MythX

REPORT 6448A5976662E0001AAC4FAE

Created	Wed Apr 26 2023 04:16:23 GMT+0000 (Coordinated Universal Time)
Number of analyses	1
User	644779608288ab6760a063fb

REPORT SUMMARY

Analyses ID	Main source file	Detected vulnerabilities
37ef7221-2412-473b-8951-6132eeb529a6	core/OrderBook.sol	10

Analysis 37ef7221-2412-473b-8951-6132eeb529a6

MythX

Started	Wed Apr 26 2023 04:16:32 GMT+0000 (Coordinated Universal Time)
Finished	Wed Apr 26 2023 05:04:15 GMT+0000 (Coordinated Universal Time)
Mode	Deep
Client Tool	Mythx-Cli-0.7.3
Main Source File	Core/OrderBook.Sol

DETECTED VULNERABILITIES

HIGH	MEDIUM	LOW
0	0	10

ISSUES

LOW SWC-103

A floating pragma is set.

The current pragma Solidity directive is ""^0.8.0"". It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Source file

core/OrderBook.sol

Locations

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

import "../libraries/math/SafeMath.sol";
```

LOW SWC-107

Read of persistent state following external call

The contract account state is accessed after an external call to a fixed address. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

core/OrderBook.sol

Locations

```
_transferInETH();

require(msg.value > minExecutionFee, "OrderBook: insufficient execution fee");

_createDecreaseOrder(
```

LOW

SWC-107

Write to persistent state following external call

The contract account state is accessed after an external call to a fixed address. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

`core/OrderBook.sol`

Locations

```
blocknum
);
decreaseOrdersIndex[_account] = _orderIndex.add(1);
decreaseOrders[_account][_orderIndex] = order;
```

LOW

SWC-107

Write to persistent state following external call

The contract account state is accessed after an external call to a fixed address. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

`libraries/utils/ReentrancyGuard.sol`

Locations

```
// By storing the original value once again, a refund is triggered (see
// https://eips.ethereum.org/EIPS/eip-2200)
_status = _NOT_ENTERED;
}
}
```

LOW

SWC-107

Read of persistent state following external call

The contract account state is accessed after an external call to a fixed address. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

`core/OrderBook.sol`

Locations

```
bool _triggerAboveThreshold
) private {
uint256 _orderIndex = decreaseOrdersIndex[_account];

uint256 blocknum = Chain.currentBlockNumber();
```

LOW

SWC-107

Write to persistent state following external call

The contract account state is accessed after an external call to a fixed address. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

`core/OrderBook.sol`

Locations

```
);
decreaseOrdersIndex[_account] = _orderIndex.add(1);
decreaseOrders[_account][_orderIndex] = order;

emit CreateDecreaseOrder(
```

LOW

SWC-113

Multiple calls are executed in the same transaction.

This call is executed following another call within the same transaction. It is possible that the call never gets executed if a prior call fails permanently. This might be caused intentionally by a malicious callee. If possible, refactor the code such that each transaction only executes one external call or make sure that all callees can be trusted (i.e. they're part of your own codebase).

Source file

`libraries/chain/Chain.sol`

Locations

```
function currentBlockNumber() internal view returns (uint256) {
if (block.chainid == ARBITRUM_CHAIN_ID || block.chainid == ARBITRUM_GOERLI_CHAIN_ID) {
return arbSys.arbBlockNumber();
}
```

LOW

SWC-120

Potential use of "blockhash" as source of randonmness.

The environment variable "blockhash" looks like it might be used as a source of randomness. Note that the values of variables like coinbase, gaslimit, block number and timestamp are predictable and can be manipulated by a malicious miner. Also keep in mind that attackers know hashes of earlier blocks. Don't use any of those environment variables as sources of randomness and be aware that use of these variables introduces a certain level of trust into miners.

Source file

`libraries/chain/Chain.sol`

Locations

```
}

return blockhash(blockNumber);
}
}
```

LOW

SWC-120

Potential use of "block.number" as source of randonmness.

The environment variable "block.number" looks like it might be used as a source of randomness. Note that the values of variables like coinbase, gaslimit, block number and timestamp are predictable and can be manipulated by a malicious miner. Also keep in mind that attackers know hashes of earlier blocks. Don't use any of those environment variables as sources of randomness and be aware that use of these variables introduces a certain level of trust into miners.

Source file

`libraries/chain/Chain.sol`

Locations

