SMART CONTRACT AUDIT REPORT

for

Rollup Finance

Prepared By: Xiaomi Huang

PeckShield

October 31, 2023

Document Properties

Client	Rollup
Title	Smart Contract Audit Report
Target	Rollup
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	October 31, 2023	Xuxian Jiang	Final Release
1.0-rc	October 15, 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 source code of the `Rollup` protocol, we outline in this 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 contract can be further improved due to the presence of the identified issues. This document outlines our audit results.

1.1 About Rollup

`Rollup` is a decentralized perpetual derivatives exchange launched initially on `zkSync Era`. The perpetual trading supports various trading modes, allowing users to perform zero-slippage trading and leveraged trading up to `500x`. The goal here is to create a robust and professional multi-dimensional decentralized derivatives exchange where traders can enjoy the scalability and security of zero knowledge roll-ups. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Rollup

Item	Description
Name	Rollup
Website	https://rollup.finance
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	October 31, 2023

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

- https://github.com/rollup-finance/contracts-v2.git (99f7ab2)

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

- https://github.com/rollup-finance/contracts-v2.git (dcf330e)

1.2 About PeckShield

PeckShield Inc. [13] 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

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [12]:

- 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 contract is considered safe regarding the check item. For any discovered issue, we might further deploy contract

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

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 contract 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 contract 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 contract from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [11], 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 contract, 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 Logic	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 `Rollup` 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	2	■■
Medium	5	■■■■■
Low	3	■■■
Informational	0	
Total	10	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of. 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 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, this smart contract is well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 high-severity vulnerabilities, 5 medium-severity vulnerabilities, and 3 low-severity vulnerabilities.

Table 2.1: Key Rollup Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Unauthorized Referrer Update in ReferralStorage	Security Features	Resolved
PVE-002	High	Unauthorized Liquidity Addition in StableVault	Security Features	Resolved
PVE-003	Medium	Possible Underflow in FeeRewardDivider::distribute()	Numeric Errors	Resolved
PVE-004	Medium	Possibly Inaccurate Rate Calculation in RewardDistributor	Business Logic	Resolved
PVE-005	Low	Incorrect _defaultToken() Logic in StableVault	Business Logic	Resolved
PVE-006	Medium	Precision Issue in Hourly Fee Calculation in Trading	Numeric Errors	Resolved
PVE-007	High	Funding Fee Avoidance via Zero-Size Increase Order	Business Logic	Resolved
PVE-008	Low	Lack of Protocol-Wide Risk Parameter Enforcement in Trading	Coding Practice	Confirmed
PVE-009	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-010	Low	Incorrect ReferralStorage Initialization Logic	Init. And Cleanup	Resolved

Beside the identified issues, we emphasize that for any user-facing applications and services, 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 should kick in at the very moment when the contract is being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Unauthorized Referrer Update in ReferralStorage

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: ReferralStorage
- Category: Security Features [6]
- CWE subcategory: CWE-287 [4]

Description

The Rollup protocol has a feature to recognize user referrals and the key logic is implemented in the ReferralStorage contract. In the process of reviewing current referral logic, we notice the related implementation exposes an issue that allows for unauthorized referrer update.

To elaborate, we show below the implementation of the related setCodeOwner() routine. It is designed with the intention to transfer current code ownership to another user. However, it comes to our attention that the new owner may not want to accept this ownership transfer. As a result, any user's referral may be updated without proper authorization.

```
    function setCodeOwner(bytes32 _code, address _newAccount) external {
        if (_code == bytes32(0)) revert("ReferralStorage: invalid _code");

        address account = codeOwners[_code];
        if (_msgSender() != account) revert("ReferralStorage: no owner");

        codeOwners[_code] = _newAccount;
        referrerTiers[_newAccount] = referrerTiers[account];
        referrerDiscountShares[_newAccount] = referrerDiscountShares[account];

        // reset old account
        delete referrerTiers[account];
        delete referrerDiscountShares[account];

        emit SetCodeOwner(msg.sender, _newAccount, _code);
```

```
    }
```

Listing 3.1: `ReferralStorage::setCodeOwner()`

Recommendation Revise the above `setCodeOwner()` logic to ensure the new owner agrees to accept the transferred ownership.

