SMART CONTRACT AUDIT REPORT

for

LionDEX

Prepared By: Xiaomi Huang

PeckShield

April 23, 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	April 23, 2023	Xuxian Jiang	Final Release
1.0-rc	April 22, 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	April 23, 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 following contacts: `Vault.sol`, `VaultUtil.sol`, `Router.sol`, `OrderBook.sol`, `InsuranceVault.sol`, and `LPGMXVault`.

- https://github.com/LionDEXSupport/LionDex.git (5ef8ec4)

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 (f3cf63b)

1.2 About PeckShield

PeckShield Inc. [11] 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 [10]:

- 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) [9], 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	4	■■■■
Low	2	■■
Informational	0	
Total	7	

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, 4 medium-severity vulnerabilities, and 2 low-severity vulnerabilities.

Table 2.1: Key LionDEX Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Improper LP Accounting in LP-Vault::leave()/leaveETH()	Business Logic	Resolved
PVE-002	Medium	Incorrect Execution Logic in FastPrice-Feed	Business Logic	Resolved
PVE-003	Low	Simplified Logic in getUserAllocation()	Coding Practices	Resolved
PVE-004	High	Possible Reward Drain from Vulnerable Pool::deposit()	Business Logic	Resolved
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-006	Low	Incorrect Pending Reward Calculation in LionSwapFeeLP	Business Logic	Resolved
PVE-007	Medium	Possible Sandwich/MEV Attacks For Reduced Returns	Time And State	Resolved

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 Improper LP Accounting in LPVault::leave()/leaveETH()

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: LPVault
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The LionDEX protocol has a key LPVault contract that allows the user to buy or sell LPs. While examining the sell logic, we notice the current implementation has a flawed logic in balance accounting.

To elaborate, we show below the code snippets of the leaveETH() routine. As the name indicates, this routine allows the current LP providers to redeem and claim their asset. By design, the protocol will collect necessary leave fee and thus the LP providers will expect to receive less. However, it comes to our attention that the LP share to burn is reduced from the collected fee, which seems contradictory to the protocol design. Note that another leave() routine shares the same issue.

```
    function leaveETH(uint256 _share) external payable nonReentrant {
        require(
            _share <= LP.balanceOf(msg.sender) && _share > 0,
            "balance too low"
        );

        updateTotalGLP();
        uint256 originalShare = _share;
        //collect leave fee
        uint256 leaveFee = _share.mul(burnLPFeeBasisPoints).div(basePoints);
        feeReserves[address(LP)] = feeReserves[address(LP)].add(leaveFee);
        _share = _share.sub(leaveFee);

        uint256 totalShares = LP.totalSupply();
        uint256 amountOutGLP = _share.mul(totalGLP).div(totalShares);
        //update global
```

```
        totalGLP = totalGLP.sub(amountOutGLP);
        LP.burn(msg.sender, _share);
         //leaveFee will transfer to treasury vault;
        IERC20(LP).safeTransfer(treasuryVault,leaveFee);
        uint256 amountSendOut = unstakeGLP(amountOutGLP, address(WETH));

        WETH.withdraw(amountSendOut);

        (bool success, ) = payable(msg.sender).call{value: amountSendOut}("");
        require(success, "Failed to send Ether");
        updateRewardPerSecond();
        compoundRewardsETHandEsGMXInternal();

        emit SellLP(
            msg.sender,
            msg.sender,
            address(WETH),
            originalShare,
            amountSendOut,
            burnLPFeeBasisPoints
        );
    }
```

Listing 3.1: `LPVault::leaveETH()`

Recommendation Revise the above mentioned routines to properly compute the right share to burn.

Status The issue has been fixed by this commit: `e335898`.

3.2 Incorrect Execution Logic in FastPriceFeed

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `FastPriceFeed`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

In the `LionDEX` protocol, there is a `FastPriceFeed` contract to provide a fast price feed and ensure prices are up to date. In the meantime,e it greatly helps to execute user orders. While examining the current order lifecycle, we notice one specific routine incorrectly implements the intended logic.

