SMART CONTRACT AUDIT REPORT

for

LionDEX

Prepared By: Xiaomi Huang

PeckShield

June 8, 2023

Document Properties

Client	LionDEX
Title	Smart Contract Audit Report
Target	LionDEX
Version	1.0
Author	Xuxian Jiang
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	June 8, 2023	Xuxian Jiang	Final Release
1.0-rc	June 7, 2023	Xuxian Jiang	Release Candidate

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the `LionDEX` protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About LionDEX

The `LionDEX` protocol provides perpetual futures services for multi-chain decentralized derivatives. The innovative `PvP-AMM` protocol allows for quick response to trading instructions that completely eliminate trading spreads. The protocol charges extremely low transaction fees without any lending and holding fees. The protocol maximizes capital efficiency, reduces traders' reserve ratio and significantly increases liquidity providers' annualized rate of return. At the same time, `LionDEX`'s original stop loss insurance provides traders with professional-level trading aids. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The `LionDEX` Protocol

Item	Description
Issuer	LionDEX
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	June 8, 2023

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note this audit only covers the contacts under the `contracts/core` subdirectory as well as and the `FastPriceFeed.sol` contract.

- https://github.com/LionDEXSupport/LionDex.git (81bc56c)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/LionDEXSupport/LionDex.git (8017df3)

1.2 About PeckShield

PeckShield Inc. [9] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

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

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [8]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [7], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the `LionDEX` implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	1	■
Medium	2	■ ■
Low	2	■ ■
Informational	0	
Total	5	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 high-severity vulnerabilitity, 2 medium-severity vulnerabilities, and 2 low-severity vulnerabilities.

Table 2.1: Key LionDEX Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Improved getNextAveragePrice() Logic in Vault	Business Logic	Resolved
PVE-002	Low	Revisited createIncreaseOrder() Logic in OrderBook	Coding Practices	Resolved
PVE-003	Low	Incorrect Payment Amount in executeIncreaseOrder()	Business Logic	Resolved
PVE-004	High	Incorrect Price Decimals in LionSwapFeeLP	Business Logic	Resolved
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Improved getNextAveragePrice() Logic in Vault

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Vault`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `LionDEX` protocol has a key `Vault` contract that allows the user to create or adjust his/her trading positions. While examining the current position-related logic, we notice the price adjustment of an increased position can be improved.

To elaborate, we show below the implementation of the `getNextAveragePrice()` routine. As the name indicates, this routine computes the next average price when a position is increased with `_sizeDelta` (line 964). Specifically, for current position of `_size` with its `_averagePrice`, if it is increased by `_sizeDelta` with the latest mark price `_nextPrice`, the next average price is currently computed as `(_size * _averagePrice + _sizeDelta * _nextPrice)/(_size + _sizeDelta)`, which needs to be revised as `(_size + _sizeDelta)/(_size / _averagePrice + _sizeDelta / _nextPrice)`.

```
959     function getNextAveragePrice(
960         address _indexToken,
961         uint256 _size,
962         uint256 _averagePrice,
963         uint256 _nextPrice, //index token price current
964         uint256 _sizeDelta
965     ) public view returns (uint256) {
966         require(
967             whitelistedTokens[_indexToken],
968             "Vault: getNextAveragePrice index token not white listed"
969         );
970         if (_size == 0) {
971             return _nextPrice;
```

```
        }
        return
            _size.mul(_averagePrice).add(_sizeDelta.mul(_nextPrice)).div(
                _size.add(_sizeDelta)
            );
    }
```

Listing 3.1: `Vault::getNextAveragePrice()`

Recommendation Revise the above routine to properly compute the next average price when a position is increased.

Status The issue has been fixed by this commit: `8017df3`.

3.2 Revisited createIncreaseOrder() Logic in OrderBook

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `OrderBook`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

In the `LionDEX` protocol, there is an `OrderBook` contract to manger the order books. While examining the creation of an increase order, we notice the current logic can be simplified.

