Order Book Protocol Audit Report - CodeHawks

Version 1.0

Tanu Gupta

July 7, 2025

Order Book Protocol Audit Report - CodeHawks

Tanu Gupta

July 7, 2025

Prepared by: Tanu Gupta Lead Security Researcher:

- Tanu Gupta

Table of Contents

- Medium
  * [M-1] Missing interface check in `OrderBook::setAllowedSellToken` function could permit non-ERC20 tokens leading to complete DOS for affected token opeartion.
  * [M-2] Precision loss in fee calculation enables zero fee on small orders resulting in revenue loss for the protocol on small trades.
  * [M-3] Unrestricted emergency withdrawal of non-Core tokens enables owner to exploit seller funds, undermining the security for participants listing non-core tokens
- Low
  * [L-1] `OrderBook::getOrderDetailsString` will have the value of variable `tokenSymbol` empty for newly added tokens meaning for tokens other than `wETH`, `wBTC`, `wSOL` this value will be `""` causing confusion among users.
  * [L-2] `OrderBook::emergencyWithdrawERC20` allows zero-amount withdrawals, leading to unnecessary resource usage.
  * [L-3] Solidity pragma should be specific, not wide
- Gas
  * [G-1] Unoptimized order status logic increases gas costs due to expensive string operations.

Protocol Summary

The OrderBook contract is a peer-to-peer trading system designed for ERC20 tokens like wETH, wBTC, and wSOL. Sellers can list tokens at their desired price in USDC, and buyers can fill them directly on-chain.

Disclaimer

Tanu Gupta makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

The findings described in this document correspond the following repo

```
https://github.com/CodeHawks-Contests/2025-07-orderbook
```

Scope

```
| src
    * OrderBook.sol
```

Roles

Owner - For doing the admin jobs

Users - For creating or buying orders and their associated operations

Executive Summary

Found the bugs using a tool called foundry. Audit duration 2 days.

Issues found

Severity	Number of issues found
High	2
Medium	3
Low	3
Info	1
Total	9

Findings

High

[H-1] Improper handling of token decimals without normalization causes massive pricing errors between tokens meaning tokens with different decimal places are treated identically in pricing calculations.

Description: Tokens with different decimal places—like WBTC (8 decimals) and WETH (18 decimals)—are stored and treated uniformly in the contract without adjusting for their precision. Since sellers set the total price in USDC, a mismatch in decimals can result in orders where the actual value of the token is either massively underpriced or overpriced. This can lead to unintentional giveaways, unfair trades, and potential exploitation by buyers who recognize the imbalance.

Example:

- 1 WBTC = 100,000,000 (8 decimals)
- 1 WETH = 1,000,000,000,000,000,000 (18 decimals)

When both are priced at $50,000 USDC (50,000,000,000 units with 6 decimals):

- WBTC: 50,000,000,000 / 100,000,000 = 500 USDC per unit
- WETH: 50,000,000,000 / 1,000,000,000,000,000,000 = 0 USDC per unit (rounds down!)

This creates broken pricing where WETH appears free per unit while WBTC has a meaningful price.

Impact: Tokens with high decimals appear artificially cheap. The protocol fee is calculated incorrectly.

Proof of Concept:

Decimal precision bug

```solidity
function test_Decimal_Precision_Bug() public {
    console2.log("=== DEMONSTRATING DECIMAL PRECISION ===");

    //Alice creates order for 1 WBTC priced at $50,000 USDC
    uint256 priceInUSDC = 50000e6; // $50,000 USDC

    //WBTC order: 1 WBTC = 100000000 (8 decimals)
    uint256 wbtcAmount = 1e8;
    vm.startPrank(alice);
    wbtc.approve(address(book), wbtcAmount);
    uint256 wbtcOrderId = book.createSellOrder(
        address(wbtc),
        wbtcAmount,
        priceInUSDC,
        3600
    );
    vm.stopPrank();

    //WETH order: 1 WETH = 1000000000000000000 (18 decimals)
    uint256 wethAmount = 1e18;
    //Bob creates order for 1 WETH priced at $50,000 USDC
    vm.startPrank(bob);
    weth.approve(address(book),wethAmount);
    uint256 wethOrderId = book.createSellOrder(
        address(weth),
        wethAmount,
        priceInUSDC,
        3600
    );
    vm.stopPrank();

    // Get order details
    (,,,uint256 storedWbtcAmount, uint256 wbtcPrice,,) = book.orders(
       wbtcOrderId);
    (,,,uint256 storedWethAmount, uint256 wethPrice,,) = book.orders(
       wethOrderId);

    console2.log("WBTC Order - Amount:", storedWbtcAmount);
    console2.log("WBTC Order - Price:", wbtcPrice);
    console2.log("WETH Order - Amount:", storedWethAmount);
    console2.log("WETH Order - Price:", wethPrice);

    // Calculate price per token unit (this shows the bug)
    // Note: WETH calculation will result in 0 due to integer division
    uint256 wbtcPricePerUnit = wbtcPrice / storedWbtcAmount;
    uint256 wethPricePerUnit = wethPrice / storedWethAmount; // This
       will be 0!

    console2.log("WBTC Price per unit:", wbtcPricePerUnit);
    console2.log("WETH Price per unit:", wethPricePerUnit);

    // The bug: WETH price per unit becomes 0 due to integer division
    // while WBTC has a meaningful price per unit
    assertEq(wethPricePerUnit, 0);
    assertGt(wbtcPricePerUnit, 0);

    console2.log("WETH price per unit is 0, while WBTC has meaningful
        price!");
}
```

