SMART CONTRACT AUDIT REPORT

for

LogX

Prepared By: Xiaomi Huang

PeckShield

November 21, 2023

Document Properties

Client	LogX
Title	Smart Contract Audit Report
Target	LogX
Version	1.0
Author	Xuxian Jiang
Auditors	Colin Zhong, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	November 21, 2023	Xuxian Jiang	Final Release
1.0-rc	November 15, 2023	Xuxian Jiang	Release Candidate #1

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 source code of the LogX 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 related to either security or performance. This document outlines our audit results.

1.1 About LogX

LogX is a decentralised exchange for trading perpetuals. It is designed to provide lightning fast execution at very low fees and zero price impact. The trading is supported by the LogX pool which contains stable coins. Liquidity providers earn based on the performance of the pool and fees collected from trading. The price feeds are supported by dark oracle, which fetches prices from Pyth oracles and other centralized exchanges to provide better aggregated prices. The aggregation is done to provide additional safety for liquidity providers. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The LogX

Item	Description
Name	LogX
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	November 21, 2023

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/eugenix-io/logX-LP.git (831b10e)

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

- https://github.com/eugenix-io/logX-LP.git (2b4c1c5)

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.

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

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

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 implementation of the LogX protocol. 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	3	■■■
Medium	4	■■■■
Low	6	■■■■■■
Informational	0	
Total	13	

We have so far identified a list of potential issues. 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 3 high-severity vulnerabilities, 4 medium-severity vulnerabilities, and 6 low-severity vulnerabilities.

Table 2.1: Key LogX Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Public Exposure of Privileged Functions	Security Feature	**Resolved**
PVE-002	High	Incorrect TP/SL Order Creation/Update in OrderManager	Business Logic	**Resolved**
PVE-003	Medium	Improper MaxTP Increase Position Creation in OrderManager	Business Logic	**Resolved**
PVE-004	High	Improper Increase Position Cancellation in OrderManager	Business Logic	**Resolved**
PVE-005	Medium	Improper Increase Position Execution Logic in OrderManager	Business Logic	**Resolved**
PVE-006	Medium	Improper Pool Amount Accounting in Vault	Business Logic	**Resolved**
PVE-007	Low	Improper Position Increase Validation in Utils	Business Logic	**Resolved**
PVE-008	Low	Revisited Position Decrease Logic in Vault	Business Logic	**Resolved**
PVE-009	Low	Revisited Position Liquidation Validation Logic in Utils	Business Logic	**Resolved**
PVE-010	Low	Inconsistent Vault Token Config Update Logic in TimeLock	Coding Practices	**Resolved**
PVE-011	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	**Resolved**
PVE-012	Low	LLP CooldownDuration Bypass in Liquidity Removal	Business Logic	**Resolved**
PVE-013	Medium	Trust Issue of Admin Keys	Security Features	**Mitigated**

All findings have been resolved in latest commit of 2b4c1c5 by `LogX`. 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 Public Exposure of Privileged Functions

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: High
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

The audited `LogX` protocol is a unique decentralized derivative exchange. To facilitate the trading and position management, the protocol has a number of privileged functions. While examining these privileged functions, we notice some of them are publicly exposed without caller verification.

In the following, we show an example privileged routine from the `PriceFeed` contract. This routine is designed to configure the latest asset price. However, this routine is public and its public exposure without any caller authentication will corrupt the protocol integrity or cripple the entire protocol functionality.

```
    function _setPrice(address _tokenAddress,  PriceArgs memory _darkOraclePrice) public
         {
        validateData(_darkOraclePrice.publishTime);
        TokenPrice memory priceObject = TokenPrice(_darkOraclePrice.price,
            _darkOraclePrice.price, _darkOraclePrice.expo, _darkOraclePrice.expo,
            _darkOraclePrice.publishTime);
        tokenToPrice[_tokenAddress] = priceObject;
        emit PriceSet(priceObject);
    }
    ...
    function compareAndSetPrice(address _tokenAddress,
        PythStructs.Price memory _pythPrice, PriceArgs memory _darkOraclePrice) public {
        uint256 pythPrice = getFinalPrice(uint64(_pythPrice.price), _pythPrice.expo);
        uint256 darkOraclePrice = getFinalPrice(uint64(_darkOraclePrice.price),
            _darkOraclePrice.expo);