To elaborate, we show below the code snippet of the `createIncreaseOrder()` routine, which requires the given `_purchaseToken` is equal to the specified `LP` (line 382). With that, the current `if` statement (line 397) on `_purchaseToken` can be simplified to remove the `then` branch and retain the `else` branch only. In addition, the internal `_createIncreaseOrder()` routine can be enhanced with the consistent use of `p.account` as the order owner, not `msg.sender`.

```
362    function createIncreaseOrder(
363        address _purchaseToken,
364        uint256 _purchaseTokenAmount,
365        address _indexToken,
366        uint256 _minOut,
367        uint256 _sizeDelta,
368        bool _isLong,
369        uint256 _insuranceLevel,
370        uint256 _triggerPrice,
371        uint256 _executionFee
372    ) external payable nonReentrant {
373        bool _triggerAboveThreshold = !_isLong;
374        validateTrigger(_triggerPrice, _indexToken, _isLong);
```

```
        _transferInETH();
        require(
            (msg.value == _executionFee) && (_executionFee >= minExecutionFee),
            "OrderBook: insufficient execution fee"
        );

        require(_purchaseToken == LP, "OrderBook: purchase token invalid");
        CreateIncreaseOrderParams memory p = CreateIncreaseOrderParams(
            msg.sender,
            _purchaseToken,
            _purchaseTokenAmount,
            _indexToken,
            _minOut,
            _sizeDelta,
            _isLong,
            _triggerPrice,
            _triggerAboveThreshold,
            _executionFee,
            _insuranceLevel,
            0
        );
        if (_purchaseToken == address(0)) {
            require(
                msg.value == _purchaseTokenAmount.add(_executionFee),
                "OrderBook: eth amount wrong"
            );
        } else {
            require(
                msg.value == _executionFee,
                "OrderBook: _executionFee wrong"
            );
        ...
    }
```

Listing 3.2: `OrderBook::createIncreaseOrder()`

Recommendation Simplify the above routine to remove unnecessary statements for improved consistency.

Status The issue has been fixed by this commit: `94d5dfd`.

3.3 Incorrect Payment Amount in executeIncreaseOrder()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `OrderBook`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

As mentioned earlier, the `OrderBook` contract in `LionDEX` is used to create various orders. In the process of analyzing the execution of an increase order, we notice the payment calculation needs to be revisited.

In the following, we show the related code snippet from the `executeIncreaseOrder()` routine. This routine requires the calculation of the order payment in `amountIn` with three components: the order's `purchaseTokenAmount`, the associated insurance cost, as well as the position fee. However, the last component is miscalculated as `order.feeLPAmount` inside the `else` branch (line 610). The miscalculation needs to be fixed as `IVault(vault).getPositionFee(order.sizeDelta)`.

```
565     function executeIncreaseOrder(
566         address _address,
567         uint256 _orderIndex,
568         address payable _feeReceiver
569     ) external override nonReentrant {
570         IncreaseOrder memory order = increaseOrders[_address][_orderIndex];
571         require(order.account != address(0), "OrderBook: non-existent order");
572 
573         // increase long should use max price
574         // increase short should use min price
575         (uint256 currentPrice, ) = validatePositionOrderPrice(
576             order.triggerAboveThreshold,
577             order.triggerPrice,
578             order.indexToken,
579             order.isLong,
580             true
581         );
582 
583         delete increaseOrders[_address][_orderIndex];
584         uint256 amountOut = order.purchaseTokenAmount;
585         {
586             {
587                 uint256 amountIn = order.purchaseTokenAmount;
588                 if (order.sizeDelta > 0) {
589                     amountIn = amountIn.add(
590                         amountIn
591                             .mul(
592                                 IVault(vault).insuranceLevel(
```

```
                                    order.insuranceLevel
                                )
                            )
                            .div(IVault(vault).BASIS_POINTS_DIVISOR())
                    );

                    if (
                        order.feeLPAmount >=
                        IVault(vault).getPositionFee(order.sizeDelta)
                    ) {
                        IFeeLP(FeeLP).burnLocked(
                            order.account,
                            address(this),
                            order.feeLPAmount,
                            true
                        );
                    } else {
                        amountIn = amountIn.add(order.feeLPAmount);
                    }
                }
                IERC20(order.purchaseToken).safeTransfer(vault, amountIn);
            }

            IVault(vault).increasePosition(
                order.account,
                order.indexToken,
                order.sizeDelta,
                amountOut,
                order.isLong,
                order.insuranceLevel,
                order.feeLPAmount
            );
            ...
        }
    }
```

Listing 3.3: `OrderBook::executeIncreaseOrder()`

Recommendation Revisit the above routine to compute the right payment token amount.

Status The issue has been fixed by this commit: `94d5dfd`.

3.4 Incorrect Price Decimals in LionSwapFeeLP

- ID: PVE-004
- Severity: High
- Likelihood: High
- Impact: High
- Target: `LionSwapFeeLP`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `LionDEX` protocol also comes with a `LionSwapFeeLP` contract that allows users to purchase vault `LP` tokens with protocol tokens, i.e., `LionToken` or `esLionToken`. The purchase naturally involves the price calculation of related tokens. While examining the current purchase price, we notice an extra decimal adjustment, which should be removed.

