orderBook Audit Report

Sagar Rana

July 8, 2025

OrderBook Audit Report

Sagar Rana

July 8, 2025

Prepared by: Sagar Rana Lead Auditors: - Sagar Rana

Table of Contents

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

Sagar Rana 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
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

Audit Details

Scope

```
# src
#-- OrderBook.sol
```

Roles

1. Owner: Can withdraw non core tokens in case of emergency and set allowed tokens for order creations.
2. User: Can create sell orders and amend or cancel them. Other users will be able to but those orders to fill them

Issues found

Severity	No. of issues
High	1
Medium	2
Low	3
Total	6

Findings

High

[H-01] Lack of checks in `OrderBook::emergencyWithdrawERC20()` function can cause broken orders and centralisation risks

Description

- `emergencyWithdrawERC20()`function is meant for emergency situations where funds need to be withdrawn from the contract when they get stuck in the contract due to accidental transfers.
- However, a malicious contract owner may drain all the non-core tokens from the contract for their own benefits.
- Even if tokens are withdrawn for emergency purposes, the withdraw may cause lack of funds for active orders and this breaks the contract

```
function emergencyWithdrawERC20(address _tokenAddress, uint256 _amount, address _to) external onlyOwner {
    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);
}
```

Risk

Likelihood:

- Whenever `emergencyWithdrawERC20()` function is called by the owner

Impact:

- Insufficient funds for active orders
- Malicious intent of contract owner

Proof of Concept

Place the following function into `TestOrderBook.t.sol` and run with `forge test --mt test_insufficientFundsForActiveOrderAfterEmergencyWithdraw`

```
function test_insufficientFundsForActiveOrderAfterEmergencyWithdraw() public {
    // deploy a non-core token
    MockWETH newMock = new MockWETH(18);

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

    newMock.mint(alice, 10e18);

    vm.startPrank(alice);
    newMock.approve(address(book), 10e18);
    uint256 aliceId = book.createSellOrder(address(newMock), 8e18, 1e18, 3 days);
    vm.stopPrank();

    vm.prank(owner);
    book.emergencyWithdrawERC20(address(newMock), 8e18, owner);

    vm.prank(dan);
    vm.expectRevert();
    book.buyOrder(aliceId);
}
```

The test passes, which means that the buy function has been reverted due to insufficient balance.

Recommended Mitigation

- Check for active orders on the token being withdrawn

```
+mapping(address=>bool) public isTokenActive;  // keeps track of token to check if it has any active order
.
.
.
function emergencyWithdrawERC20(address _tokenAddress, uint256 _amount, address _to) external onlyOwner {
    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();
    }
+   assert(!isTokenActive[_tokenAddress]);
    IERC20 token = IERC20(_tokenAddress);
    token.safeTransfer(_to, _amount);

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

- Use governance to prevent malicious owner

Medium

[M-01] USDC supports blacklisting feature, and `Orderbook` contract loses all functionality if it gets blacklisted

Description

- Users are able to place orders for their tokens on their selected price in USDC token only, and only USDC token can be used to fill orders.
- USDC is an ERC-20 token contract which is controlled by Circle. Custodial wallets owned by Circle have the ability to restrict certain addresses of their choosing to be blacklisted from all USDC transactions.

- If the `OrderBook` contract is blacklisted from USDC transactions, then no orders will be filled due to `buyOrder()` function getting reverted on USDC transactions for protocol fees.

```
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);
}
```

Risk

Likelihood:

- Only when `OrderBook` contract gets blacklisted from USDC transactions

Impact:

- No order will be filled due to failing `buyOrder()` function, hence rendering the contract unusable.

Proof of Concept

Add the following ERC-20 contract code for a mock USDC with blacklisting feature in `test/mocks/BlacklistingUSDC.sol`:

Blacklisting USDC token

```
// SPDX-License-Identifier: SEE LICENSE IN LICENSE
```

```solidity
pragma solidity 0.8.26;

import {ERC20} from "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import {Ownable} from "@openzeppelin/contracts/access/Ownable.sol";

contract BlacklistingUSDC is ERC20, Ownable {
    uint8 constant tokenDecimals = 6;
    mapping(address => bool) public isBlacklisted;

    modifier notBlacklisted(address _from, address _to) {
        require(!isBlacklisted[_from], "Sender is blacklisted");
        require(!isBlacklisted[_to], "Recipient is blacklisted");
        _;
    }

