Order Book Protocol Security Review

Reviewer: Muhammad Faran **Date:** July 8, 2025 **Repo:** CodeHawks-Contests/2025-06-orderbook

---

Table of Contents

Disclaimer

I made all effort to find as many vulnerabilities in the code in the given time period, but hold 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 view of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk-Classification

The severity of each finding combines **Impact** (the damage if exploited) and **Likelihood** (the chance of exploitation). The matrix below illustrates how these two axes map to a single severity label:

Likelihood ↓ / Impact →	**High**	**Medium**	**Low**
High	**High**	**High/Medium**	**Medium**
Medium	**High/Medium**	**Medium**	**Medium-Low**
Low	**Medium**	**Medium-Low**	**Low**

Audit Details

Scope:

```
├── src
│   └── OrderBook.sol
```

Roles/Actors:

- Seller: Any user who wants to sell tokens (e.g., wETH, wBTC, wSOL).
- Buyer: Any user who wants to buy tokens listed by sellers.
- Owner: The contract admin, set in the constructor, sets which tokens can be sold, withdraws protocol fees.

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.

Executive Summary

Severity	Number Of Issues Found
High	2
Medium	0
Low	4
Informational	2

[H-1] Incorrect Fee Calculation Precision Using Low-Denomination Constants

Description

The contract charges a protocol fee using a low-precision calculation:

```
// @> fee logic based on small integers
uint256 protocolFee = (order.priceInUSDC * FEE) / PRECISION;
```

Where:

```
uint256 public constant FEE = 3; // 3%
uint256 public constant PRECISION = 100;
```

Risk

Likelihood:

- Happens on all small-value orders.
- Occurs when fees are fractional and round to zero.

Impact:

- Protocol earns no fees on low-value orders.
- Poor precision can cause rounding errors too.

Proof of Concept

Create a sell order with a very small `priceInUSDC`, such as **1 USDC** (which is `1e6` in USDC's 6 decimal format). The current fee calculation uses:

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

Where:

```
FEE = 3;
PRECISION = 100;
```

This leads to:

```
protocolFee = (1e6 * 3) / 100 = 30000 (0.03 USDC) //0.03 = 0 in solidity as it doesnot support float
```

Protocol earns 0 fee, even though the user is using the system.

This issue is caused by Solidity's integer division — **fractions are truncated**.

Another Issue could be rounding errors Even when the fee isn't completely zero, **rounding errors** can occur due to Solidity's **integer division**, which truncates toward zero.

Solidity doesn't handle floating-point math. So when you calculate fees like this:

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

Any **fractional result is truncated**. For example:

```
(order.priceInUSDC = 101)
(FEE = 3)
(PRECISION = 100)

protocolFee = (101 * 3) / 100 = 3.03 → truncated to 3
```

So the protocol **loses 0.03 USDC** on this order. It's small, but over millions of orders it accumulates.

Recommended Mitigation

Switch to 6-decimal fee precision to match USDC, e.g., 3% = `3e4`, precision = `1e6`.

```
- uint256 public constant FEE = 3;
- uint256 public constant PRECISION = 100;
+ uint256 public constant FEE = 30000; // 3% fee
+ uint256 public constant PRECISION = 1e6;
```

[H-2] Emergency Withdrawal May Enable Partial Rug Pull of New Tokens

Description

The `emergencyWithdrawERC20()` function prohibits withdrawal of core tokens (wETH, wBTC, wSOL, USDC), but allows any other token to be withdrawn by the owner, including newly whitelisted sell tokens.

```
// @> check only excludes hardcoded core tokens
 if (
      _tokenAddress == address(iWETH) || _tokenAddress == address(iWBTC) ||
_tokenAddress == address(iWSOL) || _tokenAddress == address(iUSDC)
    ) {
```

```
        revert("Cannot withdraw core order book tokens via emergency function");
    }
```

Risk

Likelihood:

- Happens when admin whitelists a new token via `setAllowedSellToken`.
- Emergency withdrawal allows full asset drain.

Impact:

- Partial or full rug of newly supported tokens.
- Trust assumptions on "owner" role are violated.

