SMART CONTRACT AUDIT REPORT

for

OT Perpetual Protocol

Prepared By: Xiaomi Huang

PeckShield

May 8, 2023

Document Properties

Client	Onchain Trade
Title	Smart Contract Audit Report
Target	Perpetual Protocol
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	May 8, 2023	Xuxian Jiang	Final Release
1.0-rc	May 4, 2023	Luck Hu	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 smart contract source code of the OT Perpetual 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 Perpetual Protocol

The Perpetual protocol of Onchain Trade provides up to 70x leverage for traders to trade perpetual futures on chain. It introduces the trading fee and the funding fee to balance the longs and shorts for each trading pair, reducing the risk of draining insurance pool. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Perpetual Protocol

Item	Description
Name	Onchain Trade
Website	https://onchain.trade
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 8, 2023

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

- https://github.com/1kxexchange/1kx-v1/tree/masterWithOutAudit/contracts/future (495e6cd)

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

- https://github.com/1kxexchange/1kx-v1/tree/masterWithOutAudit/contracts/future (68171b4)

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 accordingly classified into four categories, 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

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

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. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

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 `OT Perpetual` implementations. 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	4	■■■■
Low	2	■■
Informational	0	
Total	8	

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

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Lack of Slippage Control in FutureRouter::_decreasePosition()	Coding Practices	Mitigated
PVE-002	High	Revisited Validation of maxLeverage in decreaseMargin()	Business Logic	Fixed
PVE-003	High	Proper Validation of sizeDelta in increasePosition()	Business Logic	Fixed
PVE-004	Medium	Improved Calculation of New Margin in validateLiquidate()	Business Logic	Fixed
PVE-005	Low	Revisited Validation of Pair Status in liquidatePosition()	Business Logic	Confirmed
PVE-006	Medium	Revised Withdrawal of ETH in cancelIncreaseOrder()	Coding Practices	Fixed
PVE-007	Medium	Revised Logic in decreasePositionETH()	Coding Practices	Fixed
PVE-008	Medium	Trust Issue on Admin Keys	Security Features	Mitigated

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 contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Lack of Slippage Control in _decreasePosition()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `FutureRouter`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1041 [1]

Description

In the `Perpetual` protocol, the `FutureRouter` contract is a router that facilitates users trading with the `Future/FutureLimit` contracts. In particular, it provides an interface for a position owner to decrease or close the position, and converts the received underlying token to the desired token specified by the user. While examining the swap from the underlying token to the desired token, we notice the current implementation lacks an effective slippage control.

To elaborate, we show below the related `_decreasePosition()` routine, which is used for position owners to decrease or close their positions. If the position owner desires a different token with the underlying token, it calls the `IFuture(future).decreasePositionByRatio()` routine (line 655) to decrease the position and receives the underlying token to the current contract, i.e., `address(this)`. Then it calls the `ISwapForFuture(swapPool).swapIn()` routine (line 675) to convert the received underlying token to the desired token.

In order to protect the conversion from possible loss, it requires the caller to provide a minimum acceptable amount of the desired token, i.e., `amountOutMin` (line 18). However, we notice that it takes 0 as the value of `amountOutMin` (line 679) in the call to the `ISwapForFuture(swapPool).swapIn()` routine, which means there is no effective slippage control for the conversion.

Based on this, we suggest to provide a reasonable `amountOutMin` value to protect users funds from possible loss.

```
644     function _decreasePosition(
645         address _collateralToken,
```

```
        address _indexToken,
        bool _isLong,
        uint256 _marginDelta,
        uint256 _notionalDelta,
        uint256 _collateralPrice,
        uint256 _indexPrice,
        address _receiver,
        address _tokenOut
    ) private returns (uint256) {
        require(msg.sender == _receiver, "Invalid caller");
        if (_collateralPrice > 0) {
            require(IFuture(future).getPrice(_collateralToken) >= _collateralPrice, "price_limit");
        }
        if (_indexPrice > 0) {
            require(IFuture(future).getPrice(_indexToken) <= _indexPrice, "price_limit");
        }
        if (address(tradeStakeUpdater) != address(0)) {...}

        if (_tokenOut != _collateralToken) {
            uint256 _amountOut = IFuture(future).decreasePositionByRatio(
                _collateralToken,
                _indexToken,
                msg.sender,
                _isLong,
                _notionalDelta,
                address(this)
            );
            IERC20(_collateralToken).approve(swapPool, _amountOut);//Luck1: safeApprove
            return
                ISwapForFuture(swapPool).swapIn(
                    _collateralToken,
                    _tokenOut,
                    _amountOut,
                    0,
                    _receiver,
                    0
                );
        }
        ...
```