    constructor() ERC20("BlacklistingUSDC", "bUSDC") Ownable(msg.sender) {}

    function decimals() public pure override returns (uint8) {
        return tokenDecimals;
    }

    function mint(address to, uint256 value) public {
        uint256 updateDecimals = uint256(tokenDecimals);
        _mint(to, (value * 10 ** updateDecimals));
    }

    function blacklist(address _account) external onlyOwner {
        isBlacklisted[_account] = true;
    }

    function unBlacklist(address _account) external onlyOwner {
        isBlacklisted[_account] = false;
    }

    function transfer(address to, uint256 value) public override notBlacklisted(msg.sender, to) returns (bool) {
        super.transfer(to, value);
        return true;
    }

    function transferFrom(address from, address to, uint256 value) public override notBlacklisted(from, to) returns (bool) {
        super.transferFrom(from, to, value);
        return true;
    }
}
```

Then place the following function in `test/TestOrderBook.t.sol` and run with `forge test --mt test_USDCBlacklistedContract`:

```solidity
function test_USDCBlacklistedContract() public {

    // Setup contract and mock USDC with blacklisting feature, and blacklist the OrderBook contract
    vm.startPrank(owner);
    BlacklistingUSDC busdc = new BlacklistingUSDC();
    book = new OrderBook(address(weth), address(wbtc), address(wsol), address(busdc), owner);
    busdc.blacklist(address(book));
    vm.stopPrank();

    // Create a Sell order from Alice
    vm.startPrank(alice);
    wbtc.approve(address(book), 2e8);
    uint256 aliceId = book.createSellOrder(address(wbtc), 2e8, 180_000e6, 2 days);
    vm.stopPrank();

    // Setup Dan to but Alice's order
    vm.startPrank(dan);
    busdc.approve(address(book), 200_000e6);
    busdc.mint(dan, 180_000e6);

    // buyOrder is expected to fail due to contract being blacklisted
    vm.expectRevert("Recipient is blacklisted");
    book.buyOrder(aliceId);
}
```

This test case passes when the `buyOrder()` function reverts due to contract being blacklisted.

Recommended Mitigation

1. Use a non-blacklistable wrapper like wUSDC token for purchases
2. Use a decentralised token like DAI for purchases

[M-02] Incorrect duration update logic in `OrderBook::amendSellOrder()` can cause orders to be able to remain active for more than intended maximum duration

Description

- `deadlineTimestamp` variable is used to keep track of the maximun timestamp till which the order will be active and is being set in `createSellOrder()`. the maximum duration limit for an order is 3 days.
- The following logic to update `deadlineTimestamp` variable can cause the order to go on for more than the intended limit of 3 days, given that the function is called with a

new duration before order expiry. This calculates the new duration from the moment the `amendSellOrder()` is called, instead of the moment when the order was created.

```
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);
```

Risk

Likelihood:

- Whenever `amendSellOrder()` function is called on an active order with a new timestamp

Impact:

- The order always gets updated with a new duration timestamp, the duration being calculated from the moment the `amendSellOrder()` is called

Proof of Concept

Place this function into `TestOrderBook.t.sol` and run with `forge test --mt test_extendDuration`. This passes, which means that the function is indeed active even after the maximum duration limit of 3 days.

```
function test_extendDuration() public {
    // alice creates sell order for wbtc
    vm.startPrank(alice);
    wbtc.approve(address(book), 2e8);
    uint256 aliceId = book.createSellOrder(address(wbtc), 2e8, 180_000e6, 3 days);
    assert(aliceId == 1);

    // fast forward to just 1 hour before expiry
    vm.warp(block.timestamp + 2 days + 23 hours);

    book.amendSellOrder(aliceId, 2e8, 180_000e6, 3 days);

    // 4 days and 23 hours after order creation
    vm.warp(block.timestamp + 2 days);

    OrderBook.Order memory order = book.getOrder(aliceId);

    assertGt(order.deadlineTimestamp, block.timestamp);
}
```

Recommended Mitigation

Add a new field in `Order` struct which keeps track of the timestamp of creation. Then update the new deadline timestamp from the creation timestamp in `amendSellOrder()`