```
}

return block.number;
}
```

LOW

SWC-123

Requirement violation.

A requirement was violated in a nested call and the call was reverted as a result. Make sure valid inputs are provided to the nested call (for instance, via passed arguments).

Source file

core/OrderBook.sol

Locations

```
) public view returns (uint256, bool) {
uint256 currentPrice = _maximizePrice
? IOracle(_oracle).getMaxPrice(_indexToken) : IOracle(_oracle).getMinPrice(_indexToken);
bool isPriceValid = _triggerAboveThreshold ? currentPrice > _triggerPrice : currentPrice < _triggerPrice;
if (_raise) {
```

Source file

core/OrderBook.sol

Locations

```
import "../oracle/interfaces/IOracle.sol";

contract OrderBook is ReentrancyGuard, IOrderBook {
using SafeMath for uint256;
using SafeERC20 for IERC20;
using Address for address payable;

uint256 public constant BASIS_POINTS_DIVISOR = 10000;
uint256 public constant PRICE_PRECISION = 1e30;
uint256 public constant USDG_PRECISION = 1e18;

struct IncreaseOrder {
address account;
address purchaseToken;
uint256 purchaseTokenAmount;
address collateralToken;
address indexToken;
uint256 sizeDelta;
bool isLong;
uint256 triggerPrice;
bool triggerAboveThreshold;
uint256 executionFee;
uint256 updatedAtBlock;
}
struct DecreaseOrder {
address account;
address collateralToken;
uint256 collateralDelta;
address indexToken;
uint256 sizeDelta;
bool isLong;
uint256 triggerPrice;
bool triggerAboveThreshold;
uint256 executionFee;
uint256 updatedAtBlock;
}
struct SwapOrder {
address account;
address[] path;
uint256 amountIn;
uint256 minOut;
uint256 triggerRatio;
bool triggerAboveThreshold;
bool shouldUnwrap;
uint256 executionFee;
```

```
uint256 updatedAtBlock;
}

mapping (address => bool) public isPositionManager;

mapping (address => mapping(uint256 => IncreaseOrder)) public increaseOrders;
mapping (address => uint256) public increaseOrdersIndex;
mapping (address => mapping(uint256 => DecreaseOrder)) public decreaseOrders;
mapping (address => uint256) public decreaseOrdersIndex;
mapping (address => mapping(uint256 => SwapOrder)) public swapOrders;
mapping (address => uint256) public swapOrdersIndex;

address public gov;
address public weth;
address public usdg;
address public router;
address public vault;
uint256 public minExecutionFee;
uint256 public minPurchaseTokenAmountUsd;
bool public isInitialized = false;

event CreateIncreaseOrder(
address indexed account,
uint256 orderIndex,
address purchaseToken,
uint256 purchaseTokenAmount,
address collateralToken,
address indexToken,
uint256 sizeDelta,
bool isLong,
uint256 triggerPrice,
bool triggerAboveThreshold,
uint256 executionFee
);
event CancelIncreaseOrder(
address indexed account,
uint256 orderIndex,
address purchaseToken,
uint256 purchaseTokenAmount,
address collateralToken,
address indexToken,
uint256 sizeDelta,
bool isLong,
uint256 triggerPrice,
bool triggerAboveThreshold,
uint256 executionFee
);
event ExecuteIncreaseOrder(
address indexed account,
uint256 orderIndex,
address purchaseToken,
uint256 purchaseTokenAmount,
address collateralToken,
address indexToken,
uint256 sizeDelta,
bool isLong,
uint256 triggerPrice,
bool triggerAboveThreshold,
uint256 executionFee,
uint256 executionPrice
);
event UpdateIncreaseOrder(
address indexed account,
uint256 orderIndex,
address collateralToken,
address indexToken,
bool isLong,
uint256 sizeDelta,
uint256 triggerPrice,
bool triggerAboveThreshold
);
event CreateDecreaseOrder(
address indexed account,
uint256 orderIndex,
address collateralToken,
uint256 collateralDelta,
address indexToken,
uint256 sizeDelta,
bool isLong,
uint256 triggerPrice,
bool triggerAboveThreshold,
uint256 executionFee
);
event CancelDecreaseOrder(
address indexed account,
uint256 orderIndex,
address collateralToken,
uint256 collateralDelta,
address indexToken,
uint256 sizeDelta,
bool isLong,
uint256 triggerPrice,
bool triggerAboveThreshold,
uint256 executionFee
);
event ExecuteDecreaseOrder(
address indexed account,
uint256 orderIndex,
address collateralToken,
uint256 collateralDelta,
address indexToken,
uint256 sizeDelta,
bool isLong,
uint256 triggerPrice,
bool triggerAboveThreshold,
uint256 executionFee,
uint256 executionPrice
);
event UpdateDecreaseOrder(
address indexed account,
uint256 orderIndex,
address collateralToken,
uint256 collateralDelta,
address indexToken,
uint256 sizeDelta,
bool isLong,
uint256 triggerPrice,
bool triggerAboveThreshold
);
event CreateSwapOrder(
address indexed account,
uint256 orderIndex,
address[] path,
uint256 amountIn,
uint256 minOut,
uint256 triggerRatio,
bool triggerAboveThreshold,
bool shouldUnwrap,
uint256 executionFee
);
event CancelSwapOrder(
address indexed account,
uint256 orderIndex,
address[] path,
uint256 amountIn,
uint256 minOut,
uint256 triggerRatio,
bool triggerAboveThreshold,
bool shouldUnwrap,
uint256 executionFee
);
event UpdateSwapOrder(
address indexed account,
uint256 ordexIndex,
address[] path,
uint256 amountIn,
uint256 minOut,
uint256 triggerRatio,
bool triggerAboveThreshold,
bool shouldUnwrap,
uint256 executionFee
);
event ExecuteSwapOrder(
address indexed account,
uint256 orderIndex,
address[] path,
uint256 amountIn,
uint256 minOut,
uint256 amountOut,
uint256 triggerRatio,
bool triggerAboveThreshold,
bool shouldUnwrap,
uint256 executionFee
);

