Monad Orderbook

Security Review

Cantina Managed review by:

Kurt Barry, Lead Security Researcher

Xmxanuel, Security Researcher

Sujith Somraaj, Associate Security Researcher

February 14, 2024

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Monad is a decentralized, developer-forward Layer 1 smart contract platform that ushers in a new paradigm of possibility through pipelined execution of Ethereum transactions.

From Feb 5th to Feb 13th the Cantina team conducted a review of monad-orderbook on commit hash 7901b20f. The team identified a total of **37** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 0
- Medium Risk: 5
- Low Risk: 7
- Gas Optimizations: 7
- Informational: 18

3 Findings

3.1 Medium Risk

3.1.1 If a `takerFees` exists, `Router.marketOrder` will always revert with `InsufficientBalance` for `buyOrders`

Severity: Medium Risk

Context: Router.sol#L45

Description: In Monad, the `takerFee` needs to be paid with the `quoteBalance` for `buyOrders`. Therefore, the `quoteBalance` of a user needs to have a sufficient balance to cover the fees as well not only the order itself.

The stateless router contract does not consider the required `fee` amount in the `quoteAmount` calculation. The `orderbook.limitOrder` call will always revert with `InsufficientBalance`:

```
quoteAmount = orderBook.baseToQuote(baseAmount, price);
```

Recommendation: Get the `takerFee` from the specific orderbook and calculate the required `feeAmount` and add it to the quote amount:

```
quoteAmount = orderBook.baseToQuote(baseAmount, price) + feeAmount;
```

3.1.2 `priceToIndex` can return the zero Index, allowing posting and executing orders at such index

Severity: Medium Risk

Context: PriceTree.sol#L41-L44

Description: Per discussion with the Monad team, the zero index and corresponding price are reserved to indicate an empty order book, and are not valid for orders to be placed or executed at. The `priceToIndex()` function attempts to exclude the lowest price by using <= when checking the price lower bound:

```
if (price <= minPrice || price > maxPrice) {
    revert Errors.PriceOutOfRange();
}
```

However, it calculates the index using truncating division that will map any price above `minPrice` but less than `minPrice + minPriceIncrement` to the zero index:

```
index = uint24((price - minPrice) / minPriceIncrement);
```

As a result, users can post orders at the price index of zero using price values strictly between `minPrice` and `minPrice + minPriceIncrement`, resulting in unintended behavior.

Proof of concept: Create an order book with `minPrice` as `1e6` and `minPriceIncrement` as `0.1e6`. Following the creation of the order book, try placing an order with any price between the `1e6` and `1.1e6`, which allows the creation of new orders at `1e6` price levels (zero index).

```
function test_ordersIntoZeroIndex() public {
    vm.startPrank(user);
    quote.approve(orderBook, quote.totalSupply());
    OrderBook(orderBook).depositQuote(quote.totalSupply());

    base.approve(orderBook, base.totalSupply());
    OrderBook(orderBook).depositBase(base.totalSupply());

    OrderBook(orderBook).limitOrder(user, 1.09e6, 1, 2);
    OrderBook(orderBook).limitOrder(user, 1.09e6, 1, 1);

    vm.stopPrank();
}
```

After running the scenario above, the following event, `OrderFilled`, indicates the order being executed at the 0th price index.

```
        emit OrderFilled(priceIndex: 0, amount: 10000000000000 [1e13], makerOrderKey:
    ↪  0x0000000000000000000000000000000000000000000000000000000000000001, takerOrderKey:
    ↪  0x0000000000000001000000000000000000000000000000000000000000000002, isBuy: true)
          0x0000000000000001000000000000000000000000000000000000000000000002
```

Recommendation: Modify the lower bound validation check to `price < minPrice + minPriceIncrement` or exclude the zero index explicitly post-calculation.

```
- if (price <= minPrice || price > maxPrice) {
+ if (price < minPrice + minPriceIncrement || price > maxPrice) {
```

3.1.3 `maxFeeNeeded` calculation for `buyOrders` can result in excessively high values for `IOC` and `PO` order types

Severity: Medium Risk

Context: OrderBook.sol#L181

Description: The monad order book has three different types of orders:

- Good 'till canceled (`GTC`).
- Immediate or cancel (`IOC`).
- Post only (`PO`).