Listing 3.1: _decreasePosition()

```
function swapIn(
    address tokenIn,
    address tokenOut,
    uint256 amountIn,
    uint256 amountOutMin,
    address to,
    uint256 deadline
```

```
) external returns (uint256);
```

Listing 3.2: **interface** ISwapForFuture

Recommendation Revisit the `FutureRouter::_decreasePosition()` routine and provide a reasonable `amountOutMin` value in the call to the `ISwapForFuture(swapPool).swapIn()` routine.

Status The issue has been mitigated in below commits by supporting only stable coins as the source and destination tokens: `bfbb515` and `3518e0b`.

3.2 Revisited Validation of maxLeverage in decreaseMargin()

- ID: PVE-002
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `Future`
- Category: Business Logic [6]
- CWE subcategory: CWE-837 [3]

Description

The `Perpetual` protocol allows users to trade perpetual futures within the maximum allowed leverage. Any operation that may exceed the max leverage is forbidden. While reviewing the margin decrease via the `_decreasePosition()/decreaseMargin()` routines, we notice they do not properly check if the new leverage has exceeded the maximum leverage or not.

To elaborate, we show below the code snippet of the `Future::decreaseMargin()` routine. The routine calculates the funding fee and the `PnL` first (line 412), and gets the left margin in the position (line 414). Then it updates the new `margin/openNotinal` of the position (lines 417−418) and transfers the margin delta to the user in the `positionSettlement()` routine (line 423).

After the update of the `margin/openNotinal`, the leverage also changes. However, it comes to our attention that it does not properly check if the new leverage exceeds the max leverage or not. As a result, the user can still decrease margin from a position where the new leverage may exceed the max leverage.

At the end of the routine, we notice it checks whether the new position can be liquidated or not by calling the `validateLiquidate()` routine (line 428), which ensures the new margin ratio cannot exceed the maintenance margin ratio. However, it cannot ensure the new leverage is in the allowed range even the new position cannot be liquidated. With that, we suggest to validate the new leverage at the end of the `decreaseMargin()` routine.

Note the same issue is also applicable to the `_decreasePosition()` routine.

```
    function decreaseMargin(
        address _collateralToken,
        address _indexToken,
        address _account,
        bool _isLong,
        uint256 _marginDelta,
        address _receiver
    ) external override nonReentrant {
        _validateRouter(_account);
        require(_account == _receiver, "Invalid caller");
        validateLiquidate(_collateralToken, _indexToken, _account, _isLong, true);

        _updateFundingFeeRate(_collateralToken, _indexToken, _isLong, 0, 0);

        bytes32 posKey = getPositionKey(_collateralToken, _indexToken, _account, _isLong
            );
        Position storage pos = _getPosition(_collateralToken, _indexToken, _account,
            _isLong);

        _validatePositionExist(pos);
        (int256 fundingFee, , int256 pnl, , ) = _calcNewPosition(...);

        uint256 leftMargin = uint256(int256(pos.margin) - fundingFee + pnl);
        require(leftMargin > _marginDelta, "margin_delta_exceed");

        pos.margin = leftMargin - _marginDelta;
        pos.openNotional = uint256(token1ToToken2(_indexToken, int256(pos.size),
            _collateralToken));
        pos.entryFundingRate = getCumulativeFundingRate(_collateralToken, _indexToken,
            _isLong);
        pos.entryCollateralPrice = getPrice(_collateralToken);
        pos.entryIndexPrice = pos.openNotional/pos.size;

        positionSettlement(...);
        emit DecreaseMargin(...);
        emit UpdatePosition(...);

        // todo replace by validatePosition, validate max usd per position
        validateLiquidate(_collateralToken, _indexToken, _account, _isLong, true);
    }
```

Listing 3.3: decreaseMargin()

Recommendation Properly validate the new leverage of the position at the end of the `decreaseMargin()` routine, and ensure it does not exceed the max allowed leverage.