Status The issue has been fixed by the following commit: `185952c`.

3.2 Unauthorized Liquidity Addition in StableVault

- ID: PVE-002
- Severity: High
- Likelihood: High
- Impact: High
- Target: `StableVault`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [4]

Description

The `Rollup` protocol has a core `StableVault` contract that implements a stablecoin vault with marginal assets for the trading of perpetual derivatives. Our analysis on the related liquidity-adding logic shows a flawed implementation that may be exploited to steal funds from approving users.

To elaborate, we show below the implementation of the related `depositFor()` helper routine. While it properly achieves the design goal in allowing users to deposit stable tokens to get the vault share, it misses the needed verification on the caller. As a result, a malicious user may trigger the deposit for another victim user (as far as the user has approved the vault to transfer the supported funds) and the fund is sourced from the victim user. Even worse, the recipient of minted vault share may be arbitrarily specified by the malicious user. In other words, the malicious user may steal funds from approving users by eventually redeeming the vault share.

```
    function depositFor(
        PriceData[] calldata _priceDataList,
        bytes[] calldata _signatureList,
        address _from,
        address _token,
        uint256 _amount,
        address _receiver
    ) public override returns (uint256, uint256) {
        if (!allowed[_token]) revert("StableVault: token not listed");

        _checkAndSetPrice(_receiver, _priceDataList, _signatureList);

        // transfer token to vault
        IERC20(_token).transferFrom(_from, address(this), _amount);
```

```
        // convert stable token to vault unit
        uint256 _uAmount = tokenAmountToStableAmount(_token, _amount);
        uint256 _fee = _uAmount * stakeFeeBasisPoints[0] / PRECISION;

        uint256 _share = toShare(_uAmount - _fee);
        _mint(_receiver, _share);

        emit AddLiquidity(_from, _token, _amount, _share, totalSupply(), lpPrice);

        return (_share, _fee);
    }
```

Listing 3.2: `StableVault::attach()`

Recommendation Verify the caller of the above `depositFor()` routine so that only authorized entity is able to move user funds.

Status The issue has been fixed by the following commit: `185952c`.

3.3 Possible Underflow in FeeRewardDivider::distribute()

- ID: PVE-003
- Severity: Medium
- Likelihood: Low
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Numeric Errors [10]
- CWE subcategory: CWE-190 [3]

Description

The `Rollup` protocol has the built-in logic of distributing rewards to intended recipient. While reviewing the logic to distribute protocol fee rewards, we notice a possible arithmetic underflow issue.

To elaborate, we show below the related `distribute()` routine in `FeeRewardDivider`. The reward distribution is performed based on their percentages with the sum expected to be full in `PRECISION`. We notice it also supports the rebate program, which basically offers discounts back to the trader and effectively reduces the overall sum to be `(PRECISION-leftPercent)/PRECISION`. In other words, after the trader rebate, we need to distribute the rewards with adjusted percentage by scaling down with `(PRECISION-leftPercent)/PRECISION` (lines $50-51$).

```
33    function distribute(address _account, address _vault, uint256 _amount, bytes32 _uuid
        ) external override onlyHandler {
34        if (recipients.length == 0) return;
35        if (_amount == 0) revert("FeeRewardDivider: !amount");

37        uint256 leftPercent = PRECISION;
```

```
        if (_account != address(0)) {
            if (_referral() != address(0)) {
                ReferralInfo memory _rInfo = IReferralStorage(_referral()).getTraderReferralInfo(_account);
                if (_rInfo.rebate > 0) {
                    leftPercent -= (_rInfo.rebate + _rInfo.subRebate);
                    uint256 rebateFee = _amount * (_rInfo.rebate + _rInfo.subRebate) / PRECISION;
                    IReferralStorage(_referral()).distribute(_rInfo.referrer, _account, _vault, rebateFee, _uuid);
                }
            }
        }

        for (uint256 i = 0; i < (recipients.length - 1); i++) {
            uint256 amount = _amount * recipients[i].percent / PRECISION;
            leftPercent -= recipients[i].percent;
            if (amount > 0) {
                IStableVault(_vault).payoutReward(recipients[i].account, amount);
            }
        }

        if (leftPercent > 0) {
            uint256 amount = _amount * leftPercent / PRECISION;
            if (amount > 0) {
                IStableVault(_vault).payoutReward(recipients[recipients.length - 1].account, amount);
            }
        }
    }
```