```

```
        if (allowedDelta(pythPrice, darkOraclePrice)) {
            _setPrice(_tokenAddress, _darkOraclePrice);
        } else {
            validateData(_pythPrice.publishTime);
            TokenPrice memory priceObject = TokenPrice(
                pythPrice > darkOraclePrice
                    ? uint64(_pythPrice.price)
                    : _darkOraclePrice.price,
                pythPrice < darkOraclePrice
                    ? uint64(_pythPrice.price)
                    : _darkOraclePrice.price,
                pythPrice > darkOraclePrice
                    ? _pythPrice.expo
                    : _darkOraclePrice.expo,
                pythPrice < darkOraclePrice
                    ? _pythPrice.expo
                    : _darkOraclePrice.expo,
                _darkOraclePrice.publishTime
            );
            tokenToPrice[_tokenAddress] = priceObject;
            emit PriceSet(priceObject);
        }
    }
```

Listing 3.1: `PriceFeed::updatePrice()/compareAndSetPrice()`

Recommendation Revisit all public functions and add necessary caller verification. Note this issue affects a few public functions, including `RewardTracker::setRewardPrecision()`.

Status This issue has been fixed by the following commit: `e0cc24c`.

3.2 Incorrect TP/SL Order Creation/Update in OrderManager

- ID: PVE-002
- Severity: High
- Likelihood: High
- Impact: High
- Target: `OrderManager`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

To facilitate the order management, the `LogX` protocol has a built-in `OrderManager` contract. In the process of analyzing the order creation logic, we notice the current implementation has an improper way to create and update orders.

In the following, we show the code snippet of the related `createOrders()` routine. This routine has a number of arguments and is defined to allow the user to create new orders. However, it comes to our attention that the new limit order has been specified with a user-controlled argument `_isIncreaseOrder` (line 971), which should be a constant `true`. Similarly, the associated `TP/SL` orders should have its `_isIncreaseOrder` with a constant `false`, not modified by the user either. In addition, the same `TP/SL` orders should also have `0` as its `_collateralDelta` argument.

```
964            if(limitPrice != 0){
965                uint256 currMarketPrice = _isLong? IPriceFeed(pricefeed).
                       getMaxPriceOfToken(_indexToken):IPriceFeed(pricefeed).
                       getMinPriceOfToken(_indexToken);
966                    _validateLimitOrderPrices(currMarketPrice, _isLong, _limitPrice);
967
968                    IERC20(_collateralToken).transferFrom(msg.sender, address(this),
                           _collateralDelta);
969                    uint256 _collateralAmountUsd = IUtils(utils).tokenToUsdMin(
                           _collateralToken, _collateralDelta);
970                    require(_collateralAmountUsd >= minPurchaseTokenAmountUsd, "
                           OrderManager: too less collateral");
971                    _createOrder(msg.sender, _collateralDelta, _collateralToken,
                           _indexToken, _sizeDelta, _isLong, _limitPrice, !_isLong,
                           minExecutionFeeLimitOrder, _isIncreaseOrder, _maxOrder);
972            }else{
973                // tpsl order or limit order when closing position
974                uint256 currMarketPrice = !_isLong? IPriceFeed(pricefeed).
                       getMaxPriceOfToken(_indexToken):IPriceFeed(pricefeed).
                       getMinPriceOfToken(_indexToken);
975                _validateTPSLOrderPrices(currMarketPrice, _isLong, _tpPrice, _slPrice);
976                if(tpPrice != 0){
977                    _createOrder(msg.sender, _collateralDelta, _collateralToken,
                           _indexToken, _sizeDelta, _isLong, _tpPrice, _isLong,
                           minExecutionFeeLimitOrder, _isIncreaseOrder, _maxOrder);
978                }
979                if(slPrice !=0){
980                    _createOrder(msg.sender, _collateralDelta, _collateralToken,
                           _indexToken, _sizeDelta, _isLong, _slPrice, !_isLong,
                           minExecutionFeeLimitOrder, _isIncreaseOrder, _maxOrder);
981                }
982            }
```

Listing 3.2: `OrderManager::createOrders()`

Moreover, the create order may be updated via a routine `updateOrder()`, which should be enhanced with the proper validation on the given `_triggerPrice` and `_triggerAboveThreshold`. We also notice the order update routine should not update the order's `_collateralDelta` without properly transferring in or out respective collateral.

