
249
250     // 5. Withdraw collateral to "_to"
251     _vaultWithdraw(collateral, _to, _collateralAmount, 0);
252 }
```

Listing 2.11: FlatMarket.sol

Impact Unknown.

Suggestion Fix the accounting problem in `depositRepayAndWithdraw()`.

2.3 Additional Recommendation

2.3.1 Extraneous Field in Strategy Data Structure

Status Confirmed and fixed.

Description The struct `StrategyData` has an extraneous field: `strategyStartDate`. It is initialized in the `setStrategy()` function, but it is not used in any contract logic.

Impact May cause extra gas usage.

Suggestion Remove this extraneous field.

2.3.2 Unclear Revert Messages

Status Confirmed and fixed.

Description Some revert messages are unclear. For example:

```
78 function setTreasuryFeeBps(uint256 _treasuryFeeBps) external onlyOwner {
79   require(_treasuryFeeBps <= 5000, "PCSYieldStrategy::setTreasuryFeeBps:: treasury fee bps");
80   treasuryFeeBps = _treasuryFeeBps;
81 }
82
83   function setTreasuryAccount(address _treasuryAccount) external onlyOwner {
84   require(_treasuryAccount != address(0), "PCSYieldStrategy::setTreasuryAccount:: treasury account
         ");
85   treasuryAccount = _treasuryAccount;
86 }
```

Listing 2.12: PCSYieldStrategy.sol

Impact Unclear revert messages may cause misunderstandings on reverted transactions.

Suggestion Make revert messages more clear.

2.3.3 Unused Internal _balanceOf Function

Status Confirmed and fixed.

Description The function `_balanceOf` in Clerk is an internal function but not used by any other functions. Besides, the contract Clerk is not inherited by any other contract. Therefore, this function is useless.

Impact Unknown.

Suggestion Remove this function.

2.3.4 Unused receive() Function

Status Confirmed and fixed.

Description There exist some logic issues regarding receiving native tokens and the `receive()` function:

- `FlatMarket.deposit()` is non-payable, which means if a user want to deposit BNB, he must first call WBNB.deposit() and then FlatMarket.deposit().
- Anyone calls `Clerk.receive()` will lose his funds, which will be locked in this contract forever.
- If there is a market for WBNB, then only FlatMarkets can call `Clerk.deposit()`, normal users would be blocked by the `allowed()` modifier.
- If there is no market for WBNB (in the extreme rare case), directly transferring native tokens to the contract Clerk also makes no sense since `Clerk.receive()` will not do anything. In this case, users should only call FlatMarket with `msg.value > 0`.

Impact Unknown.

Suggestion Remove the unused `receive()` function.

2.3.5 Unchecked Existence of State Variables

Status Confirmed and fixed.

Description In `Clerk.whitelistMarket()`, the existing market might be replaced by accident, as there does not exist any logic to check that the collateral of the new market has no market yet.

```
121 function whitelistMarket(address _market, bool _approved) public override onlyOwner {
122     // Effects
123     whitelistedMarkets[_market] = _approved;
124     address _collateral = address(IFlatMarket(_market).collateral());
125
126     if (_approved) {
127       tokenToMarket[_collateral] = _market;
128     } else {
129       tokenToMarket[_collateral] = address(0);
130     }
131
132     emit LogTokenToMarkets(_market, _collateral, _approved);
133     emit LogWhiteListMarket(_market, _approved);
134 }
```

Listing 2.13: Clerk.sol

Impact The existing market may be replaced by accident.

Suggestion Check the existence of the state variable before setting a new entries.

2.3.6 Unchecked Function Parameters

Status Confirmed and fixed.

Description In the `initialize()` function for all contracts, address parameters are not checked against zero address. For example:

```
62 function initialize(address _wbnbToken) public initializer {
63     OwnableUpgradeable.__Ownable_init();
64
65     wbnbToken = IERC20Upgradeable(_wbnbToken);
66 }
```

Listing 2.14: Clerk.sol

Impact Invoking these `initialize()` functions with incorrect parameters may cause misbehavior.

Suggestion Check zero addresses in all `initialize()` functions.

2.3.7 Unchecked Callee Address

Status Confirmed and fixed.

Description In TokenChainlinkAggregator.latestAnswer(), it is also a good choice to check that `refBNBUSD` is a non-zero address for the Token-BNB case.

```
82    // 1. Check token-BNB price ref
83    if (refBNB != address(0)) {
84      uint256 _refBNBDecimal = IAggregatorV3Interface(refBNB).decimals();
85      uint256 _refBNBUSDDecimal = IAggregatorV3Interface(refBNBUSD).decimals();
86
87      (, int256 _answer, , uint256 _updatedAt, ) = IAggregatorV3Interface(refBNB).latestRoundData
            ();
88      require(
89        _updatedAt >= block.timestamp - maxDelayTime,
90        "TokenChainlinkAggregator::latestAnswer::delayed update time"
91      );
92
93      (, int256 _bnbAnswer, , uint256 _bnbUpdatedAt, ) = IAggregatorV3Interface(refBNBUSD).
            latestRoundData();
94      require(
95        _bnbUpdatedAt >= block.timestamp - maxDelayTime,
96        "TokenChainlinkAggregator::latestAnswer::delayed bnb-usd update time"
97      );
98
99      return
100       int256(
101         (_answer.toUint256() * _bnbAnswer.toUint256() * 10**(18 - _refBNBUSDDecimal) * 10**(18 -
              _refBNBDecimal)) /
102           1e18
103       );
104   }
```

Listing 2.15: TokenChainlinkAggregator.sol

Impact Unknown.

Suggestion Check zero address before calling the address.

2.3.8 Do Not Use Elastic Supply Tokens

Status Note an issue.

Description & Suggestion Elastic supply tokens could dynamically adjust their price, supply, user's balance, etc. Such as inflationary token, deflationary token, rebasing token, and so forth. Such a mechanism makes a DeFi system over complex. For example, a DEX using deflationary token must double check the token transfer amount when taking swap action because of the difference of actual transfer amount and

parameter. The abuse of elastic supply tokens will make the DeFi system vulnerable. In reality, many security accidents are caused by the elastic supply tokens. In terms of confidentiality, integrity and availability, we highly recommend that do not use elastic supply tokens.

Impact N/A

Feedback from the Developer Elastic supply tokens WILL NOT be whitelisted as one of the market.