event Initialize(
address router,
address vault,
address weth,
address usdg,
uint256 minExecutionFee,
uint256 minPurchaseTokenAmountUsd
);
event UpdateMinExecutionFee(uint256 minExecutionFee);
event UpdateMinPurchaseTokenAmountUsd(uint256 minPurchaseTokenAmountUsd);
event UpdateGov(address gov);
event SetPositionManager(address indexed account, bool isActive);

modifier onlyGov() {
require(msg.sender == gov, "OrderBook: forbidden");
_;
}

modifier onlyPositionManager() {
require(isPositionManager[msg.sender], "OrderBook: forbidden");
_;
}

constructor() {
```

```
gov = msg.sender;
}

function initialize(
address _router,
address _vault,
address _weth,
address _usdg,
uint256 _minExecutionFee,
uint256 _minPurchaseTokenAmountUsd
) external onlyGov {
require(!isInitialized, "OrderBook: already initialized");
isInitialized = true;

router = _router;
vault = _vault;
weth = _weth;
usdg = _usdg;
minExecutionFee = _minExecutionFee;
minPurchaseTokenAmountUsd = _minPurchaseTokenAmountUsd;

emit Initialize(_router, _vault, _weth, _usdg, _minExecutionFee, _minPurchaseTokenAmountUsd);
}

receive() external payable {
require(msg.sender == weth, "OrderBook: invalid sender");
}

function setPositionManager(address _account, bool _isActive) external onlyGov {
isPositionManager[_account] = _isActive;
emit SetPositionManager(_account, _isActive);
}

function setMinExecutionFee(uint256 _minExecutionFee) external onlyGov {
minExecutionFee = _minExecutionFee;

emit UpdateMinExecutionFee(_minExecutionFee);
}

function setMinPurchaseTokenAmountUsd(uint256 _minPurchaseTokenAmountUsd) external onlyGov {
minPurchaseTokenAmountUsd = _minPurchaseTokenAmountUsd;

emit UpdateMinPurchaseTokenAmountUsd(_minPurchaseTokenAmountUsd);
}

function setGov(address _gov) external onlyGov {
gov = _gov;

emit UpdateGov(_gov);
}

function getSwapOrder(address _account, uint256 _orderIndex) override public view returns (
address path0,
address path1,
address path2,
uint256 amountIn,
uint256 minOut,
uint256 triggerRatio,
bool triggerAboveThreshold,
bool shouldUnwrap,
uint256 executionFee
) {
SwapOrder memory order = swapOrders[_account][_orderIndex];
```