Recommendation Revise the above routine to properly manage user orders.

Status This issue has been fixed by the following commits: `1863de3`. and `3ee32fc`.

3.3 Improper MaxTP Increase Position Creation in OrderManager

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `OrderManager/Utils`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

As mentioned earlier, the `LogX` protocol has a built-in `OrderManager` contract to manager user orders. While reviewing the order creation process, we notice the current implementation has attached a `maxTP` order to limit possible max profit. However, the current approach to compute associated `take-profit` price should be improved.

In the following, we show the code snippet of the related `getTPPrice()` routine. This routine is used to compute the `take-profit` price to meet the `maxProfitMultiplier` requirement. However, it comes to our attention that `profitDelta` should be computed as `(_maxTPAmount * markPrice * getMinPrice(collateralToken))/(sizeDelta * 10**vault.tokenDecimals(collateralToken))`, not current `(_maxTPAmount * markPrice * 10**(30 - vault.tokenDecimals(collateralToken)))/sizeDelta` (line 828).

```
824     function getTPPrice(uint256 sizeDelta, bool isLong, uint256 markPrice,
825             uint256 _maxTPAmount, address collateralToken)
826             view public returns(uint256)
827     {
828         uint256 profitDelta = (_maxTPAmount * markPrice * 10**(30 - vault.tokenDecimals(
                collateralToken)))/sizeDelta;
829
830         if(isLong){
831             return markPrice + profitDelta;
832         }
833
834         return markPrice - profitDelta;
835     }
```

Listing 3.3: `Utils::getTPPrice()`

Moreover, the `maxTP` order should be instantiated with the parameter `_isLong`, not using the hardcoded `true` (line 354).

Recommendation Revise the above routine to properly manage the attached `maxTP` order.

Status This issue has been fixed by the following commit: `63977b0`.

3.4 Improper Increase Position Cancellation in OrderManager

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

Description

The new increase order creation may come with the creation of associated new TP/SL/maxTP orders. In the process of analyzing their execution fee, we notice the current fee collection logic may expose a vulnerability to drain funds in OrderManager.

In the following, we show the implementation of the createIncreasePosition() routine. This routine is designed to create an increase position order. Based on the given arguments, it will also create a maxTP order as well as possibly two other TP/SL orders. We notice both TP/SL orders may be collected with the so-called minExecutionFeeLimitOrder fee while the creation of maxTP order is mandatory, but without the minExecutionFeeLimitOrder fee. However, its cancellation may always refund the order creator with the minExecutionFeeLimitOrder fee.

```
        {
            uint256 collateralAmount = _amountIn;
            bool isLong = _isLong;
            address collateralToken = _collateralToken;
            address indexToken = _indexToken;
            uint256 sizeDelta = _sizeDelta;
            tpPrice = IUtils(utils).getTPPrice(_sizeDelta, true, _acceptablePrice,
                collateralAmount * maxProfitMultiplier, collateralToken);
            _createOrder(msg.sender, 0, collateralToken, indexToken, sizeDelta, isLong,
                tpPrice, isLong, minExecutionFeeLimitOrder, false , true);
            return positionKey;
        }
```

Listing 3.4: OrderManager::createIncreasePosition()

```
    function _cancelOrder(bytes32 orderKey, uint256 _orderIndex, Order memory order)
        internal {
        require(order.account != address(0), "OrderManager: non-existent order");

        delete orders[orderKey];
        EnumerableSet.remove(orderKeys, orderKey);
        if(order.isIncreaseOrder){
            IERC20(order.collateralToken).transfer(order.account, order.collateralDelta)
                ;
        }
        (bool success, ) = (order.account).call{value: order.executionFee}("");
```

```
        require(success, "OrderManager: Exectuion Fee transfer failed");
        ...
    }
```

Listing 3.5: `OrderManager::_cancelOrder()`

Moreover, we may need to revisit the order cancellation logic to cancel the associated `TP/SL/maxTP` orders if the base increase position order is cancelled.