Status The issue has been fixed by these commits: `c6132c1` and `ba556bf`.

3.3 Proper Validation of Size Delta in increasePosition()

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

Description

The `Future` contract provides an interface, i.e., `increasePosition()`, for users to open new positions or increase the already existing positions. While reviewing the calculation of the size delta, we notice there is a lack of validation for the size delta to ensure the size delta cannot be 0.

To elaborate, we show below the related code snippet of the `Future::increasePosition()` routine. This routine accepts a notional delta, i.e., `_notionalDelta`, that the trader wants to increase. Based on the notional delta and the current prices of the collateral/index tokens, it calculates the corresponding position size delta, i.e., `sizeDelta` (line 484). The position is then updated with the new increased notional delta/size delta (lines $509-510$).

However, it comes to our attention that the routine does not properly validate the size delta. In particular, if the calculated size delta is 0, the trader can keep the position size unchanged while increasing the open notional. As a result, the position is messed up and the trader may lose for a long position or get profit for a short position. Accordingly, we suggest to properly validate the calculated size delta and ensure it is a valid value (not 0).

```
469     function increasePosition(
470         address _collateralToken,
471         address _indexToken,
472         address _account,
473         bool _isLong,
474         uint256 _notionalDelta
475     ) public override nonReentrant {
476         _validateRouter(_account);
477         require(getPairStatus(_collateralToken, _indexToken) == PairStatus.list, "
                pair_unlist");
478         validateLiquidate(_collateralToken, _indexToken, _account, _isLong, true);

480         _updateFundingFeeRate(...);

482         Position storage pos = _getPosition(_collateralToken, _indexToken, _account,
                _isLong);
483         uint256 marginDelta = _transferIn(_collateralToken);
484         uint256 sizeDelta = uint256(
485             token1ToToken2(_collateralToken, int256(_notionalDelta), _indexToken)
486         );
```

```
        (int256 fundingFee, uint256 tradingFee, , , ) = _calcNewPosition(
            _collateralToken,
            _indexToken,
            _account,
            _isLong,
            _notionalDelta,
            sizeDelta,
            true
        );

        {
            _increaseTotalSize(_collateralToken, _indexToken, _isLong, sizeDelta);
            _increaseTotalOpenNotional(_collateralToken, _indexToken, _isLong,
                _notionalDelta);

            int256 remainMargin = int256(pos.margin) +
                int256(marginDelta) -
                fundingFee -
                int256(tradingFee);
            require(remainMargin > 0, "insuff_margin");

            pos.margin = uint256(remainMargin);
            pos.openNotional = pos.openNotional + _notionalDelta;
            pos.size = pos.size + sizeDelta;
            pos.entryFundingRate = getCumulativeFundingRate(_collateralToken,
                _indexToken, _isLong);
            pos.entryCollateralPrice = getPrice(_collateralToken);
            pos.entryIndexPrice = pos.openNotional/pos.size;
        }
        ...
    }
```

Listing 3.4: Future:: increasePosition ()

Note the same issue is also applicable to the `_decreasePosition()` routine where the decreased size delta cannot be 0.

Recommendation Properly validate the calculated size delta and ensure it is a valid value (not 0).

Status The issue has been fixed by these commits: `23510a3` and `cb17f37`.

3.4 Improved Calculation of New Margin in validateLiquidate()

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

Description

In the Perpetual protocol, the Future contract timely checks if the position can be liquidated or not before each trading operation. If the position can be liquidated, it refuses any trading operation except for the liquidation on the position. While reviewing the logic to check if the position can be liquidated or not, we notice it makes use of wrong parameters values to calculate the remain margin and the margin ratio of the position.

In the following, we show the related code snippets of the validateLiquidate()/_calcNewPosition() routines. As the name indicates, the validateLiquidate() routine is used to check if the given position can be liquidated or not. It calls the _calcNewPosition() routine (line 1086) to calculate the current remain margin and the current margin ratio of the position. If the remain margin is smaller than 0 or if the margin ratio is smaller than the maintenance margin ratio of the position, the position should be liquidated.