```
return (
order.path.length > 0 ? order.path[0] : address(0),
order.path.length > 1 ? order.path[1] : address(0),
order.path.length > 2 ? order.path[2] : address(0),
order.amountIn,
order.minOut,
order.triggerRatio,
order.triggerAboveThreshold,
order.shouldUnwrap,
order.executionFee
);
}

function createSwapOrder(
address[] memory _path,
uint256 _amountIn,
uint256 _minOut,
uint256 _triggerRatio, // tokenB / tokenA
bool _triggerAboveThreshold,
uint256 _executionFee,
bool _shouldWrap,
bool _shouldUnwrap
) external payable nonReentrant {
require(_path.length == 2 || _path.length == 3, "OrderBook: invalid _path.length");
require(_path[0] != _path[_path.length - 1], "OrderBook: invalid _path");
require(_amountIn > 0, "OrderBook: invalid _amountIn");
require(_executionFee >= minExecutionFee, "OrderBook: insufficient execution fee");

// always need this call because of mandatory executionFee user has to transfer in ETH
_transferInETH();

if (_shouldWrap) {
require(_path[0] == weth, "OrderBook: only weth could be wrapped");
require(msg.value == _executionFee.add(_amountIn), "OrderBook: incorrect value transferred");
} else {
require(msg.value == _executionFee, "OrderBook: incorrect execution fee transferred");
IRouter(router).pluginTransfer(_path[0], msg.sender, address(this), _amountIn);
}

_createSwapOrder(msg.sender, _path, _amountIn, _minOut, _triggerRatio, _triggerAboveThreshold, _shouldUnwrap, _executionFee);
}

function _createSwapOrder(
address _account,
address[] memory _path,
uint256 _amountIn,
uint256 _minOut,
uint256 _triggerRatio,
bool _triggerAboveThreshold,
bool _shouldUnwrap,
uint256 _executionFee
) private {
uint256 _orderIndex = swapOrdersIndex[_account];

uint256 blocknum = Chain.currentBlockNumber();
SwapOrder memory order = SwapOrder(
_account,
_path,
_amountIn,
_minOut,
_triggerRatio,
_triggerAboveThreshold,
_shouldUnwrap,
_executionFee,
blocknum
);
swapOrdersIndex[_account] = _orderIndex.add(1);
swapOrders[_account][_orderIndex] = order;

emit CreateSwapOrder(
_account,
_orderIndex,
_path,
_amountIn,
_minOut,
_triggerRatio,
_triggerAboveThreshold,
_shouldUnwrap,
_executionFee
);
}

function cancelMultiple(
uint256[] memory _swapOrderIndexes,
uint256[] memory _increaseOrderIndexes,
uint256[] memory _decreaseOrderIndexes
) external {
for (uint256 i = 0; i < _swapOrderIndexes.length; i++) {
cancelSwapOrder(_swapOrderIndexes[i]);
}
for (uint256 i = 0; i < _increaseOrderIndexes.length; i++) {
cancelIncreaseOrder(_increaseOrderIndexes[i]);
}
for (uint256 i = 0; i < _decreaseOrderIndexes.length; i++) {
cancelDecreaseOrder(_decreaseOrderIndexes[i]);
}
}

function cancelSwapOrder(uint256 _orderIndex) public nonReentrant {
SwapOrder memory order = swapOrders[msg.sender][_orderIndex];
require(order.account != address(0), "OrderBook: non-existent order");

delete swapOrders[msg.sender][_orderIndex];

if (order.path[0] == weth) {
_transferOutETH(order.executionFee.add(order.amountIn), payable(msg.sender));
} else {
IERC20(order.path[0]).safeTransfer(msg.sender, order.amountIn);
_transferOutETH(order.executionFee, payable(msg.sender));
}

emit CancelSwapOrder(
msg.sender,
_orderIndex,
order.path,
order.amountIn,
order.minOut,
order.triggerRatio,
order.triggerAboveThreshold,
order.shouldUnwrap,
order.executionFee
);
}

function validateSwapOrderPriceWithTriggerAboveThreshold(
address[] memory _path,
uint256 _triggerRatio,
address _oracle
) public view returns (bool) {
require(_path.length == 2 || _path.length == 3, "OrderBook: invalid _path.length");