Recommendation Revise the above routine to properly refund user fee only if the fee is collected.

Status This issue has been fixed by the following commit: `a325e0d`.

3.5 Improper Increase Position Execution Logic in OrderManager

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `OrderManager`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The created increase order requests may be executed by authorized entities, i.e., `onlyPositionKeeper`. In the process of reviewing the execution of these increase position requests, we notice the current execution logic should be improved.

In the following, we show the implementation of the affected `executeIncreasePositions()` routine. As the name indicates, this routine is designed to batch-execute the created increase position requests. It has a rather straightforward logic in iterating each request for the attempted execution. If the execution is not successful, it aims to cancel the request. If the cancel also fails, the current logic simply deletes the request from the recorded `increasePositionRequestKeys` array. We argue that the request deletion upon the cancellation failure is not appropriate as it still does not refund the user funds!

```
    function executeIncreasePositions(
        uint256 _endIndex,
        address payable _executionFeeReceiver
    ) external override onlyPositionKeeper {
        uint256 index = increasePositionRequestKeysStart;
        uint256 length = increasePositionRequestKeys.length;

```

```
        if (index >= length) {
            return;
        }

        if (_endIndex > length) {
            _endIndex = length;
        }

        while (index < _endIndex) {
            bytes32 key = increasePositionRequestKeys[index];

            // if the request was executed then delete the key from the array
            // if the request was not executed then break from the loop, this can happen
                 if the
            // minimum number of blocks has not yet passed
            // an error could be thrown if the request is too old or if the slippage is
            // higher than what the user specified, or if there is insufficient
                liquidity for the position
            // in case an error was thrown, cancel the request
            try
                this.executeIncreasePosition(key, _executionFeeReceiver)
            returns (bool _wasExecuted) {
                if (!_wasExecuted) {
                    break;
                }
            } catch {
                // wrap this call in a try catch to prevent invalid cancels from
                    blocking the loop
                try
                    this.cancelIncreasePosition(key, _executionFeeReceiver)
                returns (bool _wasCancelled) {
                    if (!_wasCancelled) {
                        break;
                    }
                } catch {}
            }

            delete increasePositionRequestKeys[index];
            index++;
        }

        increasePositionRequestKeysStart = index;
    }
```

Listing 3.6: `OrderManager::executeIncreasePositions()`

Recommendation Revise the above routine to properly refund user funds when the request cancellation also fails.

Status This issue has been fixed by the following commit: `7b1d241`.

3.6 Improper Pool Amount Accounting in Vault

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Vault`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

`LogX` is a decentralised exchange for trading perpetuals with its core trading logic in the `Vault` contract. This `Vault` contract has the key accounting `poolAmounts` state to keep track of pool funds for `LLP` pricing. While analyzing various activities that may affect the pool amount, we notice an issue in the position liquidation functionality that does not properly update the pool amount.

In the following, we show the code snippet from the `liquidatePosition()` routine. This routine itself is designed to liquidate an underwater position. We notice the pool amount adjustment differs on the computed `marginFees`. If `marginFees<0`, the liquidated position may have positive funding rate and the pool amount should be increased by both `abs(marginFees)` and the position collateral. Similarly, if `marginFees>0`, we need to either reward pool by adding `position.collateral - uint(marginFees)`, or decrease pool amount by subtracting `uint(marginFees)- position.collateral`. The current adjustment only considers the pool-rewarding branch, not the pool-deduction branch.

```
        if(marginFees<0){
            _increasePoolAmount(position.collateralToken, utils.usdToTokenMin(position.
                collateralToken, uint(abs(marginFees))));
        } else {
            if (uint(marginFees) < position.collateral) {
            uint256 remainingCollateral = position.collateral - uint(marginFees);
            _increasePoolAmount(
                position.collateralToken,
                utils.usdToTokenMin(position.collateralToken, remainingCollateral)
            );
        }
        }
```

Listing 3.7: `Vault::createIncreasePosition()`

Recommendation Revise the above routine to properly adjust the pool amount when a position is liquidated.