Recommended Mitigation:

1. Normalize all token amounts to a standard decimal place (e.g., 18 decimals)
2. Store token decimal information in the contract.
3. Use normalization/denormalization functions for conversions
4. Implement proper price calculation methods that account for decimals.

```
uint256 constant NORMALIZED_DECIMALS = 18;
mapping(address => uint8) public tokenDecimals; // Store token decimals

// Normalize token amount to 18 decimals for internal calculations
function normalizeTokenAmount(address token, uint256 amount) internal
    view returns (uint256) {
    uint8 decimals = tokenDecimals[token];
    if (decimals == NORMALIZED_DECIMALS) {
        return amount;
    } else if (decimals < NORMALIZED_DECIMALS) {
        return amount * (10 ** (NORMALIZED_DECIMALS - decimals));
    } else {
        return amount / (10 ** (decimals - NORMALIZED_DECIMALS));
    }
}

// Denormalize token amount back to original decimals
function denormalizeTokenAmount(address token, uint256 normalizedAmount
    ) internal view returns (uint256) {
    uint8 decimals = tokenDecimals[token];
    if (decimals == NORMALIZED_DECIMALS) {
        return normalizedAmount;
    } else if (decimals < NORMALIZED_DECIMALS) {
        return normalizedAmount / (10 ** (NORMALIZED_DECIMALS -
            decimals));
    } else {
        return normalizedAmount * (10 ** (decimals -
            NORMALIZED_DECIMALS));
    }
}
```

Modified order functions

```solidity
function createSellOrder(
    address _tokenToSell,
    uint256 _amountToSell,
    uint256 _priceInUSDC,
    uint256 _deadlineDuration
) public returns (uint256) {

    //checks
    /// ...
    // Normalize amounts for storage
    uint256 normalizedAmount = normalizeTokenAmount(_tokenToSell,
       _amountToSell);

    orders[orderId] = Order({
        id: orderId,
        seller: msg.sender,
        tokenToSell: _tokenToSell,
        amountToSell: normalizedAmount,
        priceInUSDC: _priceInUSDC,
        deadlineTimestamp: deadlineTimestamp,
        isActive: true
    });

    return orderId;
}

function buyOrder(uint256 _orderId) public {
    ///...

    uint256 protocolFee = (order.priceInUSDC * FEE) / PRECISION;
    uint256 sellerReceives = order.priceInUSDC - protocolFee;

    // Denormalize amount for actual transfer
    uint256 actualAmount = denormalizeTokenAmount(order.tokenToSell,
       order.amountToSell);

    // Transfer logic would use actualAmount
}

// Helper function to get order details in original token decimals
function getOrderDetails(uint256 _orderId) public view returns (
   string memory details
) {
    Order storage order = orders[_orderId];
    //...
    details = string(
        abi.encodePacked(
            "Order ID: ",
            order.id.toString(),
            "\n",
            "Seller: ",
            Strings.toHexString(uint160(order.seller), 20),
            "\n",
            "Selling: ",
            //returning the amount in original token decimals
            denormalizeTokenAmount(order.tokenToSell, order.
                amountToSell).toString(),
            " ",
            tokenSymbol,
            "\n",
            "Asking Price: ",
            order.priceInUSDC.toString(),
            " USDC\n",
            "Deadline Timestamp: ",
            order.deadlineTimestamp.toString(),
            "\n",
            "Status: ",
            status
        )
    );
}
```

[H-2] Sellers bear full protocol fee while buyers pay only listed price meaning unfair pricing discourages sellers to use the protocol hence threatening protocol viability and liquidity.