Listing 3.3: `FeeRewardDivider::distribute()`

Recommendation Properly revise the above routine to take into account the trader rebate. Note a similar underflow issue is also present in other routines, including `_emitIncreasePosition()` and `_emitIncreasePosition()` in the `Trading` contract.

Status The issue has been fixed by the following commit: `185952c`.

3.4 Possibly Inaccurate Rate Calculation in RewardDistributor

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `RewardDistributor`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

As mentioned earlier, `Rollup` has the built-in logic of distributing rewards to intended recipient. In the process of analyzing the core reward-distributing contract `RewardDistributor`, we notice the computed token dissemination rate may be inaccurate.

To elaborate, we show below the code snippet from the `fetchRewardManual()` routine. The routine is intended to be invoked once every `rewardFetchInterval` seconds. However, the computed token dissemination rate is based on the following formula `rewardAmount/rewardFetchInterval`, which may not account for the remaining rewards that have not bee disseminated yet. As a result, the undistributed rewards may be lost in the contract.

```
    function fetchRewardManual() external {
        if (ChainUtils.getTime() < lastRewardFetchTime + rewardFetchInterval) revert("
            RewardDistributor: not time yet");

        uint256 _rewardAmount = IStableVault(vault).claimReward(rewardToken(), 0);
        lastRewardFetchTime = ChainUtils.getTime();
        uint256 _tokensPerInterval = _rewardAmount / rewardFetchInterval;
        _setTokensPerInterval(_tokensPerInterval);
    }
```

Listing 3.4: `RewardDistributor::fetchRewardManual()`

Recommendation Revise the above reward-disseminating routine to compute and use the accurate rate for reward token dissemination.

Status The issue has been fixed by the following commit: `dcf330e`.

3.5 Incorrect _defaultToken() Logic in StableVault

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `StableVault`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

As mentioned earlier, `Rollup` has a core `StableVault` contract that implements a stablecoin vault with marginal assets for the trading of perpetual derivatives. By design, the vault supports a number of underlying tokens. In the process of analyzing the set of supported assets, we notice an implicit assumption that may not be valid.

To elaborate, we show below the implementation of the affected `_defaultToken()` routine. The routine is designed to identify the specific token with the largest balance in the vault. However, it makes an implicit assumption that all these supported tokens have the same decimals. To fix, we need to apply `tokenAmountToStableAmount()` to the balance of each supported token (line 229).

```
224     function _defaultToken() private view returns (address) {
225         uint256 _maxBalance = 0;
226         address _maxToken = address(0);
227         for (uint256 i = 0; i < tokenCount; i++) {
228             address _token = tokens[i];
229             uint256 _balance = IERC20(_token).balanceOf(address(this));
230             if (_balance > _maxBalance) {
231                 _maxBalance = _balance;
232                 _maxToken = _token;
233             }
234         }
235         return _maxToken;
236     }
```

Listing 3.5: `StableVault::_defaultToken()`

Recommendation Revise the above routine to normalize the decimals of supported tokens and make the above implicit assumption explicit.

Status The issue has been fixed by the following commit: `185952c`.

3.6 Precision Issue in Hourly Fee Calculation in Trading

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Trading`
- Category: Numeric Errors [10]
- CWE subcategory: CWE-190 [3]

Description

`SafeMath` is a widely-used Solidity `math` library that is designed to support safe `math` operations by preventing common overflow or underflow issues when working with `uint256` operands. While it indeed blocks common overflow or underflow issues, the lack of `float` support in `Solidity` may introduce another subtle, but troublesome issue: precision loss. In this section, we examine one possible precision loss source that stems from the different orders when both multiplication (`mul`) and division (`div`) are involved.