Status This issue has been fixed by the following commit: `32bdf1f`.

3.7 Improper Position Increase Validation in Utils

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Utils`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `LogX` protocol is no exception. Specifically, if we examine the `Vault` contract, it has defined a number of protocol-wide risk parameters, such as `maxOIImbalance` and `maxLiquidityPerUser`. Our analysis shows that the `maxLiquidityPerUser` enforcement has an issue in the total liquidity calculation.

In the following, we show the implementation of the related `validateIncreasePosition()` routine. As the name indicates, this routine is designed to validate the increase position order. We notice the `availableLiquidityInUsd` state computes the available liquidity. However, its calculation does not take into account the token's decimals when adding each supported token `vault.poolAmounts(token) * price` (line 84). As a result, it greatly affects the final validation of `maxLiquidityPerUser`.

```
61     function validateIncreasePosition(
62         address _account,
63         address _collateralToken,
64         address _indexToken,
65         uint256  _sizeDelta,
66         bool   _isLong
67     ) external view override {
68 
69         if(!isValidate){
70             return;
71         }
72 
73         Position memory prevPosition = getPosition(_account, _collateralToken,
               _indexToken, _isLong);
74         uint256 sizeAfterUpdate = _sizeDelta + prevPosition.size;
75         uint256 length = vault.allWhitelistedTokensLength();
76         uint256 availableLiquidityInUsd = 0;
77 
78         for (uint256 i = 0; i < length; i++) {
79             address token = vault.allWhitelistedTokens(i);
80             if(!vault.canBeCollateralToken(token)){ // instead of whitelistedToken we
                   should check for canBeCollateralToken true false?
81                 continue;
82             }
83             uint256 price = getMinPrice(token);
```

```
        availableLiquidityInUsd += vault.poolAmounts(token) * price;
    }
    require(sizeAfterUpdate*100/(availableLiquidityInUsd) < vault.
        maxLiquidityPerUser(_indexToken), "Utils: Huge liquidity captured for single
         user");
}
```

Listing 3.8: `Utils::validateIncreasePosition()`

Recommendation Revise the above routine to properly compute the available liquidity and enforce the `maxLiquidityPerUser` requirement.

Status This issue has been fixed by the following commit: `0064a31`.

3.8 Revisited Position Decrease Logic in Vault

- ID: PVE-008
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Vault`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

As mentioned earlier, `Vault` manages active trade positions in `LogX`. In the process of reviewing the execution of a decrease position orders, we notice the current execution logic should be improved.

In the following, we show the code snippet from the affected `_decreasePosition()` routine. Within this routine, there is a need to properly return back to the user once its position is decreased with realized profit and loss. The return fund amount is saved in the `usdOutAfterFee` state. However, its transfer depends on the `usdOut` state (line 1114). Our analysis indicates that it is possible to have positive `usdOutAfterFee` while `usdOut` remains as `0`.

```
1106        (uint256 usdOut, uint256 usdOutAfterFee, int256 signedDelta) = _reduceCollateral
                (
1107            _account,
1108            _collateralToken,
1109            _indexToken,
1110            0,
1111            _sizeDelta,
1112            _isLong
1113        );
1114        if (usdOut > 0) {
1115            amountOutAfterFees = utils.usdToTokenMin(
1116                _collateralToken,
1117                usdOutAfterFee
```

```
            );
            _transferOut(_collateralToken, amountOutAfterFees, _receiver);
        }
```

Listing 3.9: `Vault::_decreasePosition()`

Recommendation Revise the above routine to properly return user funds when the position is decreased.

Status This issue has been fixed by the following commit: `04eac8c`.

3.9 Revisited Position Liquidation Validation Logic in Utils

- ID: PVE-009
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Utils`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

In Section 3.7, we have examined the logic to validate the increase position order. In this section, we analyze the validation logic related to the position liquidation and report an issue that may wrongfully liquidate a profitable position that could have small collateral, but with high margin fee.