Description: Currently, the protocol design places the entire burden of the `protocol fee` on the seller, while the buyer is only required to pay the listed price. This creates an imbalance where sellers are effectively earning less than expected, while buyers face no additional cost.

Over time, such unfair fee distribution may discourage sellers from participating in the protocol, leading to reduced listings and declining liquidity. If left unaddressed, this could severely impact the protocol's functionality and long-term sustainability.

```
    uint256 protocolFee = (order.priceInUSDC * FEE) / PRECISION;
    uint256 sellerReceives = order.priceInUSDC - protocolFee;

    iUSDC.safeTransferFrom(msg.sender, address(this), protocolFee);
    iUSDC.safeTransferFrom(msg.sender, order.seller, sellerReceives);
    IERC20(order.tokenToSell).safeTransfer(msg.sender, order.
        amountToSell);
```

Impact: This can cause adoption barrier for sellers and can greatly impact the sustainability of the protocol.

Proof of Concept: Seller is expected to receive less than expected while the buyer pays nothing for using the platform.

POC

```
    function test_seller_at_loss_than_buyer() external {
        vm.startPrank(alice);
        wbtc.approve(address(book), 2e8);
        uint256 aliceId = book.createSellOrder(address(wbtc), 2e8, 180
            _000e6, 2 days);
        uint256 orderInUSDC = book.getOrder(aliceId).priceInUSDC;
        uint256 protocolFee = (orderInUSDC * book.FEE()) / book.
            PRECISION();
        vm.stopPrank();

        assertEq(wbtc.balanceOf(alice), 0);

        vm.startPrank(dan);
        usdc.approve(address(book), 180_000e6);
        book.buyOrder(aliceId);
        vm.stopPrank();

        assertEq(wbtc.balanceOf(dan), 2e8);
        assert(usdc.balanceOf(alice) == 174_600e6);
        assert(usdc.balanceOf(address(book)) == protocolFee);
    }
```

Recommended Mitigation:

- Option 1: Buyer pays the fee (most common)

```
uint256 protocolFee = (order.priceInUSDC * FEE) / PRECISION;
//total amount paid by buyer
uint256 totalBuyerPayment = order.priceInUSDC + protocolFee;
//buyer pays the fee
iUSDC.safeTransferFrom(msg.sender, address(this), protocolFee);
iUSDC.safeTransferFrom(msg.sender, order.seller, order.priceInUSDC);
```

- Option 2: Split fees (balanced approach)

```
uint256 protocolFee = (order.priceInUSDC * FEE) / PRECISION;
uint256 buyerFee = protocolFee / 2;
uint256 sellerFee = protocolFee - buyerFee;

uint256 totalBuyerPayment = order.priceInUSDC + buyerFee;
//seller pays half of the fee
uint256 sellerReceives = order.priceInUSDC - sellerFee;
// buyer pays half of the fee
iUSDC.safeTransferFrom(msg.sender, address(this), buyerFee);
iUSDC.safeTransferFrom(msg.sender, order.seller, sellerReceives);
```

Medium

[M-1] Missing interface check in `OrderBook::setAllowedSellToken` function could permit non-ERC20 tokens leading to complete DOS for affected token opeartion.

Description: If an owner maliciously or accidentally sets a `non-ERC20` contract address as allowed token, users may attempt to create sell orders with this **token** and the `IERC20::safeTransferFrom` call in `OrderBook::createSellOrder` will always fail because the address does not implement ERC20 interface.

Impact: This can create a permanent denial-of-service (DOS) attack where users lose gas and cannot create orders with that token.

Proof of Concept:

```
function test_Allow_Random_Address_As_Token_Address() external {
    vm.prank(owner);
    vm.expectEmit();
    //using a random non-erc20 address - address(1)
    emit OrderBook.TokenAllowed(address(1), true);
    book.setAllowedSellToken(address(1), true);
}
```

Recommended Mitigation:

```
function setAllowedSellToken(address _token, bool _isAllowed) external
   onlyOwner {
    if (_token == address(0) || _token == address(iUSDC)) revert
        InvalidToken();

    // Validate ERC20 compliance
    if (_isAllowed) {
        try IERC20(_token).totalSupply() returns (uint256) {
            // Token implements ERC20 interface
        } catch {
            revert InvalidToken(); // Not a valid ERC20 token
        }
    }

    allowedSellToken[_token] = _isAllowed;
    emit TokenAllowed(_token, _isAllowed);
}
```

[M-2] Precision loss in fee calculation enables zero fee on small orders resulting in revenue loss for the protocol on small trades.

Description: With `FEE = 3` and `PRECISION = 100`, this represents a 3% fee. However, for small order sizes (e.g., orders worth 1–33 USDC), the fee calculation can **round down to zero** due to integer truncation. As a result, these trades incur no protocol fee.