In particular, we use the `Trading::getHourlyFee()` as an example. This routine is used to calculate the resulting hourly fee of a position.

```
796     function getHourlyFee(Position memory _pos) public view returns (uint256) {
797         uint256 timeDeltaInterval = (ChainUtils.getTime() - _pos.lastUpdated) /
                HOURLY_FEE_INTERVAL;
798         return timeDeltaInterval * getHourlyFeeBasisPoints(_pos.indexAsset) * _pos.
                margin / PRECISION;
799     }
```

Listing 3.6: Trading::getHourlyFee()

We notice the calculation of the resulting fee (line 798) involves mixed multiplication and devision. For improved precision, it is better to calculate the multiplication before the division, i.e., `(ChainUtils.getTime()- _pos.lastUpdated)* getHourlyFeeBasisPoints(_pos.indexAsset)* _pos.margin / PRECISION / PRECISION`. Note that the resulting precision loss may be just a small number, but it plays a critical role when certain boundary conditions are met. And it is always the preferred choice if we can avoid the precision loss as much as possible.

Recommendation Revise the above calculations to better mitigate possible precision loss.

Status The issue has been resolved as it is part of design to have 0 fee if it is less than one hour.

3.7 Funding Fee Avoidance via Zero-Size Increase Order

- ID: PVE-007
- Severity: High
- Likelihood: High
- Impact: High
- Target: `Trading`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

The core trading logic in `Rollup` is implemented in the `Trading` contract. While analyzing the inherent funding rate[1] and its associated fee collection of traders, we notice an issue that may be abused to avoid paying the funding rate fee.

To elaborate, we show below the code snippet from the `increasePosition()` routine. The routine is intended to be invoked if the user intends to create a new position or increase an existing position. We notice the funding fee collection is performed and collected within the `getNextAveragePrice()` helper routine. However, if the user input provides `0 sizeDelta` for the position change, the funding rate of the user postion is still refreshed without the associated fee being collected.

```
    function increasePosition(
        PosInfo calldata _posInfo
    ) external override nonReentrant {
        if (_tradeExt().isPaused()) revert("Trading: trading disabled");

        _validateRouter(_posInfo.account);

        // udpate funding fee
        _updateFundingFee(_posInfo.indexAsset, _posInfo.vault);

        bytes32 key = _getPosKey(_posInfo);
        Position storage position = positions[key];

        uint256 price = _getPrice(_posInfo, true);

        // first time opening a position
        if (position.size == 0) {
            position.averagePrice = price;
            position.indexAsset = _posInfo.indexAsset;
            position.isLong = _posInfo.isLong;
            position.vault = _posInfo.vault;
        }

        if (position.size > 0 && _posInfo.sizeDelta > 0) {
```

[1] The funding rate represents the difference between the mark price of the perpetual futures market and the index price, which is equivalent to the spot market of the underlying asset. The funding rate ensures that the funding mechanism aligns the futures market price with the index price.

```
            // update average price for exists position
            position.averagePrice = getNextAveragePrice(
                position,
                price,
                _posInfo.sizeDelta
            );
        }

        // use vault uni stable token as margin asset
        if (_posInfo.sizeDelta > 0) {
            // check min position size delta
            if (_posInfo.sizeDelta < _tradeExt().minPos(_posInfo.vault)) revert("Trading
                : min pos size delta not met");
        }

        // collect fee
        uint256 fee = _collectIncreasePositionFee(position, _posInfo);

        // update position
        uint256 _oriMargin = position.margin;
        position.margin += _posInfo.marginDelta;
        if (position.margin < fee) revert("Trading: margin not enough for fee");
        position.margin -= fee;

        position.entryFundingRate = _tradeExt().getAccInterest(_posInfo.indexAsset,
            _posInfo.vault, _posInfo.isLong);
        position.size += _posInfo.sizeDelta;
        position.lastUpdated = ChainUtils.getTime();

        if (position.size == 0) revert("Trading: position size is 0");
        _validatePosition(position.size, position.margin);
        validateLiquidation(_posInfo, true);

        // modify oi and borrow asset
        _updateOi(_posInfo, true);

        _emitIncreasePosition(key, _posInfo, price, fee, _oriMargin);
    }
```

Listing 3.7: `Trading::increasePosition()`

Recommendation Revise the above routine to reliably compute and collect funding fees for each user position.