In the _calcNewPosition() routine, it uses the _notionalDelta parameter to calculate the trading fee for the current trading and uses the _sizeDelta parameter to calculate the trading fee rate. In the validateLiquidate() routine, it calls the _calcNewPosition() routine by taking the open notional (pos.openNotional) as the value of the _notionalDelta parameter (line 1091) and the position size (pos.size) as the value of the _sizeDelta parameter (line 1092). However, it comes to our attention that the validateLiquidate() is a read-only routine that will not impact the open notional or the position size. So it shall not charge any trading fee for the validation and our analysis shows that it shall take 0 as the values of both the _notionalDelta/_sizeDelta parameters in the call to the _calcNewPosition() routine.

```
1072     function validateLiquidate(
1073         address _collateralToken,
1074         address _indexToken,
1075         address _account,
1076         bool _isLong,
1077         bool _raise
1078     ) public view override returns (bool shouldLiquidate) {
1079         bytes32 posKey = getPositionKey(_collateralToken, _indexToken, _account, _isLong
             );
1080         Position storage pos = positions[posKey];
1081
```

```
        if (pos.size == 0) {
            return false;
        }

        (, , , int256 remainMargin, int256 marginRatio) = _calcNewPosition(
            _collateralToken,
            _indexToken,
            _account,
            _isLong,
            pos.openNotional,
            pos.size,
            false
        );

        if (remainMargin < 0) {
            shouldLiquidate = true;
            if (_raise) {
                revert("should_liquidate");
            }
        } else {
            uint256 mantainanceMarginRatio = IFutureUtil(futureUtil).
                getMaintanenceMarginRatio(
                _collateralToken,
                _indexToken,
                _account,
                _isLong
            );
            if (marginRatio < int256(mantainanceMarginRatio)) {
                shouldLiquidate = true;
                if (_raise) {
                    revert("should_liquidate");
                }
            }
        }
    }
```

Listing 3.5: `Future::validateLiquidate()`

```
    function _calcNewPosition(
        address _collateralToken,
        address _indexToken,
        address _account,
        bool _isLong,
        uint256 _notionalDelta, // for trading fees
        uint256 _sizeDelta,
        bool _isIncreasePosition // if is increasing, calc funding fee for
            _notionalDelta
    )
        private
        view
        returns (
            int256 fundingFee,
            uint256 tradingFee,
```

```
            int256 pnl,
            int256 remainMargin,
            int256 marginRatio
        )
    {
        bytes32 posKey = getPositionKey(_collateralToken, _indexToken, _account, _isLong
            );
        Position storage pos = positions[posKey];

        tradingFee = calculateTradingFee(_collateralToken, _indexToken, _isLong,
            _notionalDelta, _sizeDelta, _isIncreasePosition);

        fundingFee = calculateFundingFee(
            _collateralToken,
            _indexToken,
            pos,
            _isLong,
            _notionalDelta,
            _isIncreasePosition
        );
        ...
    }
```

Listing 3.6: Future::_calcNewPosition()

Recommendation Revisit the `validateLiquidate()` routine and take 0 as the values of both the `_notionalDelta/_sizeDelta` parameters in the call to the `_calcNewPosition()` routine.

Status The issue has been fixed by this commit: `b02c299`.

3.5 Revisited Validation of Pair Status in liquidatePosition()

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

Description

As mentioned in Section 3.4, the `Future` contract checks if the position can be liquidated or not before each trading operation. If the position can be liquidated, it allows some liquidator to liquidate the position. While reviewing the logic to liquidate a position, we notice it does not properly validate the pair status to ensure the pair is not unlisted.

To elaborate, we show below the code snippet of the `Future::liquidatePosition()` routine. As the name indicates, it is used for the liquidator to liquidate a position. It validates the open notional of the position (line 928) to ensure the position is existing. Then it calls the `validateLiquidate()` routine (line 939) to check if the position can be liquidated or not. If the position is existing and can be liquidated, it closes the position.