Bots can exploit this by programmatically placing and filling numerous small-value orders that fall below the effective fee threshold. Over time, this allows them to trade for free, and gain unfair economic advantage — while the protocol silently loses expected fee revenue.

Impact: No fees get collected on micro-transactions leading to revenue loss for the protocol.

Proof of Concept:

```
function test_precision_loss_in_fee_calculation() external {
    //Arrange
    vm.startPrank(alice);
    wbtc.approve(address(book), 2e8);
    //Alice has created an order for $33 USDC
    uint256 aliceId = book.createSellOrder(address(wbtc), 2e8, 0.000033
        e6, 2 days);
    vm.stopPrank();

    //Act
    uint256 orderInUSDC = book.getOrder(aliceId).priceInUSDC;
    //Due to integer truncation, protocolFee is 0
    uint256 protocolFee = (orderInUSDC * book.FEE()) / book.PRECISION()
        ;

    //Assert
    assertEq(protocolFee, 0);
}
```

Recommended Mitigation:

1. Use higher precision constants

```
FEE = 300_000;
PRECISION = 10_000_000;

protocolFee = (order.priceInUSDC * FEE) / PRECISION;
```

2. Reject orders below a minimum value

```
// Require price to be high enough to collect at least 1 unit of fee
if ((priceInUSDC * FEE) / PRECISION == 0) revert OrderTooSmall();
```

3. Enforce a mininum fee: Ensure that the protocol always charges at least a minimal non-zero fee.

```
uint256 protocolFee = (order.priceInUSDC * FEE) / PRECISION;
if (protocolFee == 0 && FEE > 0) {
    protocolFee = MINIMUM_FEE;
}
```

[M-3] Unrestricted emergency withdrawal of non-Core tokens enables owner to exploit seller funds, undermining the security for participants listing non-core tokens

Description: The `OrderBook::emergencyWithdrawERC20` function allows the contract `owner` to withdraw any token not explicitly blocked (i.e., not wETH, wBTC, wSOL, or USDC). This opens a critical attack vector: if a seller creates an order using a newly added or unrecognized token (not hardcoded in the check), the owner can call `emergencyWithdrawERC20` to maliciously drain the seller's funds from the contract.

```
if (
  _tokenAddress == address(iWETH) ||
  _tokenAddress == address(iWBTC) ||
  _tokenAddress == address(iWSOL) ||
  _tokenAddress == address(iUSDC)
) {
  revert("Cannot withdraw core order book tokens via emergency function
     ");
}
```

Impact: It directly undermines the trust model and asset security for participants listing non-core tokens.

Proof of Concept:

```
function test_EmergencyWithdrawal_Of_NonCore_Tokens() external {
        TestToken token = new TestToken();

        vm.prank(owner);
        book.setAllowedSellToken(address(token), true);

        vm.startPrank(alice);
        token.mint(alice, 2e18);
        token.approve(address(book), 2e18);
        book.createSellOrder(address(token), 2e18, 3000e6, 3600);
        vm.stopPrank();

        assertEq(token.balanceOf(address(book)), 2e18);

        vm.prank(owner);
        vm.expectEmit();
        emit OrderBook.EmergencyWithdrawal(address(token), 2e18, owner)
            ;
        book.emergencyWithdrawERC20(address(token), 2e18, owner);

        assertEq(token.balanceOf(address(book)), 0);
        assertEq(token.balanceOf(owner), 2e18);
    }
```

Recommended Mitigation:

1. Introduce a separate mapping for allowed tokens and only permit `emergency withdrawal` for tokens not used in **active** or **historical orders**.
2. Make emergency withdrawal `time-locked` or permanently disabled once any order using a given token is created, ensuring full safety for user assets.

Low

[L-1] `OrderBook::getOrderDetailsString` will have the value of variable `tokenSymbol` empty for newly added tokens meaning for tokens other than `wETH, wBTC, wSOL` this value will be "" causing confusion among users.

Description: The `getOrderDetailsString` function assigns a `tokenSymbol` by explicitly checking if the token address matches one of three hardcoded tokens: `wETH, wBTC, or wSOL`. For any newly added or unsupported token, this check fails, and `tokenSymbol` remains an empty string. As a result, the returned order details string lacks a visible token identifier.

Impact: This can lead to user confusion and poor UX when interacting with orders involving non-hardcoded tokens.