Proof of Concept

The `owner` role can whitelist a new token using `setAllowedSellToken()` and allow users to create sell orders with it. Later, the owner can call `emergencyWithdrawERC20()` to **steal all user-deposited tokens** of that new type.

Whitelist a new token (e.g., `XToken`)

```
orderBook.setAllowedSellToken(address(XToken), true);
```

Users create sell orders with the new token

```
orderBook.createSellOrder(address(XToken), 1000e18, 100e6, 3600);
```

Tokens are now held by the contract.

Owner can drain all XToken via emergencyWithdraw

```
orderBook.emergencyWithdrawERC20(address(XToken), 1000e18, ownerAddress);
```

- Users lose their deposits. There is no check to prevent the owner from withdrawing **non-core, but still active** tokens with open orders.

Recommended Mitigation

Maintain a list of "protected" tokens and block their withdrawal, or forbid withdrawal of any token in `allowedSellToken == true`.

```
- if (_tokenAddress == iWETH || ...)
+ if (allowedSellToken[_tokenAddress] || _tokenAddress == address(iUSDC)) revert();
```

This fixes the problem but now there is no use of `emergencyWithdrawERC20` remaining and to increase user trust in protocol this function shall be removed completely.

[L-1] Inconsistent Order State Misleads Order Status

Description

The `Order` struct contains an `isActive` flag that indicates whether an order is fillable. This flag is toggled off during order cancellation or successful purchase. However, it remains `true` even after the deadline expires, making the order appear active in the `getOrder` function and related UI logic.

Risk

Likelihood:

- Users interacting with the UI may attempt to buy sold or canceled orders.
- Off-chain systems may misrepresent sold and canceled orders as active.

Impact:

- Users will revert on `buyOrder` due to `OrderExpired`.
- Creates inconsistent state between contract logic and off-chain representations.

Proof of Concept

Create a sell order with a **very short deadline** (e.g., 10 seconds). After waiting for it to expire, call `getOrder()`. It will misleadingly report the status as `"Active"` despite the order being expired and unfillable.

```
// 1. Assume msg.sender is a seller with enough wETH approved for this contract

orderBook.createSellOrder(
    address(wETH),       // _tokenToSell
    1e18,                // _amountToSell (1 wETH)
    100e6,               // _priceInUSDC (100 USDC)
    10                   // _deadlineDuration = 10 seconds
);
```

Wait for **10+ seconds** (until after the deadline), then call:

```
string memory details = orderBook.getOrder(1);
```

Expected output (current behavior):

```
isActive: True
```

But the order is no longer valid due to deadline expiry.

Recommended Mitigation

- Make `isActive` false when `deadlineTimestamp` is exceeded or remove the `isActive` logic totally and rely on `deadlineTimestamp` only.

[L-2] Deadline Extension via Amend Order Function

Description

The `amendSellOrder()` function allows sellers to increment the order's deadline for up to 3 days from the current block timestamp. This can be abused to indefinitely prolong an order's lifetime by continuously extending the deadline.

```
// @> order.deadlineTimestamp = block.timestamp + _newDeadlineDuration;
```

Risk

Likelihood:

- Happens when sellers intentionally keep re-extending deadline.
- Happens any time before expiry, since it’s allowed unconditionally.

Impact:

- Defeats the purpose of having expiration semantics.

Proof of Concept

Create a sell order then **repeatedly amend** it by adding up to 3 more days each time — effectively **keeping the order alive forever** and bypassing the `MAX_DEADLINE_DURATION` restriction.

```
// Create initial order with 10s deadline
orderBook.createSellOrder(
    address(wETH),     // Token
    1e18,              // Amount
    100e6,             // Price (100 USDC)
    3 days             // Deadline duration = 3 days
);
```

```
orderBook.amendSellOrder(
    1,                 // Order ID
    1e18,              // Same amount
    100e6,             // Same price
    3 days             // Extend by max allowed
);
```

Recommended Mitigation

Cap amendments so the new deadline is not further than `originalDeadline + 3 days`.

```
- uint256 newDeadlineTimestamp = block.timestamp + _newDeadlineDuration;
+ require(block.timestamp + _newDeadlineDuration <= order.deadlineTimestamp +
```

```
MAX_DEADLINE_DURATION, "Too far");
```

[L-3] Token Symbol May Be Empty in Order Details String

Description

The function `getOrderDetailsString` relies on hardcoded token address checks to determine the symbol. If a future token is allowed via `setAllowedSellToken()`, the `tokenSymbol` will remain uninitialized and empty in the UI.

```
// @> tokenSymbol unassigned for unknown tokens
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";
        }
```

Risk

Likelihood:

- Occurs when admin adds new token via `setAllowedSellToken`.
- UI presents order details for non core tokens.