In the following, we show the code snippet from the related `validateLiquidation()` routine. This routine is designed to validate whether the position can be liquidated or not. We notice it mainly compares the position collateral with margin fee that will be collected, including `borrowing fee`, `position fee`, `funding fee` as well as possible `liquidation fee`. It comes to our attention that it does not credit the position with the profit when comparing with margin fee. As a result, a profitable position that may have small collateral, but with high margin fee is considered as liquidatable.

```
        int256 marginFees = int(getBorrowingFee(
            _account,
            _collateralToken,
            _indexToken,
            _isLong,
            position.size,
            position.entryBorrowingRate
        ));
        marginFees =
            marginFees +
            int(
                getPositionFee(
                    _account,
```

```
                    _collateralToken,
                    _indexToken,
                    _isLong,
                    position.size
                )
            );

        marginFees = marginFees + getFundingFee(_account, _collateralToken, _indexToken, _isLong, position.size, position.entryFundingRate);
        if (!hasProfit && position.collateral < delta) {
            if (_raise) {
                revert("Vault: losses exceed collateral");
            }
            return (1, marginFees);
        }

        uint256 remainingCollateral = position.collateral;
        if (!hasProfit) {
            remainingCollateral = position.collateral - (delta);
        }

        if(marginFees<0){
            remainingCollateral = remainingCollateral + uint(abs(marginFees));
        } else {
            if (remainingCollateral < uint(marginFees)) {
                if (_raise) {
                    revert("Vault: fees exceed collateral");
                }
            // cap the fees to the remainingCollateral
                return (1, int(remainingCollateral));
            }
            remainingCollateral = remainingCollateral - uint(marginFees);
        }
```

Listing 3.10: `Utils::validateLiquidation()`

Recommendation Revise the above routine to properly validate whether is position can be liquidated or not.

Status This issue has been fixed by the following commit: `ffb5d3d`.

3.10 Inconsistent Vault Token Config Update Logic in TimeLock

- ID: PVE-010
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `TimeLock`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

To standardize the configuration of protocol-wide parameters, `LogX` has a built-in `TimeLock` contract to apply a time delay in the activation of new risk parameters. While reviewing their activation, we notice a specific update should be revisited.

In the following, we show the implementation of the related `signalVaultSetTokenConfig()` and `vaultSetTokenConfig()` routines. Specifically, the former routine indicates the need to update the vault token configuration while the latter makes the change effective after a certain time delay is passed. However, we notice the former computes the configuration hash by including the `_canBeCollateralToken` and `_canBeIndexToken` arguments while the latter does not include these two. As a result, it is unlikely for the two routines to compute the same hash values, which makes the intended update of vault token configuration non-functional.

```
    function signalVaultSetTokenConfig(
        address _vault,
        address _token,
        uint256 _tokenDecimals,
        uint256 _tokenWeight,
        uint256 _minProfitBps,
        uint256 _maxUsdlAmount,
        bool _isStable,
        bool _canBeCollateralToken,
        bool _canBeIndexToken
    ) external onlyAdmin {
        bytes32 action = keccak256(abi.encodePacked(
            "vaultSetTokenConfig",
            _vault,
            _token,
            _tokenDecimals,
            _tokenWeight,
            _minProfitBps,
            _maxUsdlAmount,
            _isStable,
            _canBeCollateralToken,
            _canBeIndexToken
        ));
```

```

        _setPendingAction(action);

        emit SignalVaultSetTokenConfig(
            _vault,
            _token,
            _tokenDecimals,
            _tokenWeight,
            _minProfitBps,
            _maxUsdlAmount,
            _isStable,
            _canBeCollateralToken,
            _canBeIndexToken
        );
    }

    function vaultSetTokenConfig(
        address _vault,
        address _token,
        uint256 _tokenDecimals,
        uint256 _tokenWeight,
        uint256 _minProfitBps,
        uint256 _maxUsdlAmount,
        bool _isStable,
        bool canBeCollateralToken,
        bool canBeIndexToken,
        uint _maxLeverage,
        uint256 _maxLiquidityPerUser,
        uint256 _maxOiImbalance
    ) external onlyAdmin {
        bytes32 action = keccak256(abi.encodePacked(
            "vaultSetTokenConfig",
            _vault,
            _token,
            _tokenDecimals,
            _tokenWeight,
            _minProfitBps,
            _maxUsdlAmount,
            _isStable
        ));

        _validateAction(action);
        _clearAction(action);

        IVault(_vault).setTokenConfig(
            _token,
            _tokenDecimals,
            _minProfitBps,
            _isStable,
            canBeCollateralToken,
            canBeIndexToken,
            _maxLeverage,
```

```
            _maxLiquidityPerUser,
            _maxOiImbalance
        );
    }
```

Listing 3.11: `TimeLock::signalVaultSetTokenConfig()/vaultSetTokenConfig()`

Recommendation Revise the above routines to properly compute the token configuration hash values.

