OrderBook Smart Contract Security Audit Report

Prepared by: Sean Sun

July 9, 2025

Confidential - For internal use only

Contents

1 Project Overview

The OrderBook contract is designed for decentralized order matching, supporting the trading of multiple ERC20 assets.

2 Disclaimer

This report only analyzes the security of the contract code itself and does not represent an endorsement of the project's business or product. The audit is time-boxed and may not uncover all vulnerabilities. The project team is advised to conduct multiple rounds of audits and thorough testing before mainnet deployment.

3 Risk Classification

The following risk levels are used in this report:

- **High**: Issues that can directly lead to loss of funds or contract compromise.
- **Medium**: Issues that may affect normal contract operation or cause partial loss.
- **Low**: Issues that do not affect fund security but impact contract robustness or usability.
- **Info**: Style, convention, or optimization suggestions.

4 Vulnerability Statistics

Risk Level	Count
High	1
Medium	4
Low	4

5 Audit Details

5.1 Scope

OrderBook.sol and its related dependencies.

5.2 Roles

- Owner: The contract owner with administrative privileges.
- User: Regular trading users.

6 Audit Summary

The contract is generally well-designed and the core logic is clear. However, several security risks and improvement suggestions were identified, especially regarding reentrancy protection, privilege management, and robustness.

7 Detailed Vulnerability Analysis

7.1 High Risk Issues

7.1.1 Reentrancy Attack

Location: `OrderBook.sol`, function `buyOrder`, lines 194–213

```
function buyOrder(uint256 _orderId) public {
    ......
    IERC20(order.tokenToSell).safeTransfer(msg.sender,
       order.amountToSell);
    ......s
    emit OrderFilled(_orderId, msg.sender, order.seller);
}
```

Listing 1: OrderBook.sol, buyOrder (lines 194–213)

Description

The function calls an external contract (`IERC20(order.tokenToSell).safeTransfer`) after changing the order state, but before all logic is complete. If `order.tokenToSell` is a malicious contract, it could reenter this function or other vulnerable functions.

Mitigation

Add the `nonReentrant` modifier (from OpenZeppelin's ReentrancyGuard) to this and similar functions to prevent reentrancy attacks.

7.2 Medium Risk Issues

7.2.1 Excessive Owner Privileges

Location: `OrderBook.sol`, functions `setAllowedSellToken`, `emergencyWithdrawERC-20`, `withdrawFees`

```
function setAllowedSellToken(address _token, bool _isAllowed)
   external onlyOwner {
        .......
    emit TokenAllowed ( _token , _isAllowed ) ;
}
function emergencyWithdrawERC20(address _tokenAddress, uint256
   _amount, address _to) external onlyOwner {
         .......
    emit EmergencyWithdrawal(_tokenAddress, _amount, _to);
}
function withdrawFees(address _to) external onlyOwner {
        .......
    emit FeesWithdrawn(_to);
}
```

Listing 2: OrderBook.sol, owner-only functions

Description

The owner can arbitrarily add or remove tradable tokens, withdraw fees, and transfer non-core assets. If the owner's private key is compromised, contract assets and rules can be maliciously changed.

Mitigation

After mainnet launch, transfer owner privileges to a multisig or DAO governance contract. Consider adding a timelock for critical operations.

7.2.2 Timestamp Dependency

Location: `OrderBook.sol`, function `buyOrder`, line 200; function `amendSellOrder`, line 150

```
// In buyOrder:
if (block.timestamp >= order.deadlineTimestamp) revert
   OrderExpired();
// In amendSellOrder:
if (block.timestamp >= order.deadlineTimestamp) revert
   OrderExpired();
```

Listing 3: OrderBook.sol, buyOrder (line 200), amendSellOrder (line 150)

Description

Order expiration relies on `block.timestamp`, which can be slightly manipulated by miners.

Mitigation

Generally not a major issue, but be aware of the risk.

7.2.3 Order ID Overflow

Location: `OrderBook.sol`, function `createSellOrder`, line 124

```
uint256 orderId = _nextOrderId++;
```

Listing 4: OrderBook.sol, createSellOrder (line 124)

Description

No check is performed before incrementing `_nextOrderId`, which could theoretically overflow.

Mitigation

Add a require statement to ensure `_nextOrderId` $< 2^{256} - 1$ before incrementing.

7.2.4 Uninitialized Local Variable

Location: `OrderBook.sol`, function `getOrderDetailsString`, lines 224–241

```
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";
}
// tokenSymbol may be empty if not matched
```

Listing 5: OrderBook.sol, getOrderDetailsString (lines 224–241)

Description

The tokenSymbol variable in getOrderDetailsString is not set by default. If the token is not wETH/wBTC/wSOL, it will be empty.

Mitigation

Add an else branch or default value.

7.3 Low Risk Issues

7.3.1 Naming Convention

Location: `OrderBook.sol`, throughout parameter names

Description

Parameter naming is inconsistent (e.g., _orderId, _tokenToSell).

Mitigation

Unify parameter naming style for better readability.

7.3.2 Variable Shadowing

Location: `OrderBook.sol`, constructor, line 85

```
constructor(address _weth, address _wbtc, address _wsol, address
    _usdc, address _owner) Ownable(_owner) {
    // ...
}
```

Listing 6: OrderBook.sol, constructor (line 85)

Description

The _owner parameter in the constructor shadows the Ownable _owner variable.

Mitigation

Avoid using the same name for different variables.

7.3.3 Solidity Version Inconsistency

Location: `OrderBook.sol`, pragma line 2

```
pragma solidity ^0.8.0;
```

Listing 7: OrderBook.sol, pragma (line 2)

Description

The contract uses `^0.8.0`, while dependencies use `^0.8.20`.

Mitigation

Unify to `^0.8.20`.

7.3.4 Expired Orders Not Auto-Cleared

Location: `OrderBook.sol`, function `cancelSellOrder`, lines 177–192

```
function cancelSellOrder(uint256 _orderId) public {
    Order storage order = orders[_orderId];

    // Validation checks
    if (order.seller == address(0)) revert OrderNotFound();
    if (order.seller != msg.sender) revert NotOrderSeller();
    if (!order.isActive) revert OrderAlreadyInactive(); //
       Already inactive (filled or cancelled)


    // Mark as inactive
    order.isActive = false;

    // Return locked tokens to the seller
    IERC20(order.tokenToSell).safeTransfer(order.seller,
       order.amountToSell);

    emit OrderCancelled(_orderId, order.seller);
}
```

Listing 8: OrderBook.sol, cancelSellOrder (lines 177–192)

Description

Expired orders are not automatically cleared and require manual cancellation by the seller.

Mitigation

Consider adding auto-clear or batch cancellation.

8 Conclusion

This audit identified several security risks and improvement suggestions. The project team is advised to address these issues before mainnet deployment and conduct multiple rounds of security testing and community audits.