```
struct Order {
    uint256 id;
    address seller;
    address tokenToSell; // Address of wETH, wBTC, or wSOL
    uint256 amountToSell; // Amount of tokenToSell
    uint256 priceInUSDC; // Total USDC price for the entire amountToSell
    uint256 deadlineTimestamp; // Block timestamp after which the order expires
    bool isActive; // Flag indicating if the order is available to be bought
+   uint256 creationTimestamp;
}
.
.
.
function createSellOrder(
    address _tokenToSell,
    uint256 _amountToSell,
    uint256 _priceInUSDC,
    uint256 _deadlineDuration
) public returns (uint256) {
    if (!allowedSellToken[_tokenToSell]) revert InvalidToken();
    if (_amountToSell == 0) revert InvalidAmount();
    if (_priceInUSDC == 0) revert InvalidPrice();
    if (_deadlineDuration == 0 || _deadlineDuration >
       MAX_DEADLINE_DURATION) revert InvalidDeadline();

    uint256 deadlineTimestamp = block.timestamp + _deadlineDuration;
+   uint256 creationTimestamp = block.timestamp;
    uint256 orderId = _nextOrderId++;

    IERC20(_tokenToSell).safeTransferFrom(msg.sender, address(this),
       _amountToSell);

    // Store the order
    orders[orderId] = Order({
        id: orderId,
        seller: msg.sender,
        tokenToSell: _tokenToSell,
        amountToSell: _amountToSell,
        priceInUSDC: _priceInUSDC,
        deadlineTimestamp: deadlineTimestamp,
-       isActive: true
+       isActive: true,
+       creationTimestamp = creationTimestamp
    });

    emit OrderCreated(orderId, msg.sender, _tokenToSell, _amountToSell,
        _priceInUSDC, deadlineTimestamp);
    return orderId;
}
.
.
.
-   uint256 newDeadlineTimestamp = block.timestamp +
   _newDeadlineDuration;
+   uint256 newDeadlineTimestamp = order.creationTimestamp +
   _newDeadlineDuration;
+   assert(newDeadlineTimestamp > block.timestamp);
```

Low

[L-01] Lack of function to cleanup expired orders causes tokens deposited for orders that have are now expired to be locked forever unless cancelled manually by seller

Description

- All sell orders have an expiration timestamp. Once this expiration timestamp is passed, amendment and purchase of these orders ar enot permitted.
- However, there are no functions to deactivate order and return the locked tokens to the seller

for the concerned order. Only the user can manually call `cancelOrder()` to deactivate the order and claim locked tokens

```
function createSellOrder(
    address _tokenToSell,
    uint256 _amountToSell,
    uint256 _priceInUSDC,
    uint256 _deadlineDuration
) public returns (uint256) {
    if (!allowedSellToken[_tokenToSell]) revert InvalidToken();
    if (_amountToSell == 0) revert InvalidAmount();
    if (_priceInUSDC == 0) revert InvalidPrice();
@>  if (_deadlineDuration == 0 || _deadlineDuration > MAX_DEADLINE_DURATION) revert InvalidDeadline();

@>  uint256 deadlineTimestamp = block.timestamp + _deadlineDuration;
```

Risk

Likelihood:

- Whenever an order expires without getting bought

Impact:

- Tokens are locked and have to be manually claimed through `cancelOrder()`, which can be done only by user

Proof of Concept

Place the following function into `test/TestOrderBook.t.sol` and run with `forge test --mt test_expiredOrders`

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

    assert(aliceId == 1);
    assert(wbtc.balanceOf(alice) == 0);
    assert(wbtc.balanceOf(address(book)) == 2e8);

    // warp to 3 days later, order should have expired due to duration being 2 days
    vm.warp(block.timestamp + 3 days);

    vm.startPrank(dan);
    usdc.approve(address(book), 200_000e6);
    // expect tx to fail due to expired duration
    vm.expectRevert();
    book.buyOrder(aliceId);
    vm.stopPrank();

    // cancel order to claim tokens
    vm.startPrank(alice);
    book.cancelSellOrder(aliceId);
}
```

The test passes, showing that the issue is indeed real.

Recommended Mitigation

Add a `sweepExpiredOrder()` function to clean up expired orders and return the tokens to users. This function may be triggered by a centralised entity

```
+function sweepExpiredOrders() public {
+    for (uint256 _orderId = 1; _orderId < _nextOrderId; _orderId++) {
+        Order storage order = orders[_orderId];
+
+        if (block.timestamp >= order.deadlineTimestamp) {
+            order.isActive = false;
+            IERC20 token = IERC20(order.tokenToSell);
+            token.safeTransfer(order.seller, order.amountToSell);
+        }
+    }
+}
```

[L-02] `OrderBook::getOrderDetailsString()` does not consider tokens other than the core tokens for returning the token symbol