Status The issue has been fixed by the following commit: `eeb09d5`.

3.8 Lack of Protocol-Wide Risk Parameter Enforcement in Trading

- ID: PVE-008
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `Rollup` protocol is no exception. Specifically, if we examine the `PairsContract` contract, it has defined a number of protocol-wide risk parameters, such as `minLeverage` and `maxLeverage`. In the following, we show the corresponding routines that allow for their changes.

```
function addAsset(
    uint256 _assetId,
    Asset memory _asset
)
    external onlyOwner
{
    bytes memory _assetSymbol = bytes(_idToAsset[_assetId].symbol);
    if (_assetSymbol.length > 0) revert("PairsContract: asset already exists");
    if (bytes(_asset.symbol).length == 0) revert("PairsContract: bad symbol");
    if (_asset.minLeverage > _asset.maxLeverage  _asset.minLeverage < MIN_LEVERAGE)
        revert("PairsContract: bad leverage");
    if (_asset.baseFundingRate > maxBaseFundingRate) revert("PairsContract: base
        funding rate too high");
    if (_asset.spread > MAX_SPREAD) revert("PairsContract: spread too high");
    if (_asset.maxLoss > maxLoss) revert("PairsContract: max loss too high");
    if (_asset.maxProfit > maxProfit) revert("PairsContract: max profit too high");

    allowedAsset[_assetId] = true;
    _idToAsset[_assetId].symbol = _asset.symbol;
    _idToAsset[_assetId].groupId = _asset.groupId;

    _idToAsset[_assetId].minLeverage = _asset.minLeverage;
    _idToAsset[_assetId].maxLeverage = _asset.maxLeverage;
    _idToAsset[_assetId].baseFundingRate = _asset.baseFundingRate;
    _idToAsset[_assetId].spread = _asset.spread;

    _idToAsset[_assetId].maxLoss = _asset.maxLoss == 0 ? maxLoss : _asset.maxLoss;
    _idToAsset[_assetId].maxProfit = _asset.maxProfit == 0 ? maxProfit : _asset.
        maxProfit;

    _idToAsset[_assetId].maxOi = _asset.maxOi;
    _idToAsset[_assetId].userOiLimit = _asset.userOiLimit;
```

```
        _idToAsset[_assetId].feeBasisPoint = _asset.feeBasisPoint;
        _idToAsset[_assetId].maxPriceFluctuation = _asset.maxPriceFluctuation;

        emit AssetAdded(_assetId, _asset.symbol, _asset.groupId);
    }

    /**
     * @dev Update the leverage allowed per asset
     * @param _assetId index of the asset
     * @param _minLeverage minimum leverage allowed
     * @param _maxLeverage Maximum leverage allowed
     */
    function updateAssetLeverage(uint256 _assetId, uint256 _minLeverage, uint256 _maxLeverage) external onlyOwner {
        if (_maxLeverage > 0) {
            _idToAsset[_assetId].maxLeverage = _maxLeverage;
        }
        if (_minLeverage >= MIN_LEVERAGE) {
            _idToAsset[_assetId].minLeverage = _minLeverage;
        }

        if (_idToAsset[_assetId].maxLeverage < _idToAsset[_assetId].minLeverage) revert("PairsContract: bad leverage");
    }
```

Listing 3.8: PairsContract::addAsset() and PairsContract::updateAssetLeverage()

These parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when configuring or updating them. Our analysis shows the update logic on these parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, we notice each position update will be subject to necessary validation of the above risk parameters. However, such validation is currently missing in the `_validatePosition()` routine.

```
    function _validatePosition(uint256 _size, uint256 _margin) private pure {
        if (_size == 0) {
            if (_margin > 0) revert("Trading: size=0,margin>0");
            return;
        }
        if (_size < _margin) revert("Trading: size<margin");
    }
```

Listing 3.9: Trading::_validatePosition()

Recommendation Properly validate protocol-wide risk parameters once a user position is updated.