For each of these orders. `fees` can occur, which are paid in the quote balance for `buy` orders. Therefore, if a new order is placed, additional `quote` tokens are required in the user's balance. Sell orders will receive fewer `quote` tokens after the match, so it is not relevant.

For `GTC` orders, if the order can be immediately matched, the `takerFee` will be applied, if added to the order book the `makerFee`.

It cannot be known in advance which fee will be applied before the matching. Therefore, the `maxFeeNeeded` considers the worst-case scenario and takes the `Utils.max(takerFee, makerFee)` in the `maxFeeNeeded` calculation:

```
uint256 maxFeeNeeded = Utils.divide(orderQuantity * Utils.max(takerFee, makerFee), feePrecision, true);
```

This is the correct calculation for `GTC` orders. However, the same calculation is applied for all three order types. `IOC` orders will always pay `takerFee`, and `PO` orders `makerFee` (assuming the fee is positive).

This requires the user to have a higher balance in the account than necessary. If the fee that should not be applied is higher than the actual fee, it can result in unnecessary costs.

Recommendation: The correct fee calculations are the following:

- `IOC` orders:

  ```
  uint256 maxFeeNeeded = Utils.divide(orderQuantity * takerFee, feePrecision, true);
  ```

- `PO` orders (if the `makerFee` is positive):

  ```
  uint256 maxFeeNeeded = Utils.divide(orderQuantity * makerFee, feePrecision, true);
  ```

3.1.4 Exploitable router zero balance check in `Router.marketOrder`

Severity: Medium Risk

Context: Router.sol#L34

Description: The `Router.marketOrder` checks if the router has no balance in the Orderbook. If a balance exists, the router would revert because each call should withdraw all balances in the end.

However, an attacker could call the orderbook directly with a `limitOrder` and enter the router address as `owner` (see Orderbook.sol#L140), which would result in a non-zero balance in the router after the order has been matched. Hence, all calls to the `Router.marketOrder` for the specific `orderBook` would revert from this point in time and a redeployment of the router contract would be required.

Recommendation: Consider removing the check. In the current design, a zero balance for the router can not be guaranteed and is not a blocker for the `marketOrder` call.

3.1.5 `Router.marketOrder` checks `msg.sender` instead of `address(this)` for the `quoteBalance`

Severity: Medium Risk

Context: Router.sol#L34

Description: The `Router` is a stateless contract that allows interactions with different `Orderbooks`. Furthermore, it only allows the `IOC` order type, which means an order can be fulfilled immediately or it should be canceled. After the `limitOrder`, all balances are withdrawn and returned to the `msg.sender`. Therefore, the `Router` should never have a balance in an orderbook for a new call.

The following check should ensure this is always the case.

```
require(orderBook.getQuoteBalance(msg.sender) == 0, "Router: quote balance not zero");
require(orderBook.getBaseBalance(msg.sender) == 0, "Router: base balance not zero");
```

However, to check the balance of the `Router` contract in the orderbook, `address(this)` should be passed instead of `msg.sender`.

Recommendation: Consider removing the check because it can be exploited. Users could place orders directly in the order book and enter the router's address as beneficiary. Which would result in a router balance after the order is matched.

3.2 Low Risk

3.2.1 Use `safeApprove` in `Router.sol`

Severity: Low Risk

Context: Router.sol#L47, Router.sol#L54

Description: The `marketOrder()` function in `Router.sol` allows users to place market buy/sell orders to a monad orderbook. To place limit orders to the orderbook, the router contract should approve base/quote tokens to the orderbook contract. The contract implements `SafeERC20` for the base/quote tokens, but the function fails to use the `safeApprove` method:

```
function marketOrder(
  address orderBookAddress,
  uint256 unitQuantity,
  uint256 price,
  bool isBuy
 ) public {
    // ...
    quoteToken.approve(address(orderBook), quoteAmount);
    // ...
    baseToken.approve(address(orderBook), baseAmount);
    // ...
}
```

Recommendation: Consider using `safeApprove()` instead of `approve()` function to handle non-conformant tokens like `USDT`.

```
-  quoteToken.approve(address(orderBook), quoteAmount);
+  quoteToken.safeApprove(address(orderBook), quoteAmount);
```

3.2.2 `depositQuote` and `depositQuote` should make the ERC20 transfer before book-keeping

Severity: Low Risk

Context: OrderBook.sol#L94

Description: The concrete ERC20 implementation is unkown. Some `ERC20` tokens could have callbacks (before or after the transfer) and allow re-entrancy.