To elaborate, we show below the code snippet of the `setPricesWithBitsAndExecute()` routine, which not only provides price updates, but also executes available orders, including both `IncreasePositions` and `DecreasePositions`. However, the `maxEndIndexForDecrease` variable is incorrectly computed as `_positionRouter.increasePositionRequestKeysStart().add(_maxDecreasePositions)`, which needs to be corrected as `_positionRouter.decreasePositionRequestKeysStart().add(_maxDecreasePositions)`!

```
    function setPricesWithBitsAndExecute(
        uint256 _priceBits,
        uint256 _timestamp,
        uint256 _endIndexForIncreasePositions,
        uint256 _endIndexForDecreasePositions,
        uint256 _maxIncreasePositions,
        uint256 _maxDecreasePositions
    ) external onlyUpdater {
        _setPricesWithBits(_priceBits, _timestamp);

        IRouter _positionRouter = IRouter(positionRouter);
        uint256 maxEndIndexForIncrease = _positionRouter.
            increasePositionRequestKeysStart().add(_maxIncreasePositions);
        uint256 maxEndIndexForDecrease = _positionRouter.
            increasePositionRequestKeysStart().add(_maxDecreasePositions);

        if (_endIndexForIncreasePositions > maxEndIndexForIncrease) {
            _endIndexForIncreasePositions = maxEndIndexForIncrease;
        }

        if (_endIndexForDecreasePositions > maxEndIndexForDecrease) {
            _endIndexForDecreasePositions = maxEndIndexForDecrease;
        }

        _positionRouter.executeIncreasePositions(_endIndexForIncreasePositions, payable(
            msg.sender));
        _positionRouter.executeDecreasePositions(_endIndexForDecreasePositions, payable(
            msg.sender));
    }
```

Listing 3.2: `FastPriceFeed::setPricesWithBitsAndExecute()`

Recommendation Correct the above routine to compute the right positions for user request execution.

Status The issue has been fixed by this commit: `e335898`.

3.3 Simplified Logic in LionIDO::getUserAllocation()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `LionIDO`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

There is an associated `LionIDO` contract to raise funds from investors. And the IDO logic is designed with two offering tokens: `LION` and `esLION`. While examining the IDO logic, we notice a helper routine can be improved.

In the following, we show the related code snippet from the `getUserAllocation()` helper routine. This routine computes the user allocation based on the invested amount. Note the current amount (line 686) is computed as `userInfo[_user].amount.mul(1e12).div(totalAmount).div(1e6)`, which can be improved as `userInfo[_user].amount.mul(1e6).div(totalAmount)` without any loss of precision.

```
684     // allocation 100000 means 0.1(10%)
685     function getUserAllocation(address _user) public view returns(uint256) {
686         return userInfo[_user].amount.mul(1e12).div(totalAmount).div(1e6);
687     }
```

Listing 3.3: `LionIDO::getUserAllocation()`

Recommendation Revisit the above routine to avoid unnecessary computation.

Status The issue has been fixed by this commit: `93b8a48`.

3.4 Possible Reward Drain from Vulnerable Pool::deposit()

- ID: PVE-004
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `Pool`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

To incentivize the protocol adoption, the `LionDEX` protocol has designed a `Pool` contract that rewards users to deposit intended tokens to specific pools. The rewards may come from the protocol airdrops or collected free from the protocol operation. While examining the current deposit logic for rewards, we notice the implementation has a flaw that may be exploited to drain available rewards.

To elaborate, we show below the related `deposit()` routine. It basically accumulates the latest pool rewards, computes the reward amount for the calling user, transfers in user deposit, and then updates the balances. Unfortunately, this routine does not validate the give transfer-in token so that the user may craft a malicious one to regain the execution before the user balance is updated. Worse, the current logic does not enforce much-needed reentrancy protection, which may be exploited to re-enter the deposit and drain the reward available in the current pool.

```
    function deposit(
        uint256 _pid,
        IERC20 depositToken,
        uint256 _amount
    ) public {
        PoolInfo storage pool = poolInfo[_pid];
        require(pool.startTime > 0, "BasePools: pool not exist");
        UserInfo storage user = userInfo[_pid][msg.sender];
        updatePool(_pid);
        if (user.amount.length == 0) {
            user.amount = new uint256[](pool.depositTokens.length);
        }
        if (user.rewardDebt.length == 0) {
            user.rewardDebt = new uint256[](pool.rewardTokens.length);
        }

        uint256 tokenLength = pool.rewardTokens.length;
        for (uint i; i < tokenLength; i++) {
            if (user.weight > 0) {
                uint256 pending = user
                    .weight
                    .mul(pool.accRewardTokenPerShare[i])
                    .div(precise)
                    .sub(user.rewardDebt[i]);
                if (pending > 0) {
                    pool.rewardTokens[i].safeTransfer(msg.sender, pending);
                }
            }
        }

        if (_amount > 0) {
            depositToken.safeTransferFrom(
                address(msg.sender),
                address(this),
                _amount
            );
            for (uint i; i < pool.depositTokens.length; i++) {
                if (pool.depositTokens[i] == depositToken) {
                    user.amount[i] = user.amount[i].add(_amount);
                }
            }
```

```
            user.totalAmount = user.totalAmount.add(_amount);
            uint256 buff = user.totalAmount.mul(user.buff).div(BasePoint);
            user.weight = user.weight.add(buff);
            pool.totalWeight = pool.totalWeight.add(buff);
        }
        for (uint i; i < pool.rewardTokens.length; i++) {
            user.rewardDebt[i] = user
                .weight
                .mul(pool.accRewardTokenPerShare[i])
                .div(precise);
        }
        emit Deposit(msg.sender, _pid, depositToken, _amount);
    }
```

Listing 3.4: `Pool::deposit()`

Recommendation Revisit the above logic to ensure the reentrancy protection is enforced and the provided user input (e.g., the deposit token) needs to validated.