// limit orders don't need this validation because minOut is enough
// so this validation handles scenarios for stop orders only
// when a user wants to swap when a price of tokenB increases relative to tokenA
address tokenA = _path[0];
address tokenB = _path[_path.length - 1];
require(tokenA != usdg, "tokenA is usdg, not permit");
require(tokenB != usdg, "tokenB is usdg, not permit");

uint256 tokenAPrice;
uint256 tokenBPrice;

tokenAPrice = IOracle(_oracle).getMinPrice(tokenA);
tokenBPrice = IOracle(_oracle).getMaxPrice(tokenB);

uint256 currentRatio = tokenBPrice.mul(PRICE_PRECISION).div(tokenAPrice);

bool isValid = currentRatio > _triggerRatio;
return isValid;
}

function updateSwapOrder(uint256 _orderIndex, uint256 _minOut, uint256 _triggerRatio, bool _triggerAboveThreshold) external nonReentrant {
SwapOrder storage order = swapOrders[msg.sender][_orderIndex];
require(order.account != address(0), "OrderBook: non-existent order");

order.minOut = _minOut;
order.triggerRatio = _triggerRatio;
order.triggerAboveThreshold = _triggerAboveThreshold;
order.updatedAtBlock = Chain.currentBlockNumber();

emit UpdateSwapOrder(
msg.sender,
_orderIndex,
order.path,
order.amountIn,
_minOut,
_triggerRatio,
_triggerAboveThreshold,
order.shouldUnwrap,
order.executionFee
);
}

function executeSwapOrder(address _account, uint256 _orderIndex, address payable _feeReceiver, address _oracle) override external nonReentrant onlyPositionManager {
SwapOrder memory order = swapOrders[_account][_orderIndex];
require(order.account != address(0), "OrderBook: non-existent order");
{
for (uint i = 0; i < order.path.length; i++) {
if (order.path[i] == usdg) {
continue;
}
if (order.updatedAtBlock > IOracle(_oracle).minOracleBlockNumbers(order.path[i])) {
revert("order.updatedAtBlock > oracle.minOracleBlockNumbers");
}
}
}

if (order.triggerAboveThreshold) {
// gas optimisation
// order.minAmount should prevent wrong price execution in case of simple limit order
require(
validateSwapOrderPriceWithTriggerAboveThreshold(order.path, order.triggerRatio, _oracle),
"OrderBook: invalid price for execution"
);
}

delete swapOrders[_account][_orderIndex];

IERC20(order.path[0]).safeTransfer(vault, order.amountIn);

uint256 _amountOut;
if (order.path[order.path.length - 1] == weth && order.shouldUnwrap) {
_amountOut = _swap(order.path, order.minOut, address(this));
_transferOutETH(_amountOut, payable(order.account));
} else {
_amountOut = _swap(order.path, order.minOut, order.account);
}

// pay executor
_transferOutETH(order.executionFee, _feeReceiver);

emit ExecuteSwapOrder(
_account,
_orderIndex,
order.path,
order.amountIn,
order.minOut,
_amountOut,
order.triggerRatio,
order.triggerAboveThreshold,
order.shouldUnwrap,
order.executionFee
);
}

function validatePositionOrderPrice(
bool _triggerAboveThreshold,
uint256 _triggerPrice,
address _indexToken,
bool _maximizePrice,
bool _raise,
address _oracle
) public view returns (uint256, bool) {
uint256 currentPrice = _maximizePrice
? IOracle(_oracle).getMaxPrice(_indexToken) : IOracle(_oracle).getMinPrice(_indexToken);
bool isPriceValid = _triggerAboveThreshold ? currentPrice > _triggerPrice : currentPrice < _triggerPrice;
if (_raise) {
require(isPriceValid, "OrderBook: invalid price for execution");
}
return (currentPrice, isPriceValid);
}

function getDecreaseOrder(address _account, uint256 _orderIndex) override public view returns (
address collateralToken,
uint256 collateralDelta,
address indexToken,
uint256 sizeDelta,
bool isLong,
uint256 triggerPrice,
bool triggerAboveThreshold,
uint256 executionFee
) {
```

```
DecreaseOrder memory order = decreaseOrders[_account][_orderIndex];
return (
order.collateralToken,
order.collateralDelta,
order.indexToken,
order.sizeDelta,
order.isLong,
order.triggerPrice,
order.triggerAboveThreshold,
order.executionFee
);
}

function getIncreaseOrder(address _account, uint256 _orderIndex) override public view returns (
address purchaseToken,
uint256 purchaseTokenAmount,
address collateralToken,
address indexToken,
uint256 sizeDelta,
bool isLong,
uint256 triggerPrice,
bool triggerAboveThreshold,
uint256 executionFee
) {
IncreaseOrder memory order = increaseOrders[_account][_orderIndex];
return (
order.purchaseToken,
order.purchaseTokenAmount,
order.collateralToken,
order.indexToken,
order.sizeDelta,
order.isLong,
order.triggerPrice,
order.triggerAboveThreshold,
order.executionFee
);
}