Recommendation: It would be safer for deposit functions to first take the tokens and update the book-keeping afterward.

3.2.3 `depositedAmountPrecision` calculation can result in zero, which would revert all `limitOrder` transactions

Severity: Low Risk

Context: OrderBook.sol#L72

Description: The `initialize` method calculates the precision which is used for the `depositedAmount` of an order. An entire order should fit into one storage slot, therefore only limited precision is available.

However, some orderbook configurations can result in a `depositedAmountPrecision` of zero. Users could still deposit/withdraw balances in the orderbook, but it would be not possible to place a limit order (depositedAmountPrecision of zero would result in a division by zero error).

Example:

```
uint basePrecision = 10**6;
uint quotePrecision = 10**18;
uint pricePrecision = 10**6;
uint minPriceIncrement = 0.1e6;
uint minQuantityIncrement = 1e14;
uint feePrecision = 10**6;
```

Recommendation: Require a reasonable minimum for the `depositedAmountPrecision` otherwise the `initialize` method should revert.

3.2.4 Add `sanity checks` for `Orderbook.initialize` parameter

Severity: Low Risk

Context: OrderBook.sol#L51

Description: The Monad orderbook has some constraints about key orderbook parameters, which are defined via the `initialize` method.

Recommendation: Add the following sanity checks to the `initialize` function:

- **MakerFee/TakerFee Maximum Check**:

```
if (makerFee > feePrecision || makerFee < (feePrecision * -1)) {
    return Errors.MakerFeeOutOfRange();
}
if (takerFee > feePrecision) {
    return Errors.TakerFeeOutOfRange();
}
```

- **Base and Quote Token Precision**:

```
uint private constant MAX_TOKEN_DECIMALS = 18;
uint private constant MIN_TOKEN_DECIMALS = 6;

uint8 baseTokenDecimals =  IERC20Metadata(baseToken).decimals();
uint8 quoteTokenDecimals =  IERC20Metadata(baseToken).decimals();

if (baseTokenDecimals < MIN_TOKEN_DECIMALS || baseTokenDecimals > MAX_TOKEN_DECIMALS) {
    return Errors.UnsupportedTokenPrecision();
}
if (quoteTokenDecimals < MIN_TOKEN_DECIMALS || quoteTokenDecimals > MAX_TOKEN_DECIMALS) {
    return Errors.UnsupportedTokenPrecision();
}
```

3.2.5 `indexToPrice()` does not reject the zero index as invalid

Severity: Low Risk

Context: PriceTree.sol#L48

Description: The `indexToPrice()` function only enforces an upper bound on the index passed, but not a lower bound, and hence will return the price corresponding to index 0 when the `index` argument is `0`. Per discussion with the Monad team, the zero index and corresponding price are reserved to indicate an empty order book and are not valid for orders to be placed or executed at.

Recommendation: Add a lower bound check on the `index` argument (`index > 0`).

3.2.6 `UUPSUpgradeable` admin slot with `_changeAdmin` function could be used instead of `admin` variable in `Orderbook`

Severity: Low Risk

Context: OrderBook.sol#L49

Description: The `UUPSUpgradeable` contracts have their own functions to `_changeAdmin` and `_getAdmin`, which store the admin under a specific `slot` in storage to avoid storage collisions in upgrades.