However, we notice the pair of the position may have been unlisted. In this case, there is no need to liquidate the position. Our analysis shows that we can add a validation to ensure the pair is not unlisted at the beginning of the `Future::liquidatePosition()` routine.

```
921     function liquidatePosition(
922         address _collateralToken,
923         address _indexToken,
924         address _account,
925         bool _isLong
926     ) public override nonReentrant {
927         Position storage pos = _getPosition(_collateralToken, _indexToken, _account,
                _isLong);
928         require(pos.openNotional > 0, "position_not_exist");
929
930         _updateFundingFeeRate(
931             _collateralToken,
932             _indexToken,
933             _isLong,
934             -int256(pos.openNotional),
935             -int256(pos.size)
936         );
937
938         {
939             bool shouldLiquidate = validateLiquidate(
940                 _collateralToken,
941                 _indexToken,
942                 _account,
943                 _isLong,
944                 false
945             );
946             require(shouldLiquidate, "position_cannot_liquidate");
947         }
948         ...
949     }
```

Listing 3.7: `Future::liquidatePosition()`

Recommendation Revisit the `Future::liquidatePosition()` routine and add a validation to ensure the pair of the position is not unlisted.

Status The issue has been confirmed by the team that this is designed behavior to allow liquidation when the pair status is `list/stop_open/stop`.

3.6 Revised Withdrawal of ETH in cancelIncreaseOrder()

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: FutureRouter
- Category: Coding Practices [5]
- CWE subcategory: CWE-1041 [1]

Description

In the Perpetual protocol, the FutureRouter contract is a router that facilitates user trading with the Future/FutureLimit contracts. To facilitate user trading by directly using the native token, i.e., ETH, it automatically converts between ETH and WETH. When WETH is to be refunded to the user, it withdraws ETH from WETH and transfers ETH to the user. While reviewing the withdrawal of ETH in the cancelIncreaseOrder() routine, we notice it uses the futureLimit as WETH by mistake.

In the following, we show the code snippet of the FutureRouter::cancelIncreaseOrder() routine, which is used to cancel an existing IncreaseOrder. If the collateral token of the order is WETH, it cancels the order and is expected to withdraw ETH from WETH. However, we notice it calls IWETH (futureLimit).withdraw() (line 304) to withdraw ETH, which will fail. It shall be corrected to call IWETH(weth).withdraw() to withdraw ETH.

```
294     function cancelIncreaseOrder(uint256 _orderIndex) public {
295         (address collateralToken, , , , uint256 _marginDelta, , , ) = IFutureLimit(
                futureLimit)
296             .getIncreaseOrder(msg.sender, _orderIndex);
297         if (collateralToken == weth && _marginDelta > 0) {
298             IFutureLimit(futureLimit).cancelIncreaseOrder(
299                 msg.sender,
300                 _orderIndex,
301                 address(this),
302                 payable(msg.sender)
303             );
304             IWETH(futureLimit).withdraw(_marginDelta);
305             _transferOutETH(_marginDelta, payable(msg.sender));
306         } else {...}
307     }
```

Listing 3.8: FutureRouter::cancelIncreaseOrder()

Recommendation Revisit the cancelIncreaseOrder() routine and withdraw ETH from WETH.

Status The issue has been fixed by this commit: 61695e9.

3.7 Revised Logic in decreasePositionETH()

- ID: PVE-007
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: FutureRouter
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

As mentioned in Section 3.6, the FutureRouter contract is a router that facilitates user trading using ETH directly with the protocol. The refund in WETH will be converted to ETH and transferred to the user. While reviewing the refund from the decrease of a WETH position, we notice a possible denial-of-service issue within current implementation.

In the following, we show the related code snippets of the decreasePositionETH()/_decreasePosition() routines. As the name indicates, the decreasePositionETH() routine is used to decrease a WETH position and refund ETH to the position owner. In order to withdraw ETH form WETH, it is expected to withdraw WETH to the current contract first. So the _receiver of the _decreasePosition() routine is set to the current contract address, i.e., address(this) (line 498).