Recommended Mitigation:

1. Add a Token-to-Symbol Mapping: Introduce a mapping in the contract to store symbol strings for each supported token:

```
//Add this additonal mapping in the storage and store values in it
    while enabling the tokens
mapping(address => string) public tokenSymbols;
function setAllowedSellToken(address _token, bool _isAllowed, string
    memory _symbol) external onlyOwner {
     if (_token == address(0) || _token == address(iUSDC)) revert
         InvalidToken(); // Cannot allow null or USDC itself
     allowedSellToken[_token] = _isAllowed;
     tokenSymbols[_token] = _symbol;
     emit TokenAllowed(_token, _isAllowed);
}
```

2. Fallback to *UNKNOWN* or Token Address: If no match is found, display a fallback symbol:

```
string memory tokenSymbol;
if (order.tokenToSell == address(iWETH)) {
    tokenSymbol = "wETH";
} else if (order.tokenToSell == address(iWBTC)) {
    tokenSymbol = "wBTC";
} else if (order.tokenToSell == address(iWSOL)) {
    tokenSymbol = "wSOL";
} else {
    tokenSymbol = "UNKNOWN"; // or Strings.toHexString(order.
        tokenToSell)
}
```

[L-2] `OrderBook::emergencyWithdrawERC20` allows zero-amount withdrawals, leading to unnecessary resource usage.

Description: The `OrderBook::emergencyWithdrawERC20` function permits the contract owner to withdraw a zero amount of tokens. A simple input check to prevent zero-amount withdrawals would improve efficiency and clarity.

Impact: Allowing zero-value withdrawals results in unnecessary transaction execution, consuming gas and emitting redundant events. This can clutter logs, waste resources.

Proof of Concept:

```
function testEmergencyWithdraw() external {
    TestToken token = new TestToken();

    vm.prank(owner);
    book.setAllowedSellToken(address(token), true);

    vm.startPrank(alice);
    token.mint(alice, 2e18);
    token.approve(address(book), 2e18);
    book.createSellOrder(address(token), 2e18, 3000e6, 3600);
    vm.stopPrank();

    assertEq(token.balanceOf(address(book)), 2e18);

    vm.prank(owner);
    vm.expectEmit();
    emit OrderBook.EmergencyWithdrawal(address(token), 0, owner);
    book.emergencyWithdrawERC20(address(token), 0, owner);
}
```

Recommended Mitigation:

```
function emergencyWithdrawERC20(address _tokenAddress, uint256 _amount,
     address _to) external onlyOwner {
    //adding this check will revert the withdrawl with zero amount
    if(_amount <= 0) revert InvalidTokenAmount();
    if (
        _tokenAddress == address(iWETH) || _tokenAddress == address(
            iWBTC) || _tokenAddress == address(iWSOL)
             || _tokenAddress == address(iUSDC)
    ) {
        revert("Cannot withdraw core order book tokens via emergency
            function");
    }
    if (_to == address(0)) {
        revert InvalidAddress();
    }
    IERC20 token = IERC20(_tokenAddress);
    token.safeTransfer(_to, _amount);

    emit EmergencyWithdrawal(_tokenAddress, _amount, _to);
}
```

[L-3] Solidity pragma should be specific, not wide

Description: Consider using a specific version of Solidity in your contracts instead of a wide version. For example, instead of `pragma solidity ^0.8.0;`, use `pragma solidity 0.8.0;`

Proof of Concept:

Wide pragma

```
pragma solidity ^0.8.0;
```

Recommended Mitigation: use `pragma solidity 0.8.0;` instead.

Gas

[G-1] Unoptimized order status logic increases gas costs due to expensive string operations.

Description: The logic used to determine and display an `orders status` relies on repetitive conditional checks combined with string operations.

- Duplicate Logic: The first assignment is completely overwritten by the second
- Gas Waste: String operations are expensive, doing them twice is wasteful
- Inconsistent Status Messages: Different messages for the same state

```
string memory status = order.isActive
        ? (block.timestamp < order.deadlineTimestamp ? "Active" : "
            Expired (Active but past deadline)")
        : "Inactive (Filled/Cancelled)";
    if (order.isActive && block.timestamp >= order.deadlineTimestamp) {
        status = "Expired (Awaiting Cancellation)";
    } else if (!order.isActive) {
        status = "Inactive (Filled/Cancelled)";
    } else {
        status = "Active";
    }
```

Impact: Increase in both execution cost and code complexity.

Recommended Mitigation: Following code can be used for status computation:

```
    // SINGLE status determination
    string memory status;
    if (!order.isActive) {
        status = "Inactive (Filled/Cancelled)";
    } else if (block.timestamp >= order.deadlineTimestamp) {
        status = "Expired (Awaiting Cancellation)";
    } else {
        status = "Active";
    }
```