Recommendation: Instead of defining an `admin` variable in `Storage.sol`, the `initialize` function could use `_changeAdmin` and the `onlyOwner` modifier could use the `_getAdmin` function:

In initialize function:

```
_changeAdmin(_admin);
```

changing the modifier to:

```
modifier onlyOwner() {
    if (msg.sender != _getAdmin()) {
        revert Errors.NotAdmin();
    }
    _;
}
```

and adding a `view` function to the orderbook:

```
function admin() public view return(address) {
    return _getAdmin();
}
```

3.2.7 `PriceTree.indexToPrice` takes always the closed lowest price index which is a disadvantage for sellers

Severity: Low Risk

Context: PriceTree.sol#L40

Description: The Monad orderbook only supports a discrete range of prices. Defined by a `minPrice` and a `minPriceIncrement`, prices are indexed and stored in a `PriceTree` to efficiently find orders for a certain price. The `priceTree` allows to store up to `10m` different prices, each with a corresponding index.

```
function priceToIndex(uint256 price) public view returns (uint24 index) {
    if (price <= minPrice || price > maxPrice) {
        revert Errors.PriceOutOfRange();
    }
    index = uint24((price - minPrice) / minPriceIncrement);
}
```

A seller or buyer calling the `limitOrder` function can pass any `price`, which needs to be converted to an index.

This conversion process may lead to situations where a price does not exactly aligny with a multiple of `minPriceIncrement`, affecting therefore the `index`.

For example

```
minPriceIncrement = 10 USDC
minPrice = 0 USDC
```

The user could pass a price of `15 USDC`, which would result in an index of `1` (`10 USDC`). This makes sense for the `buyer`, and the actual match price would be a bit lower which would result in a cheaper buy.

However, from the seller's perspective, this is not ideal because the lower price would result in less money than expected. The seller would favor an index rounding up to `2` (`20 USDC`) in this example.

Recommendation: Consider one of the following options:

- Rounding the index in favor of buyer or seller.
- Allowing to only pass a multiple of the `minPriceIncrement` as price in the `limitOrder` function.
- Using the price `index` itself.
- Documenting the disadvantage for the seller in the current implementation.

3.3 Gas Optimization

3.3.1 `depositQuote()` and `depositBase()` can be optimized

Severity: Gas Optimization

Context: OrderBook.sol#L94, OrderBook.sol#L103

Description: `depositQuote()` and `depositBase()` allow users to deposit quote and base tokens to the orderbook. These functions accept an `uint256` parameter `amount`, which is cast to `uint128` inside the functions.

The functions would revert during the casting if the value is greater than `type(uint128).max`. However, adding a check earlier in the function would save 90% on gas costs for the users.

Recommendation: Consider adding an upper bound check to reduce the gas costs by reverting earlier without registering the user.

```
  function depositBase(uint256 amount) public {
+       if (amount == 0 || amount > type(uint128).max) {
            revert Errors.InsufficientBalance();
        }
        uint48 userId = _registerUser(msg.sender);
        tokenBalance[userId].baseBalance += Utils.toUint128(amount);
        IERC20Metadata(baseToken).safeTransferFrom(msg.sender, address(this), amount);
  }
```

By optimizing this, the gas cost for an unbounded deposit dropped from `~55691 GAS TO ~5329 GAS`.

3.3.2 Instead of recalculating `orderQuantity` in `_postOrder` it could be passed as a parameter by `limitOrder`

Severity: Gas Optimization

Context: OrderBook.sol#L388

Description: Instead of recalculating `orderQuantity` in `_postOrder` it could be passed as a parameter by `limitOrder`.

Recommendation: The `_limitOrder` function already calculated the `orderQuantity`. It could be passed as a parameter to save gas.

3.3.3 Moving `_bestOrderIdByIndex` call in `_matchOrder` after the `if` break conditions can save gas

Severity: Gas Optimization

Context: OrderBook.sol#L577

Description: The returned `orderId = _bestOrderIdByIndex(!isBuy, bestIndex);` is only needed if the `while` loop continues.

Recommendation: Move the `_bestOrderIdByIndex` after the `if` break conditions to save gas.

3.3.4 `unitOrderQuantity` in `Orderbook.limitOrder` only needs to be recalculated after a `_matchOrder` call

Severity: Gas Optimization

Context: OrderBook.sol#L211

Description: `unitQuantity` in `limitOrder` could only be calculated if `_matchOrder` has been called. Otherwise, the `unitQuantity` parameter could be re-used.