Impact:

- `getOrderDetailsString` will display no token symbol.
- UI looks broken or confusing.

Proof of Concept

The function `getOrderDetailsString()` uses hardcoded token address comparisons to set the token symbol (wETH, wBTC, wSOL). If a **new token is added via `setAllowedSellToken()`**, its symbol will be blank in the string output.

```
// Assume `newToken` is a valid ERC20 token (e.g., wDOGE) with 18 decimals
orderBook.setAllowedSellToken(address(newToken), true);

// Create sell order using the new token
orderBook.createSellOrder(
    address(newToken),
    1e18,
    100e6,
    3600
);
```

```
// View order string details
string memory details = orderBook.getOrderDetailsString(1);
```

Output will contain:

```
Selling: 1000000000000000000
```

- Notice the **missing token symbol**, which should be `"wDOGE"` or something meaningful.

Recommended Mitigation

Introduce `mapping(address => string) public tokenSymbols` and update it during token registration.

```
- string memory tokenSymbol;
+ string memory tokenSymbol = tokenSymbols[order.tokenToSell];
```

[L-4] Unnecessary Use of Strings Utility Library, Increases Bytecode of the Contract

Description

The `Strings` utility is imported and used solely for the `getOrderDetailsString()` function, which is an off-chain utility. Including the entire `Strings` library increases bytecode size and gas usage during deployment, unnecessarily.

```
import {Strings} from "@openzeppelin/contracts/utils/Strings.sol";
```

Risk

Likelihood:

- Always present at compile/deployment time

Impact:

- Increased bytecode size
- Higher deployment cost

Proof of Concept

Remove `getOrderDetailsString()` and the `Strings` import — observe reduced bytecode size.

Recommended Mitigation

Consider moving the logic off-chain (via The Graph or frontend rendering), or keep it in a separate view-only helper contract.

```
- import {Strings} from "@openzeppelin/contracts/utils/Strings.sol";
```

[I-1] Internal Functions Should Be Marked External for Gas Efficiency

Description

Functions like `createSellOrder()` and `getOrder()` are marked `public` but are not used internally. In Solidity, `external` functions are slightly more gas-efficient, especially when handling calldata directly.

```
// @> createSellOrder is public
function createSellOrder(...) public returns (uint256) { ... }
```

Risk

Likelihood:

- Occurs during every transaction using those functions

Impact:

- Slightly higher gas cost
- Missed optimization opportunities

Proof of Concept

Mark `public` functions as `external` and benchmark gas usage using Foundry or Hardhat.

Recommended Mitigation

Update `public` to `external` for all externally invoked functions that aren't used internally.

```
- function createSellOrder(...) public returns (uint256)
+ function createSellOrder(...) external returns (uint256)
```

[I-2] Proxy-Based Token: USDC is a Proxy Contract

Description

USDC is deployed as a **proxy contract**. Interacting with it via a standard `IERC20` interface without considering proxy-specific behaviors (e.g., delegatecall, upgradeability) may cause compatibility or security issues.

Risk

Likelihood:

- High, especially if USDC is upgraded

Impact:

- Compatibility issues
- Unexpected behavior or security risks if assumptions about the contract structure change

Proof of Concept

Inspect USDC's address (e.g., on Etherscan) — it's implemented using the OpenZeppelin Transparent Proxy pattern.

Recommended Mitigation

Continue using `IERC20` for basic operations, but avoid assumptions about storage layout or contract behavior. Use `IERC20Metadata` or proxy-aware wrappers when accessing metadata or non-transfer functions.