However, we notice that in the _decreasePosition() routine, it requires the _receiver must be the position owner, i.e., msg.sender (line 655). As a result, the validation of the _receiver will fail and the transaction reverts. Note the same issue is also applicable to the decreaseMarginETH() routine where the call to the Future::decreaseMargin() routine requires the _receiver must be the position owner.

What's more, in the _decreasePosition() routine, the last parameter, i.e., _tokenOut, gives the desired token the user wants to receive from the position decrease. However, in the _decreasePosition() routine, it uses address(this) (line 499) as the desired token, though the current contract is not a token contract.

```
479     function decreasePositionETH(
480         address _collateralToken,
481         address _indexToken,
482         bool _isLong,
483         uint256 _marginDelta,
484         uint256 _notionalDelta,
485         uint256 _collateralPrice,
486         uint256 _indexPrice,
487         address payable _receiver
488     ) external {
489         require(_collateralToken == weth, "invalid_collateral");
490         uint256 amountOut = _decreasePosition(
491             _collateralToken,
```

```
            _indexToken,
            _isLong,
            _marginDelta,
            _notionalDelta,
            _collateralPrice,
            _indexPrice,
            address(this),
            address(this)
        );
        IWETH(weth).withdraw(amountOut);
        _receiver.sendValue(amountOut);
    }
```

Listing 3.9: `FutureRouter::decreasePositionETH()`

```
    function _decreasePosition(
        address _collateralToken,
        address _indexToken,
        bool _isLong,
        uint256 _marginDelta,
        uint256 _notionalDelta,
        uint256 _collateralPrice,
        uint256 _indexPrice,
        address _receiver,
        address _tokenOut
    ) private returns (uint256) {
        require(msg.sender == _receiver, "Invalid caller");
        if (_collateralPrice > 0) {
            require(IFuture(future).getPrice(_collateralToken) >= _collateralPrice, "
                price_limit");
        }
        if (_indexPrice > 0) {
            require(IFuture(future).getPrice(_indexToken) <= _indexPrice, "price_limit")
                ;
        }
        if  (address(tradeStakeUpdater) != address(0)) {...}

        if (_tokenOut != _collateralToken) {
            uint256 _amountOut = IFuture(future).decreasePositionByRatio(
                _collateralToken,
                _indexToken,
                msg.sender,
                _isLong,
                _notionalDelta,
                address(this)
            );
            IERC20(_collateralToken).approve(swapPool, _amountOut);
            return
                ISwapForFuture(swapPool).swapIn(
                    _collateralToken,
                    _tokenOut,
                    _amountOut,
                    0,
```

```
                    _receiver,
                    0
                );
        }
        ...
    }
```

Listing 3.10: `FutureRouter::_decreasePosition()`

Recommendation Revise the `decreasePositionETH()/decreaseMarginETH()` routines and ensure they can receive `WETH` into the `FutureRouter` contract and a valid desired token address can be provided to the `_decreasePosition()` routine.

Status The issue has been fixed by this commit: `91f7ea1`.

3.8 Trust Issue on Admin Keys

- ID: PVE-008
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `Perpetual` protocol, there is a privileged account, i.e., `owner`, that plays a critical role in regulating the protocol-wide operations (e.g., add future pair, set trading fee rate for a pair). Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the `Future` contract as an example and show the representative functions potentially affected by the privileges of the `owner` account.