Recommendation: Move the calculation inside the `if` block and re-use the `unitQuanity` parameter.

```
    // ...
    // update unit quantity
    unitQuantity = orderQuantity / minQuantityIncrement;
}

// 3. Add unmatched quantity to book
if(unitQuantity > 0) {
    // if GTC or post-only then post order
    if (options == 1 || options == 2 || options == 5 || options == 6) {
        _postOrder(
            owner,
            ownerId,
            senderId,
            index,
            Utils.toUint64(unitQuantity),
            isBuy,
            orderKey
        );
    }
    // if IOC then do nothing as the order should not be posted
}
```

3.3.5 `limitOrder()` can be optimized

Severity: Gas Optimization

Context: OrderBook.sol#L140

Description: The `limitOrder()` function allows users to place three orders (`GTC`, `IOC`, and `PO`) to Monad's order book. The function has certain order type checks, which can be simplified to save some gas.

For, In OrderBook.sol#L214, the function checks if the order is not `IOC`. The check can be simplified as follows: options 3 and 4 are the only `IOC` orders.

```
- if (options == 1 || options == 2 || options == 5 || options == 6) {
+ if (options < 3 || options > 4)
```

Another similar check in OrderBook.sol#L157 for options validity checks if options are not 0 (or) greater than 6, however, this check can be simplified as below to make the call a bit cheaper.

```
- if (options == 0 || options > 6)
+ if (options < 1 || options > 6)
```

Recommendation: Consider Implementing the suggested changes above to save gas costs.

3.3.6 `listExists()` in `LinkedList.sol` can be optimized

Severity: Gas Optimization

Context: LinkedList.sol#L53-L57

Description: The body of `listExists()` could be replaced with a single line, reducing gas costs and code complexity:

```
  function listExists(LinkedList storage self) internal view returns (bool) {
-     if (self.list[HEAD].NEXT != HEAD) {
-         return true;
-     } else {
-         return false;
-     }
+     return self.list[HEAD].NEXT != HEAD;
  }
```

Recommendation: Consider Implementing the suggested change above.

3.3.7 `nodeExists()` in `LinkedList.sol` can be optimized

Severity: Gas Optimization

Context: LinkedList.sol#L32

Description: The function `nodeExists()` checks if a node exists on the input linked list. The function has multiple levels of branching which can be optimized to save gas costs.

```
function nodeExists(
    LinkedList storage self,
    uint40 _node
) internal view returns (bool) {
    if (self.list[_node].PREV > 0 || self.list[_node].NEXT > 0) {
        return true;
    } else {
        if (self.list[HEAD].NEXT == _node) {
            return true;
        } else {
            return false;
        }
    }
}
```

Recommendation: Consider combining the logic as shown below to save some gas (~26 GAS per call):

```
function nodeExists(
    LinkedList storage self,
    uint40 _node
) internal view returns (bool) {
    return (self.list[_node].PREV > 0 || self.list[_node].NEXT > 0 || self.list[HEAD].NEXT == _node);
}
```

3.4 Informational

3.4.1 Comments and naming of `closestBit*` functions in `BitMath.sol` are misleading

Severity: Informational

Context: BitMath.sol#L11-L17, BitMath.sol#L23-L34

Description: The comments on the `closestBitLeft()` and `closestBitRight()` functions describe them as returning the closest bit to the left or the right of the indicated search position. However, both are *inclusive* of this boundary bit. For example, if bit index zero is passed to `closestBitLeft()` and the index-zero bit is set in the search value, then the zero index is returned. Hence these functions are really returning the closest bit that is not to the right or left or the search bit, respectively, for `closestBitLeft()` and `closestBitRight()`.

Recommendation: Clarify the comments and possibly also the naming to make it clear the search is inclusive of the bit index passed in.

3.4.2 Favor `if(condition)` over `if(!condition)` for `if/else` blocks

Severity: Informational

Context: OrderBook.sol#L547

Description/Recommendation: Switching `if/else` lines in `Orderbook._matchOrders` and changing the `if(!isBuy)` to `if(isBuy)` in the extra solvency checks would be more consistent with other places within the codebase where `if(isBuy)` is used.