To elaborate, we show below the related `swap()` routine. It basically swaps the given protocol tokens (`LionToken` or `esLionToken`) to the vault `LPs`. The required protocol token amount (line 98) is computed as `needLion = LPAmount.mul(LPPrice).div(LionPrice).div(1e24)`, which needs to be revised as `needLion = LPAmount.mul(LPPrice).div(LionPrice)`. In other words, the decimals adjustment of `div(1e24)` at the end is not necessary. The same issue is also applicable to the `getLionAmount()` routine.

```
90      function swap(IERC20 buyToken, uint256 LPAmount, uint256 maxLion) public {
91          require(discountLevel[LPAmount] > 0, "LionSwapFeeLP: invalid level");
92          require(
93              buyToken == LionToken  buyToken == esLionToken,
94              "LionSwapFeeLP: buy token invalid"
95          );
96          uint256 LionPrice = getLionPrice();
97          uint256 LPPrice = vault.getMaxPrice(address(LPToken));
98          uint256 needLion = LPAmount.mul(LPPrice).div(LionPrice).div(1e24);
99          require(needLion <= maxLion, "LionSwapFeeLP: slippage");
100         require(
101             buyToken.balanceOf(msg.sender) >= needLion,
102             "LionSwapFeeLP: Lion balance invalid"
103         );
104         require(
105             buyToken.allowance(msg.sender, address(this)) >= needLion,
106             "LionSwapFeeLP: Lion allowance invalid"
107         );
108         buyToken.safeTransferFrom(msg.sender, address(this), needLion);
109         uint256 feeLPAmount = getDiscount(LPAmount);
110         feeLP.mintTo(msg.sender, feeLPAmount);
111
112         splitLionOrEsLion(buyToken, needLion);
113
114         emit Swap(msg.sender, buyToken, LPAmount, needLion, feeLPAmount);
```

```
    }
```

Listing 3.4: `LionSwapFeeLP::swap()`

Recommendation Revisit the above logic to ensure the right swap price is used for vault `LP` purchase.

Status The issue has been fixed by this commit: `a376cc9`.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `LionDEX` protocol, there are certain privilege accounts in the `owner` list that play critical role in governing and regulating the system-wide operations (e.g., configure protocol parameters, update the contract, or adjust various pools or roles). In the following, we use the `LionDEXVault` contract as an example and show the representative functions potentially affected by the privileges of the owners.

```
    function setLP(ILPToken _LP) external onlyOwner {
        LP = _LP;
    }
    function setVault(IVault _vault) external onlyOwner {
        vault = _vault;
    }

    function setSlippage(uint256 _slippage) external onlyOwner {
        require(_slippage <= basePoints, "LionDEXVault: not in range");
        slippage = _slippage;
    }

    function setKeeper(address addr, bool active) public onlyOwner {
        keeperMap[addr] = active;
        emit SetKeeper(msg.sender,addr,active);
    }
    ...
    function setSplitFeeParams(
        address _teamAddress,
        address _earnAddress,
        address _startPool,
        address _otherPool
    ) external onlyOwner {
```

```
        teamAddress = _teamAddress;
        earnAddress = _earnAddress;
        startPool = _startPool;
        otherPool  =_otherPool;
    }
    function setGMXNotEntryFlag(bool  _GMXNotEntryFlag) external onlyOwner {
        GMXNotEntryFlag = _GMXNotEntryFlag;
    }
```

Listing 3.5: Example Privileged Operations in the `LionDEXVault` Contract

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the privileged accounts may also be a counter-party risk to the protocol users. It is worrisome if the privileged accounts are plain EOA accounts. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO.

Recommendation Promptly transfer the privileged accounts to the intended `DAO`-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been confirmed and the team plans to use a multi-sig to manage the admin account.

4 Conclusion

In this audit, we have analyzed the LionDEX protocol design and implementation. The LionDEX protocol provides perpetual futures services for multi-chain decentralized derivatives. The innovative PvP-AMM protocol allows for quick response to trading instructions that completely eliminate trading spreads. The protocol charges extremely low transaction fees without any lending and holding fees. The protocol maximizes capital efficiency, reduces traders' reserve ratio and significantly increases liquidity providers' annualized rate of return. At the same time, LionDEX's original stop loss insurance provides traders with professional-level trading aids. During the audit, we notice that the current code base is well organized and those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[3] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[4] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[5] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[6] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[7] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[8] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[9] PeckShield. PeckShield Inc. https://www.peckshield.com.