Specifically, the privileged functions in `Future` allow for the `owner` to update a specific pair, e.g., set the max total sizes, set the pair status, set the trading fee rate, set the max leverage, set the max/min maintenance margin ratio, etc. What's more, the `owner` can set the price feed and withdraw funds from the insurance pool, etc.

```
    function setMaxTotalSize(
        address _collateralToken,
        address _indexToken,
        uint256 _maxLongSize,
        uint256 _maxShortSize
    ) external onlyOwner {
        bytes32 pairKey = getPairKey(_collateralToken, _indexToken);

        maxTotalLongSizes[pairKey] = _maxLongSize;
```

```
        maxTotalShortSizes[pairKey] = _maxShortSize;
        emit UpdateMaxTotalSize(...);
    }

    // set pair status
    function setPairStatus(
        address _collateralToken,
        address _indexToken,
        PairStatus _status
    ) external onlyOwner {
        bytes32 pairKey = getPairKey(_collateralToken, _indexToken);
        PairStatus oldStatus = pairs[pairKey].status;

        require(oldStatus != PairStatus.unlist, "wrong_old_status");

        if (_status == PairStatus.stop_open) {
            require(
                oldStatus == PairStatus.list  oldStatus == PairStatus.stop,
                "wrong_old_status"
            );
        } else if (_status == PairStatus.stop) {
            require(oldStatus == PairStatus.stop_open, "wrong_old_status");
        } else if (_status == PairStatus.list) {} else {
            revert("wrong_status");
        }
        pairs[pairKey].status = _status;
    }

    // set tradingFeeRate for a specific pair
    function setTradingFeeRate(
        address _collateralToken,
        address _indexToken,
        uint256 tradingFeeRate
    ) external onlyOwner {
        bytes32 key = getPairKey(_collateralToken, _indexToken);

        require(tradingFeeRate < FutureMath.TRADING_FEE_RATE_PRECISION, "invalid_rate");
        tradingFeeRates[key] = tradingFeeRate;

        emit UpdateTradingFeeRate(...);
    }

    // set max leverage
    function setMaxLeverage(
        address _collateralToken,
        address _indexToken,
        uint256 maxPositionUsdWithMaxLeverage,
        uint256 maxLeverage
    ) external onlyOwner {
        bytes32 key = getPairKey(_collateralToken, _indexToken);

        require(maxLeverage >= FutureMath.LEVERAGE_PRECISION, "invalid_leverage");
```

```
        require(maxPositionUsdWithMaxLeverage > 0, "invalid_usd_value");
        maxPositionUsdWithMaxLeverages[key] = maxPositionUsdWithMaxLeverage;
        maxLeverages[key] = maxLeverage;

        emit UpdateMaxLeverage(...);
    }

    // set max/min maintenance margin ratio
    function setMarginRatio(
        address _collateralToken,
        address _indexToken,
        uint256 _minMaintanenceMarginRatio,
        uint256 _maxMaintanenceMarginRatio
    ) external onlyOwner {
        bytes32 key = getPairKey(_collateralToken, _indexToken);
        require(_minMaintanenceMarginRatio <= _maxMaintanenceMarginRatio, "
            invalid_min_max");
        require(_minMaintanenceMarginRatio > 0, "invalid_margin_ratio");

        minMaintanenceMarginRatios[key] = _minMaintanenceMarginRatio;
        maxMaintanenceMarginRatios[key] = _maxMaintanenceMarginRatio;

        emit UpdateMarginRatio(...);
    }

    // set system router
    function setSystemRouter(address _router, bool allowed) external onlyOwner {
        systemRouters[_router] = allowed;
        emit SetSystemRouter(_router, allowed);
    }

    // set price feed address
    function setPriceFeed(address _priceFeed) external onlyOwner {
        futurePriceFeed = _priceFeed;
    }

    // set util contract address
    function setFutureUtil(address _futureUtil) external onlyOwner {
        futureUtil = _futureUtil;
    }

    // set address to receive protocol revenue
    function setProtocolFeeTo(address _feeto) external onlyOwner {
        protocolFeeTo = _feeto;
    }

    // withdraw funds from insurance pool
    function decreaseInsuranceFund(
        address _collateralToken,
        uint256 _amount,
        address _receiver
    ) public onlyOwner nonReentrant {
```

```
        require(collateralInsuranceFunds[_collateralToken] > _amount, "
            insuff_insurance_fund");
        emit UpdateInsuranceFund(...);
        collateralInsuranceFunds[_collateralToken] =
            collateralInsuranceFunds[_collateralToken] -
            _amount;
        _transferOut(_collateralToken, _amount, _receiver);
    }
```

Listing 3.11: Example Privileged Operations in `Future`

We understand the need of the privileged functions for proper operations, but at the same time the extra power to the `owner` may also be a counter-party risk to the `Perpetual` 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`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `OT Perpetual` protocol , which provides up to 70x leverage for traders to trade perpetual futures on chain. It introduces the trading fee and the funding fee to balance the longs and shorts for each trading pair, reducing the risk of draining insurance pool. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.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.