3.4.3 Constants in `Storage.sol` are not named with all capital letters

Severity: Informational

Context: Storage.sol#L60, Storage.sol#L16

Description: As per the Solidity style guide, constant variables should be named in all capital letters. In `Storage.sol`, they are named in mixed case, which is inconsistent.

```
    uint256 public constant pricePrecision = 1e6;
    // ...
    uint256 public constant feePrecision = 1_000_000;
```

Recommendation: Consider changing the variables to capital letters to adhere to the solidity style guide.

```
- uint256 public constant pricePrecision = 1e6;
+ uint256 public constant PRICE_PRECISION = 1e6;
  // ...
- uint256 public constant feePrecision = 1_000_000;
+ uint256 public constant FEE_PRECISION = 1e6;
```

3.4.4 Unnecessary `uint40(orderId)` casting in `Orderbook.cancelOrder`

Severity: Informational

Context: OrderBook.sol#L261

Description: The `uint40` type casting `_removeOrder(isBuy, uint40(orderId), index);` is not necessary since `orderId` is already a `uint40`.

Recommendation: Remove the type casting.

3.4.5 `tokenBalance` mapping in `Storage.sol` misses visibility specifier

Severity: Informational

Context: Storage.sol#L36

Description: The mapping `tokenBalance` maps a userId of type `uint48` in `Storage.sol`. However, this mapping lacks a visibility specifier. Since there are individual helper functions to query a user's quote and base balance, declaring the visibility of this mapping to internal would reduce the code size by eliminating the need for one more public function.

Contract size diff: | | | | |-|-|-| | Storage | 0.801 | 23.775 | | Storage | 0.895 | 23.681 |

The first row indicates the `Storage. sol` contract size when the mapping is declared internal.

Recommendation: Please make sure to specify the visibility of the mapping explicitly. If you consider the two individual balance fetch functions are sufficient, consider changing the mapping to `internal`.

3.4.6 `Storage` and `PriceTree` contracts are missing `abstract` keyword

Severity: Informational

Context: Storage.sol#L7, PriceTree.sol#L9

Description: `PriceTree.sol` and `Storage.sol` are base contracts (not deployed individually). `Storage` is inherited by `PriceTree`, which is further inherited by `OrderBook.sol`. Both contracts are intended to serve as a base contract with standard storage variables and common functionality.

```
contract PriceTree is Storage, IPriceTree {
    // ...
}

contract Storage {
    // ...
}
```

Recommendation: Since `Storage` and `PriceTree` are not meant to be standalone contracts with complete implementations for all their functions and variables, consider adding the `abstract` keyword.

```
+ abstract contract PriceTree is Storage, IPriceTree {
      // ...
  }

+ abstract contract Storage {
      // ...
  }
```

3.4.7 Code consistency: `PriceTree._bestIndex` is the only function in `PriceTree` which is not a low level function

Severity: Informational

Context: PriceTree.sol#L275

Description: The `_bestIndex` function in the `PriceTree` is the only function that indirectly knows about the `Orderbook` logic. In the case of `buy`, the `sell` orders need to be returned and vice-versa.

Recommendation: Move the function to the `Orderbook` contract or keep the call low level and handle the logic in the `Orderbook`.

3.4.8 Duplicated code in `Orderbook._postOrder` in `if/else` conditions

Severity: Informational

Context: OrderBook.sol#L421

Description: The `Orderbook._postOrder` function contains the following duplicated code in the `if` and `else`.

```
// add order to linked list
Order memory order = Order({
    owner: ownerId,
    unitOrderQuantity: unitOrderQuantity,
    depositedAmount: depositedAmount
});
_addOrder(isBuy, idCounter, index, order);
```

Recommendation: Move the code snippet outside the `if/else`.

3.4.9 Code consistency: `isBuy` is always the first parameter in PriceTree except in `_getOrder`

Severity: Informational

Context: PriceTree.sol#L108

Description: In `_getOrder`, the `isBuy` parameter is not the first parameter.

Recommendation: The first parameter should always be `isBuy`, like in all other functions within `PriceTree`.

3.4.10 `multicall()` in `OrderBook` can be further generalized

Severity: Informational

Context: OrderBook.sol#L271

Description: The `multicall()` function in `OrderBook.sol` allows the user to batch actions. Users can execute the functions `limitOrder()` and `cancelOrder()` multiple times in one transaction using this function.