Description

- The `getOrderDetailsString()` returns a string that shows all the info related to an order.
- However, due the following lines the function does not consider the symbols for tokens other than the core tokens. hence, for non-core tokens the symbol does not return any value.

```
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:

*Whenever `getOrderDetailsString()` is called for an order which involves a non core token

Impact:

- Returned string does not contain token symbol

Proof of Concept

Place the following function in `TestOrderBook.t.sol` and run with `forge test --mt test_wrongTokenSymbol -vv`:

```
function test_wrongTokenSymbol() public {
    // deploy a new token and set it to be allowed on the contract
    MockWETH newMock = new MockWETH(18);

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

    newMock.mint(alice, 10e18);

    vm.startPrank(alice);
    newMock.approve(address(book), 10e18);
    uint256 aliceId = book.createSellOrder(address(newMock), 8e18, 1e18, 3 days);

    string memory res = book.getOrderDetailsString(aliceId);

    console2.log(res);
}
```

In the output we get:

```
[.] Compiling...
[.] Compiling 1 files with Solc 0.8.26
[.] Solc 0.8.26 finished in 530.20ms
Compiler run successful!

Ran 1 test for test/TestOrderBook.t.sol:TestOrderBook
[PASS] test_wrongTokenSymbol() (gas: 1292211)
Logs:
  Order ID: 1
Seller: 0xaf6db259343d020e372f4ab69cad536aaf79d0ac
Selling: 8000000000000000000
Asking Price: 1000000000000000000 USDC
Deadline Timestamp: 259201
Status: Active

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 2.71ms (680.12us CPU time)

Ran 1 test suite in 3.62ms (2.71ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

As we can see, the Selling field does not mention the symbol of the token being sold.

Recommended Mitigation

Use the `symbol()` function from ERC20 contract to access the token symbol

```
function getOrderDetailsString(uint256 _orderId) public view returns (string memory details) {
    Order storage order = orders[_orderId];
    if (order.seller == address(0)) revert OrderNotFound(); // Check if order exists

-   string memory tokenSymbol;
+   string memory tokenSymbol = IERC20(order.tokenToSell).symbol();
-   if (order.tokenToSell == address(iWETH)) {
-       tokenSymbol = "wETH";
-   } else if (order.tokenToSell == address(iWBTC)) {
-       tokenSymbol = "wBTC";
-   } else if (order.tokenToSell == address(iWSOL)) {
-       tokenSymbol = "wSOL";
-   }
```

[L-03] Unidentical repetition of status checks in `getOrderDetailsString()` causes one result to be shadowed

Description

- The `getOrderDetailsString()` function checks for the order status

- However, the logi for status checking has been placed twice with different outputs, shadowing one of the logics and wasting gas.

```
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";
    }
```

As we can see, status is getting checked twice and for expired orders we have different strings - “Expired (Active but past deadline)” and “Expired (Awaiting Cancellation)”. This can lead to irregularities on frontends that depend on this function

Risk

Likelihood:

- Whenever `getOrderDetailsString()` is called

Impact:

- unexpected output where output from first logic is expected

Proof of Concept

Place the following function into `TestOrderbook.t.sol` and run with `forge test --mt test_wrongTokenStatus -vv`:

```
function test_wrongTokenStatus() public {
    weth.mint(alice, 10e18);

    vm.startPrank(alice);
    weth.approve(address(book), 10e18);
    uint256 aliceId = book.createSellOrder(address(weth), 8e18, 1e18, 3 days);

    vm.warp(block.timestamp + 4 days);

    string memory res = book.getOrderDetailsString(aliceId);

    console2.log(res);
}
```

We get this output:

```
[.] Compiling...
[.] Compiling 2 files with Solc 0.8.26
[.] Solc 0.8.26 finished in 537.32ms
Compiler run successful!

Ran 1 test for test/TestOrderBook.t.sol:TestOrderBook
[PASS] test_wrongTokenStatus() (gas: 313139)
Logs:
  Order ID: 1
Seller: 0xaf6db259343d020e372f4ab69cad536aaf79d0ac
Selling: 8000000000000000000 wETH
Asking Price: 1000000000000000000 USDC
Deadline Timestamp: 259201
Status: Expired (Awaiting Cancellation)

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 2.42ms (354.09us CPU time)

Ran 1 test suite in 6.12ms (2.42ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

For frontend expecting “Expired (Active but past deadline)”, this output can be confusing.

Recommended Mitigation

Remove one of the logics and ensure only one consistent pattern is followed

```
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";
-    }
```