Status This issue has been fixed by the following commit: `256aa22`.

3.11 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-011
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `transfer()` routine and possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. Specifically, the `transfer()` routine does not have a return value defined and implemented. However, the `IERC20` interface has defined the `transfer()` interface with a `bool` return value. As a result, the call to `transfer()` may expect a return value. With the lack of return value of `USDT`'s `transfer()`, the call will be unfortunately reverted.

```
    function transfer(address _to, uint _value) public onlyPayloadSize(2 * 32) {
        uint fee = (_value.mul(basisPointsRate)).div(10000);
        if (fee > maximumFee) {
            fee = maximumFee;
        }
        uint sendAmount = _value.sub(fee);
        balances[msg.sender] = balances[msg.sender].sub(_value);
        balances[_to] = balances[_to].add(sendAmount);
        if (fee > 0) {
            balances[owner] = balances[owner].add(fee);
            Transfer(msg.sender, owner, fee);
        }
        Transfer(msg.sender, _to, sendAmount);
```

```
    }
```

Listing 3.12: USDT::transfer()

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful.

In current implementation, if we examine the `BaseToken::withdrawToken()` routine that is designed to recover the funds that may be accidentally sent to this contract. To accommodate the specific idiosyncrasy, there is a need to use `safeTransfer()`, instead of `transfer()` (line 65).

```
63      // to help users who accidentally send their tokens to this contract
64      function withdrawToken(address _token, address _account, uint256 _amount) external
            override onlyGov {
65          IERC20(_token).transfer(_account, _amount);
66      }
```

Listing 3.13: BaseToken::withdrawToken()

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()/transfer()/transferFrom()`. Note other contracts are also affected, including `YieldToken`, `TimeLock`, `RewardTracker`, `RewardRouter`, `BaseOrderManager`, `OrderManager`, `LLPManager`, and `Vault`.

Status This issue has been fixed by the following commit: `aea82d8`.

3.12 LLP CooldownDuration Bypass in Liquidity Removal

- ID: PVE-012
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `LLPManager`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `LogX` protocol has a `LLPManager` contract that allows the minting and redemption of `LLP`, the platform's liquidity provider token. We notice there is a cooldown duration after minting LLP. The cooldown duration represents the time that needs to pass for the user before it can be redeemed. Our analysis shows that this cooldown enforcement can be bypassed.

To elaborate, we show below the related `_removeLiquidity()` routine. When the intended liquidity is requested for removal, this routine will validate the cooldown duration is passed. However, it can trivially bypassed by transfering the `LLP` to another new account and instructing the new account to perform the liquidity removal – without further being constrained by the cooldown duration.

```
    function _removeLiquidity(
        address _account,
        address _tokenOut,
        uint256 _llpAmount,
        uint256 _minOut,
        address _receiver
    ) private returns (uint256) {
        require(_llpAmount > 0, "LlpManager: invalid _llpAmount");
        require(
            lastAddedAt[_account] + (cooldownDuration) <= block.timestamp,
            "LlpManager: cooldown duration not yet passed"
        );

        // calculate aum before sellusdl
        uint256 aumInusdl = utils.getAumInUsdl(false);
        uint256 llpSupply = IERC20(llp).totalSupply();

        uint256 usdlAmount = (_llpAmount * (aumInusdl)) / (llpSupply);
        uint256 usdlBalance = IERC20(usdl).balanceOf(address(this));
        if (usdlAmount > usdlBalance) {
            IUSDL(usdl).mint(address(this), usdlAmount - (usdlBalance));
        }

        IMintable(llp).burn(_account, _llpAmount);

        IERC20(usdl).transfer(address(vault), usdlAmount);
        uint256 amountOut = vault.sellUSDL(_tokenOut, _receiver);
        require(amountOut >= _minOut, "LlpManager: insufficient output");

        emit RemoveLiquidity(
            _account,
            _tokenOut,
            _llpAmount,
            aumInusdl,
            llpSupply,
            usdlAmount,
            amountOut
        );

        return amountOut;
    }
```

Listing 3.14: `LLPManager::_removeLiquidity()`

Recommendation Revise the `LLP` routine to honor the above cooldown duration.