However, this function executes actions sequentially and is highly limited. For example, users can only execute `x` limit orders followed by `y` cancel orders. Other scenarios, where users might want to cancel multiple orders before executing limit orders, or a mix-up case where users might want to create and cancel orders continuously without a sequential pattern, are not supported.

```
function multicall(
    bytes[] calldata openOrderData,
    bytes32[] calldata cancelOrderData
) external {
    for (uint256 i = 0; i < openOrderData.length; ++i) {
        (address owner, uint256 price, uint256 unitQuantity, uint8 options) = abi.decode(
            openOrderData[i],
            (address, uint256, uint256, uint8)
        );
        limitOrder(owner, price, unitQuantity, options);
    }
    for (uint256 i = 0; i < cancelOrderData.length; ++i) {
        cancelOrder(cancelOrderData[i]);
    }
}
```

Recommendation: Consider implementing a generalized `multicall()` functionality, where users can define the execution sequence.

3.4.11 Inherited functions miss `override` keyword in `OrderBook.sol` and `PriceTree.sol`

Severity: Informational

Context: OrderBook.sol#L94, OrderBook.sol#L103, OrderBook.sol#L112, OrderBook.sol#L121, OrderBook.sol#L140, OrderBook.sol#L234, OrderBook.sol#L271, PriceTree.sol#L40, PriceTree.sol#L47, PriceTree.sol#L54, PriceTree.sol#L59, PriceTree.sol#L67, PriceTree.sol#L76, PriceTree.sol#L85

Description: `OrderBook.sol` and `PriceTree.sol` inherits functions from its interfaces `IOrderBook.sol` and `IPriceTree.sol` respectively; however, the overriden inherited function misses the `override` keyword.

Recommendation: Consider adding the `override` keyword for overridden functions from the interface.

```
- function depositQuote(uint256 amount) public {
+ function depositQuote(uint256 amount) public override {
  // ...
- function depositBase(uint256 amount) public {
+ function depositBase(uint256 amount) public override {
  // ...
```

3.4.12 Explicitly use `@inheritdoc` wherever possible in `OrderBook.sol`

Severity: Informational

Context: OrderBook.sol#L94, OrderBook.sol#L103, OrderBook.sol#L112, OrderBook.sol#L121, OrderBook.sol#L140, OrderBook.sol#L234, OrderBook.sol#L271

Description: `OrderBook.sol` is the primary contract allowing users to trade any base-quote pairs in Monad. The contract's interface, `IOrderBook.sol` is also imported into the file. However, there is either a duplication of documentation for the abovementioned functions (or missing documentation).

Recommendation: Consider importing documentation using the `@inheritdoc` keyword to improve code readability and documentation.

```
+ @inheritdoc IOrderBook
  function depositQuote(uint256 amount) public {
      // ...

+ @inheritdoc IOrderBook
  function depositBase(uint256 amount) public {
      // ...

- /**
-  * @notice Allows a user to post a GTC, IOC or post-only order
- * @param owner trader that owns the liquidity of the order
-  * @param price price to post
-  * @param unitQuantity amount of base tokens a user would like to buy / sell
-  *                     (multiple of minQuantityIncrement)
-  * @param options - 1,3,5 is a buy GTC, IOC and post-only order respectively
-  *                - 2,4,6 is a sell GTC, IOC and post-only order respectively
-  * @return bytes32 returns the order key of the posted order or 0
-  */
+ @inheritdoc IOrderBook
  function limitOrder(
      // ....
```

3.4.13 Error message in `depositQuote()` and `depositBase()` could be made more accurate

Severity: Informational

Context: OrderBook.sol#L96, OrderBook.sol#L105

Description: `depositBase()` and `depositQuote()` are used by users to deposit base and quote tokens to the orderbook contract. Both functions take an `amount` parameter from the user, representing the number of tokens the user wants to deposit/withdraw from the order book. The contract will revert when the user enters zero as `amount`. However, the error message must be more accurate and might confuse the users.

```
// ...
function depositQuote(uint256 amount) public {
    if (amount == 0) {
         revert Errors.InsufficientBalance();
    }
// ...
}

function depositBase(uint256 amount) public {
    if (amount == 0) {
        revert Errors.InsufficientBalance();
    }
// ...
}
// ....
```

Recommendation: Consider changing the error message to represent the revert case accurately, e.g. `ZERO_INPUT_AMOUNT()`.