Status The issue has been confirmed.

3.9 Trust Issue of Admin Keys

- ID: PVE-009
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [6]
- CWE subcategory: CWE-287 [4]

Description

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

```
    function setAddr(AddressType _type, address _addr) external onlyOwner {
        if (_addr == address(0)) revert("Config: !_addr");
        _addresses[_type] = _addr;
    }

    // set uint
    function setUint(UintType _type, uint256 _value) external onlyOwner {
        _uints[_type] = _value;
    }

    function setVault(address _vault, bool _allowed) external onlyOwner {
        if (_vault == address(0)) revert("Config: !_vault");
        isAllowedVault[_vault] = _allowed;
    }
```

Listing 3.10: Example Privileged Operations in Config

```
    function setNode(address _node, bool _isNode) external onlyOwner {
        isNode[_node] = _isNode;
    }

    /**
     * @dev changes the minimum position size
     * @param  _vault vault
     * @param _min minimum position size 18 decimals
     */
    function setMinPositionSize(
        address  _vault,
        uint256 _min
    )
        external
        onlyOwner
    {
```

```
        minPositionSize[_vault] = _min;
    }

    function setPaused(bool _paused) external onlyOwner {
        paused = _paused;
    }
```

Listing 3.11: Example Privileged Operations in `TradingExtension`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to these privileged account may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

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 The issue has been mitigated as the team confirm they are using a multi-sig account as the `owner`.

3.10 Incorrect ReferralStorage Initialization Logic

- ID: PVE-010
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `ReferralStorage`
- Category: Initialization and Cleanup [9]
- CWE subcategory: CWE-1188 [2]

Description

The `Rollup` protocol has a number of contracts and many of them have a `initialize()` function which is used to set up a number of key parameters. Using the `ReferralStorage` contract as an example, its `initialize()` function is used to configure the default referral tier. To facilitate our discussion, we show below the related code snippet.

```
    function initialize(
        address _cfg,
        uint256 _defaultRebate,
        uint256 _defaultDiscountShare
    ) external initializer {
        __Ownable_init();
```

```
        if (_defaultRebate > PRECISION) revert("ReferralStorage: invalid _defaultRebate"
            );
        if (_defaultDiscountShare > PRECISION) revert("ReferralStorage: invalid
            _defaultDiscountShare");

        cfg = IConfig(_cfg);
        isPrivateMode = true;

        // default tiers 0
        tiers[0] = Tier(_defaultRebate, _defaultDiscountShare, 0);
        tierCount = 0;
    }
```

Listing 3.12: `ReferralStorage::initialize()`

The above logic ensures the initialization can be called only once. However, we notice the `tierCount` is initialized to be 0, which does not take into account the default tier. In other words, we need to initialize it to be 1.

Recommendation Revise the `initialize()` function to properly configure the `tierCount` state. Otherwise, the default tier will be overwritten when there is an `addTier()` call.

Status The issue has been fixed by the following commit: `185952c`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Rollup` protocol, which a decentralized perpetual derivatives exchange launched initially on `zkSync Era`. The perpetual trading supports various trading modes, allowing users to perform zero-slippage trading and leveraged trading up to `500x`. The goal here is to create a robust and professional multi-dimensional decentralized derivatives exchange where traders can enjoy the scalability and security of zero knowledge roll-ups. The current code base is well organized and 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-1188: Insecure Default Initialization of Resource. https://cwe.mitre.org/data/definitions/1188.html.

[3] MITRE. CWE-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

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

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

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

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

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

[9] MITRE. CWE CATEGORY: Initialization and Cleanup Errors. https://cwe.mitre.org/data/definitions/452.html.

[10] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

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

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

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