function createIncreaseOrder(
address[] memory _path,
uint256 _amountIn,
address _indexToken,
uint256 _minOut,
uint256 _sizeDelta,
address _collateralToken,
bool _isLong,
uint256 _triggerPrice,
bool _triggerAboveThreshold,
uint256 _executionFee,
bool _shouldWrap
) external payable nonReentrant {
// always need this call because of mandatory executionFee user has to transfer in ETH
_transferInETH();

require(_executionFee >= minExecutionFee, "OrderBook: insufficient execution fee");
if (_shouldWrap) {
require(_path[0] == weth, "OrderBook: only weth could be wrapped");
require(msg.value == _executionFee.add(_amountIn), "OrderBook: incorrect value transferred");
} else {
require(msg.value == _executionFee, "OrderBook: incorrect execution fee transferred");
IRouter(router).pluginTransfer(_path[0], msg.sender, address(this), _amountIn);
}

```

```
address _purchaseToken = _path[_path.length - 1];
uint256 _purchaseTokenAmount;
if (_path.length > 1) {
require(_path[0] != _purchaseToken, "OrderBook: invalid _path");
IERC20(_path[0]).safeTransfer(vault, _amountIn);
_purchaseTokenAmount = _swap(_path, _minOut, address(this));
} else {
_purchaseTokenAmount = _amountIn;
}

{
uint256 _purchaseTokenAmountUsd = IVault(vault).tokenToUsdMin(_purchaseToken, _purchaseTokenAmount);
require(_purchaseTokenAmountUsd >= minPurchaseTokenAmountUsd, "OrderBook: insufficient collateral");
}

_createIncreaseOrder(
msg.sender,
_purchaseToken,
_purchaseTokenAmount,
_collateralToken,
_indexToken,
_sizeDelta,
_isLong,
_triggerPrice,
_triggerAboveThreshold,
_executionFee
);
}

function _createIncreaseOrder(
address _account,
address _purchaseToken,
uint256 _purchaseTokenAmount,
address _collateralToken,
address _indexToken,
uint256 _sizeDelta,
bool _isLong,
uint256 _triggerPrice,
bool _triggerAboveThreshold,
uint256 _executionFee
) private {
uint256 _orderIndex = increaseOrdersIndex[msg.sender];

uint256 blocknum = Chain.currentBlockNumber();
IncreaseOrder memory order = IncreaseOrder(
_account,
_purchaseToken,
_purchaseTokenAmount,
_collateralToken,
_indexToken,
_sizeDelta,
_isLong,
_triggerPrice,
_triggerAboveThreshold,
_executionFee,
blocknum
);
increaseOrdersIndex[_account] = _orderIndex.add(1);
increaseOrders[_account][_orderIndex] = order;

emit CreateIncreaseOrder(
_account,
_orderIndex,
_purchaseToken,
_purchaseTokenAmount,
_collateralToken,
_indexToken,
_sizeDelta,
_isLong,
_triggerPrice,
_triggerAboveThreshold,
_executionFee
);
}

function updateIncreaseOrder(uint256 _orderIndex, uint256 _sizeDelta, uint256 _triggerPrice, bool _triggerAboveThreshold) external nonReentrant {
IncreaseOrder storage order = increaseOrders[msg.sender][_orderIndex];
require(order.account != address(0), "OrderBook: non-existent order");

order.triggerPrice = _triggerPrice;
order.triggerAboveThreshold = _triggerAboveThreshold;
order.sizeDelta = _sizeDelta;
order.updatedAtBlock = Chain.currentBlockNumber();

emit UpdateIncreaseOrder(
msg.sender,
_orderIndex,
order.collateralToken,
order.indexToken,
order.isLong,
_sizeDelta,
_triggerPrice,
_triggerAboveThreshold
);
}

function cancelIncreaseOrder(uint256 _orderIndex) public nonReentrant {
IncreaseOrder memory order = increaseOrders[msg.sender][_orderIndex];
require(order.account != address(0), "OrderBook: non-existent order");

delete increaseOrders[msg.sender][_orderIndex];

if (order.purchaseToken == weth) {
_transferOutETH(order.executionFee.add(order.purchaseTokenAmount), payable(msg.sender));
} else {
IERC20(order.purchaseToken).safeTransfer(msg.sender, order.purchaseTokenAmount);
_transferOutETH(order.executionFee, payable(msg.sender));
}

emit CancelIncreaseOrder(
order.account,
_orderIndex,
order.purchaseToken,
order.purchaseTokenAmount,
order.collateralToken,
order.indexToken,
order.sizeDelta,
order.isLong,
order.triggerPrice,
order.triggerAboveThreshold,
order.executionFee
);
}

function executeIncreaseOrder(address _address, uint256 _orderIndex, address payable _feeReceiver, address _oracle) override external nonReentrant onlyPositionManager {
IncreaseOrder memory order = increaseOrders[_address][_orderIndex];
```