Status The issue has been fixed by this commit: `93b8a48`.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [5]
- 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;
    }

    function isKeeper(address addr) public view returns (bool) {
        return keeperMap[addr];
    }
    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.

3.6 Incorrect Pending Reward Calculation in LionSwapFeeLP

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `BasePools`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

As mentioned earlier, the `LionDEX` protocol incentivizes the protocol users with rewards. While examining the current reward logic, we notice the user pending reward is computed incorrectly.

To elaborate, we show below the related `pendingReward()` routine. As the name indicates, this routine computes the pending reward for the given user. While it properly makes use of user weight to take a share of the pool's entire rewards, it does not properly consider the pool's `accRewardTokenPerShare`!

```
    function pendingReward(
        uint256 _pid,
        address _user
    ) external view returns (uint256[] memory rewards) {
        PoolInfo memory pool = poolInfo[_pid];
        UserInfo memory user = userInfo[_pid][_user];

        uint256 tokenLength = pool.rewardTokens.length;
        rewards = new uint256[](tokenLength);

        for (uint i; i < tokenLength; i++) {
            uint256 multipier = getMultiplier(
                pool.lastRewardTime,
                block.timestamp
            );
            uint256 reward = multipier.mul(pool.rewardTokenPerSecond[i]);
            uint256 accRewardPerShare = reward
                .mul(user.weight)
                .mul(precise)
                .div(pool.totalWeight);

            rewards[i] = user.amount[i].mul(accRewardPerShare).div(precise).sub(
                user.rewardDebt[i]
            );
        }
    }
```

Listing 3.6: `BasePools::pendingReward()`

Recommendation Revisit the logic to properly compute the pending reward for the given user.

Status The issue has been fixed by this commit: `1025b5f`.

3.7 Potential Front-Running/MEV With Reduced Return

- ID: PVE-007
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `LionSwapFeeLP`
- Category: Time and State [8]
- CWE subcategory: CWE-682 [3]

Description

As mentioned earlier, the `LionDEX` protocol has a constant need of swapping one asset to another. With that, the protocol has provided related helper routines to facilitate the asset conversion. Accordingly, the protocol has implemented the functionality to query the current `LION` price. However, we notice the current price is obtained from the instantaneous feed from a public `UniswapRouterV3`-like DEX engine, which may subject to possible manipulation.

```
    function swap(IERC20 buyToken, uint256 LPAmount, uint256 maxLion) public {
        require(discountLevel[LPAmount] > 0, "LionSwapFeeLP: invalid level");
        require(
            buyToken == LionToken  buyToken == esLionToken,
            "LionSwapFeeLP: buy token invalid"
        );
        uint256 LionPrice = getLionPrice();
        uint256 LPPrice = vault.getMaxPrice(address(LPToken));
        uint256 needLion = LPAmount.mul(LPPrice).div(LionPrice);
        require(needLion <= maxLion, "LionSwapFeeLP: slippage");
        require(
            buyToken.balanceOf(msg.sender) >= needLion,
            "LionSwapFeeLP: Lion balance invalid"
        );
        require(
            buyToken.allowance(msg.sender, address(this)) >= needLion,
            "LionSwapFeeLP: Lion allowance invalid"
        );
        buyToken.safeTransferFrom(msg.sender, address(this), needLion);
        uint256 feeLPAmount = getDiscount(LPAmount);
        feeLP.mintTo(msg.sender, feeLPAmount);

        splitLionOrEsLion(buyToken, needLion);

        emit Swap(msg.sender, buyToken, LPAmount, needLion, feeLPAmount);
    }
```

Listing 3.7: `LionSwapFeeLP::swap()`

```
    function getLionPrice() public view returns (uint256) {
        bytes memory path = abi.encodePacked(
            address(LionToken),
            pairFee,
            address(usdc)
        );
        return router.quoteExactInput(path, 1e18);
    }
```

Listing 3.8: `LionSwapFeeLP::getLionPrice()`

To elaborate, we show above the related helper routine. We notice the conversion is routed to `UniswapV3` in order to determine the current `LION` price. Apparently, the instant DEX price is highly volatile and there is a need to consider the use of `TWAP` and further specify necessary restriction on the swap operation on possible slippage, so that it is not vulnerable to possible front-running attacks.

Note that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user because the swap rate is lowered by the preceding sell. As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade or referencing the `TWAP` or `time-weighted average price` of `UniswapV2`. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

Recommendation Develop an effective mitigation (e.g., slippage control) to the above front-running attack to better protect the interests of farming users.

Status This issue has been fixed in the following commit: `f3cf63b`.

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-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

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

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

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

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

[8] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

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

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

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