Status This issue has been resolved by turning on the `LLP's private` mode, which basically disables `LLP` transfers.

3.13 Trust Issue of Admin Keys

- ID: PVE-013
- Severity: Medium
- Likelihood: Medium
- Impact: High
- Target: Multiple Contracts
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the LogX protocol, there is a privileged administrative account owner. The administrative account plays a critical role in governing and regulating the protocol-wide operations. Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the TimeLock contract as an example and show the representative functions potentially affected by the privileges of the administrative account.

```
    function setKeeper(address _keeper, bool _isActive) external onlyAdmin {
        isKeeper[_keeper] = _isActive;
    }

    function setBuffer(uint256 _buffer) external onlyAdmin {
        require(_buffer <= MAX_BUFFER, "Timelock: invalid _buffer");
        require(_buffer > buffer, "Timelock: buffer cannot be decreased");
        buffer = _buffer;
    }

    function setMaxLeverage(address _vault, uint256 _maxLeverage, address _token) external onlyAdmin {
      require(_maxLeverage > MAX_LEVERAGE_VALIDATION, "Timelock: invalid _maxLeverage");
      IVault(_vault).setMaxLeverage(_maxLeverage, _token);
    }

    function setBorrowingRate(address _vault, uint256 _borrowingInterval, uint256 _borrowingRateFactor) external onlyKeeperAndAbove {
        require(_borrowingRateFactor < MAX_BORROWING_RATE_FACTOR, "Timelock: invalid _borrowingRateFactor");
        IVault(_vault).setBorrowingRate(_borrowingInterval, _borrowingRateFactor);
    }

    function setFundingRate(address _vault, uint256 _fundingInterval, uint256 _fundingRateFactor, uint256 _fundingExponent) external onlyKeeperAndAbove {
        require(_fundingRateFactor < MAX_FUNDING_RATE_FACTOR, "Timelock: invalid _fundingRateFactor");
        IVault(_vault).setFundingRate(_fundingInterval, _fundingRateFactor, _fundingExponent);
    }

    function setTokenConfig(
        address _vault,
```

```
        address _token,
        uint256 _minProfitBps,
        uint _maxLeverage,
        uint256 _maxLiquidityPerUser,
        uint256 _maxOiImbalance
    ) external onlyKeeperAndAbove {
        ...
    }
```

Listing 3.15: Example Privileged Operations in `TimeLock`

```
    //function is added only for testing purposes to prevent locking of funds.
    //Main-net will not have this function.
    function withdrawFunds(address _token, uint256 _amount) external onlyAdmin {
        uint balance  = IERC20(_token).balanceOf(address(this));
        require(_amount <= balance,"OrderManager: Requested amount exceeds OrderManager balance");
        IERC20(_token).transfer(admin, _amount);
    }
```

Listing 3.16: Example Privileged Operations in `OrderManager`

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the administrative account may also be a counter-party risk to the protocol users. It would be worrisome if the privileged administrative account is a plain EOA account. 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 account to the intended `DAO`-like governance contract. All changes 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 mitigated with the plan to transfer the privileged account to a `multi-sig` account.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `LogX` protocol, which is a decentralised exchange for trading perpetuals. It is designed to provide lightning fast execution at very low fees and zero price impact. The trading is supported by the `LogX` pool which contains stable coins. Liquidity providers earn based on the performance of the pool and fees collected from trading. The price feeds are supported by dark oracle, which fetches prices from `Pyth` oracles and other centralized exchanges to provide better aggregated prices. The aggregation is done to provide additional safety for liquidity providers. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that `Solidity`-based 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.