```solidity
require(order.account != address(0), "OrderBook: non-existent order");

{
if (order.updatedAtBlock > IOracle(_oracle).minOracleBlockNumbers(order.purchaseToken)) {
revert("order.updatedAtBlock > oracle.minOracleBlockNumbers1");
}
if (order.updatedAtBlock > IOracle(_oracle).minOracleBlockNumbers(order.collateralToken)) {
revert("order.updatedAtBlock > oracle.minOracleBlockNumbers2");
}
if (order.updatedAtBlock > IOracle(_oracle).minOracleBlockNumbers(order.indexToken)) {
revert("order.updatedAtBlock > oracle.minOracleBlockNumbers3");
}
}

// increase long should use max price
// increase short should use min price
(uint256 currentPrice, ) = validatePositionOrderPrice(
order.triggerAboveThreshold,
order.triggerPrice,
order.indexToken,
order.isLong,
true,
_oracle
);

delete increaseOrders[_address][_orderIndex];

IERC20(order.purchaseToken).safeTransfer(vault, order.purchaseTokenAmount);

if (order.purchaseToken != order.collateralToken) {
address[] memory path = new address[](2);
path[0] = order.purchaseToken;
path[1] = order.collateralToken;

uint256 amountOut = _swap(path, 0, address(this));
IERC20(order.collateralToken).safeTransfer(vault, amountOut);
}

IVault(vault).increasePosition(order.account, order.collateralToken, order.indexToken, order.sizeDelta, order.isLong);

// pay executor
_transferOutETH(order.executionFee, _feeReceiver);
emitExecuteIncreaseOrder(order, _orderIndex, currentPrice);
}

function emitExecuteIncreaseOrder(IncreaseOrder memory order, uint256 _orderIndex, uint256 currentPrice) private {
emit ExecuteIncreaseOrder(
order.account,
_orderIndex,
order.purchaseToken,
order.purchaseTokenAmount,
order.collateralToken,
order.indexToken,
order.sizeDelta,
order.isLong,
order.triggerPrice,
order.triggerAboveThreshold,
order.executionFee,
currentPrice
);
}

function createDecreaseOrder(
address _indexToken,
uint256 _sizeDelta,
address _collateralToken,
uint256 _collateralDelta,
bool _isLong,
uint256 _triggerPrice,
bool _triggerAboveThreshold
) external payable nonReentrant {
_transferInETH();

require(msg.value > minExecutionFee, "OrderBook: insufficient execution fee");

_createDecreaseOrder(
msg.sender,
_collateralToken,
_collateralDelta,
_indexToken,
_sizeDelta,
_isLong,
_triggerPrice,
_triggerAboveThreshold
);
}

function _createDecreaseOrder(
address _account,
address _collateralToken,
uint256 _collateralDelta,
address _indexToken,
uint256 _sizeDelta,
bool _isLong,
uint256 _triggerPrice,
bool _triggerAboveThreshold
) private {
uint256 _orderIndex = decreaseOrdersIndex[_account];

uint256 blocknum = Chain.currentBlockNumber();
DecreaseOrder memory order = DecreaseOrder(
_account,
_collateralToken,
_collateralDelta,
_indexToken,
_sizeDelta,
_isLong,
_triggerPrice,
_triggerAboveThreshold,
msg.value,
blocknum
);
decreaseOrdersIndex[_account] = _orderIndex.add(1);
decreaseOrders[_account][_orderIndex] = order;

emit CreateDecreaseOrder(
_account,
_orderIndex,
_collateralToken,
_collateralDelta,
_indexToken,
_sizeDelta,
_isLong,
_triggerPrice,
_triggerAboveThreshold,
msg.value
```

```
);
}

function executeDecreaseOrder(address _address, uint256 _orderIndex, address payable _feeReceiver, address _oracle) override external nonReentrant onlyPositionManager {
DecreaseOrder memory order = decreaseOrders[_address][_orderIndex];
require(order.account != address(0), "OrderBook: non-existent order");