3.4.14 Emit event in critical state-changing functions in `OrderBook.sol`

Severity: Informational

Context: OrderBook.sol#L94, OrderBook.sol#L103, OrderBook.sol#L112, OrderBook.sol#L121

Description: `depositQuote()`, `depositBase()`, `withdrawQuote()` and `withdrawBase()` don't emit an `event` on successful deposit/withdrawal of quote and base tokens to the order book:

```
contract OrderBook is IOrderBook, PriceTree, Initializable, UUPSUpgradeable {
    // ...
    function depositQuote(uint256 amount) public {
        // ...
    }

    function depositBase(uint256 amount) public {
        // ...
    }

    function withdrawQuote(uint256 amount) public {
        // ...
    }

    function withdrawBase(uint256 amount) public {
        // ...
    }
}
```

Recommendation: Consider emitting an event in the functions mentioned above.

3.4.15 `OrderBookFactory.activeMarkets` array could be a `mapping`

Severity: Informational

Context: OrderBookFactory#L10

Description: The `orderBookFactory` keeps an array with all deployed orderbooks called `activeMarkets`.

Recommendation: `activeMarkets` should be a mapping. This would allow for the `Router` to verify in `O(1)` if an Orderbook has been deployed by the `OrderbookFactory` and would guarantee the correct implementation.

A new `marketEvent` could be emitted to allow an easy off-chain iteration of all markets.

3.4.16 Replace hardcoded values with `constants` in `Router` and `OrderBook`

Severity: Informational

Context: Router.sol#L42, Router.sol#L50, OrderBook.sol#L214, OrderBook.sol#L194

Description: Hardcoded magic numbers reduce code readability and can lead to errors.

The function `marketOrder()` in `Router.sol` executes market orders in any order book. Hence, the order options values are limited to `3` (for market buy) and `4` (for market sell), which are hardcoded in the function.

```
function marketOrder(
 address orderBookAddress,
 uint256 unitQuantity,
 uint256 price,
 bool isBuy
) public {
   // ...
   if (isBuy) {
       options = 3; // buy IOC
       // ...
    } else {
       options = 4; // sell IOC
       // ...
    }
// ...
}
```

The function `limitOrder()` in `OrderBook.sol` helps users place a new order to the order book and does some equality type checks for options (order types). These values are hardcoded and can be replaced:

```
function limitOrder(
        address owner,
        uint256 price,
        uint256 unitQuantity,
        uint8 options
    ) public returns(bytes32) {
   // ...
   if (options == 5 || options == 6) {
   // ...
    if (options == 1 || options == 2 || options == 5 || options == 6) {
   // ...
}
```

Also, there are a few other hardcoded values, including `address(0)`, `9_999_999` and `0`, which could improve code readability.

Recommendation: Consider using a constant for the value of `3` and `4`, or supply it via the `constructor`

3.4.17 Typos

Severity: Informational

Context: IOrderBook.sol#L69, IPriceTree.sol#L80, OrderBook.sol#L276

Description: There are a few typos in the code and comments:

- IOrderBook.sol#L69:

```
- @param amount- base amount a user would like to withdraw
+ @param amount base amount a user would like to withdraw
```

- IPriceTree.sol#L80:

```
- * @notice Convert quote amount to a amount amount
+ * @notice Convert quote amount to base amount
```

- OrderBook.sol#L276:

```
- (address owner, uint256 price, uint256 unitQuantity, uint8 options) = abi.decode(
+ (address owner, uint256 price, uint256 unitQuantity, uint8 option) = abi.decode(
```

Recommendation: Consider fixing the typos listed above.

3.4.18 `Orderbook._matchOrder` break condition can be simplified

Severity: Informational

Context: OrderBook.sol#L578

Description: The following break condition in the `_matchOrder` function

```
if (isBuy) {
    if (bestIndex >= index + 1) { break; }
} else {
    if (bestIndex <= index - 1) { break; }
}
```

can be simplified to a simpler logical equivalent.

Recommendation: Consider using the following logical equivalent:

```
if (isBuy) {
    if (bestIndex > index) { break; }
} else {
    if (bestIndex < index ) { break; }
}
```