{
if (order.updatedAtBlock > IOracle(_oracle).minOracleBlockNumbers(order.collateralToken)) {
revert("order.updatedAtBlock > oracle.minOracleBlockNumbers2");
}
if (order.updatedAtBlock > IOracle(_oracle).minOracleBlockNumbers(order.indexToken)) {
revert("order.updatedAtBlock > oracle.minOracleBlockNumbers");
}
}

// decrease long should use min price
// decrease short should use max price
(uint256 currentPrice, ) = validatePositionOrderPrice(
order.triggerAboveThreshold,
order.triggerPrice,
order.indexToken,
!order.isLong,
true,
_oracle
);

delete decreaseOrders[_address][_orderIndex];

uint256 amountOut = IVault(vault).decreasePosition(
order.account,
order.collateralToken,
order.indexToken,
order.collateralDelta,
order.sizeDelta,
order.isLong,
address(this)
);

// transfer released collateral to user
if (order.collateralToken == weth) {
_transferOutETH(amountOut, payable(order.account));
} else {
IERC20(order.collateralToken).safeTransfer(order.account, amountOut);
}

// pay executor
_transferOutETH(order.executionFee, _feeReceiver);
emitExecuteDecreaseOrder(order, _orderIndex, currentPrice);
}

function emitExecuteDecreaseOrder(DecreaseOrder memory order, uint256 _orderIndex, uint256 currentPrice) private {
emit ExecuteDecreaseOrder(
order.account,
_orderIndex,
order.collateralToken,
order.collateralDelta,
order.indexToken,
order.sizeDelta,
order.isLong,
order.triggerPrice,
order.triggerAboveThreshold,
order.executionFee,
currentPrice
);
}

function cancelDecreaseOrder(uint256 _orderIndex) public nonReentrant {
DecreaseOrder memory order = decreaseOrders[msg.sender][_orderIndex];
require(order.account != address(0), "OrderBook: non-existent order");

delete decreaseOrders[msg.sender][_orderIndex];
_transferOutETH(order.executionFee, payable(msg.sender));

emit CancelDecreaseOrder(
order.account,
_orderIndex,
order.collateralToken,
order.collateralDelta,
order.indexToken,
order.sizeDelta,
order.isLong,
order.triggerPrice,
order.triggerAboveThreshold,
order.executionFee
);
}

function updateDecreaseOrder(
uint256 _orderIndex,
uint256 _collateralDelta,
uint256 _sizeDelta,
uint256 _triggerPrice,
bool _triggerAboveThreshold
) external nonReentrant {
DecreaseOrder storage order = decreaseOrders[msg.sender][_orderIndex];
require(order.account != address(0), "OrderBook: non-existent order");

order.triggerPrice = _triggerPrice;
order.triggerAboveThreshold = _triggerAboveThreshold;
order.sizeDelta = _sizeDelta;
order.collateralDelta = _collateralDelta;
order.updatedAtBlock = Chain.currentBlockNumber();

emit UpdateDecreaseOrder(
msg.sender,
_orderIndex,
order.collateralToken,
_collateralDelta,
order.indexToken,
_sizeDelta,
order.isLong,
_triggerPrice,
_triggerAboveThreshold
);
}

function _transferInETH() private {
if (msg.value != 0) {
IWETH(weth).deposit{value: msg.value}();
}
}

function _transferOutETH(uint256 _amountOut, address payable _receiver) private {
IWETH(weth).withdraw(_amountOut);
_receiver.sendValue(_amountOut);
```

```
}

function _swap(address[] memory _path, uint256 _minOut, address _receiver) private returns (uint256) {
if (_path.length == 2) {
return _vaultSwap(_path[0], _path[1], _minOut, _receiver);
}
if (_path.length == 3) {
uint256 midOut = _vaultSwap(_path[0], _path[1], 0, address(this));
IERC20(_path[1]).safeTransfer(vault, midOut);
return _vaultSwap(_path[1], _path[2], _minOut, _receiver);
}

revert("OrderBook: invalid _path.length");
}

function _vaultSwap(address _tokenIn, address _tokenOut, uint256 _minOut, address _receiver) private returns (uint256) {
uint256 amountOut;

if (_tokenOut == usdg) { // buyUSDG
amountOut = IVault(vault).buyUSDG(_tokenIn, _receiver);
} else if (_tokenIn == usdg) { // sellUSDG
amountOut = IVault(vault).sellUSDG(_tokenOut, _receiver);
} else { // swap
amountOut = IVault(vault).swap(_tokenIn, _tokenOut, _receiver);
}

require(amountOut >= _minOut, "OrderBook: insufficient amountOut");
return amountOut;
}
}
```