Protocol Audit Report

@akronim26

July 9, 2025

Prepared by: akronim26

Table of Contents

Protocol Summary

The OrderBook protocol is a decentralized peer-to-peer trading system built on Ethereum that enables users to trade wrapped tokens (wETH, wBTC, wSOL) against USDC through an on-chain order book mechanism. The protocol operates as a non-custodial marketplace where sellers can list their tokens at fixed prices with expiration deadlines, and buyers can directly fill these orders.

Core Functionality

Order Management: - Sellers lock their tokens in the contract and create sell orders with specified prices in USDC and deadline timestamps - Orders are tracked using unique order IDs and remain active until filled or expired - Sellers retain full control over their listings and can amend or cancel orders before expiration

Trading Mechanism: - Buyers can purchase tokens by paying the listed USDC amount plus a 3% protocol fee - Token transfers are secured using OpenZeppelin's SafeERC20 library - Orders automatically expire after their deadline, allowing sellers to reclaim their tokens

Key Features: - Support for wETH, wBTC, wSOL trading against USDC - Order amendment capabilities (price, amount, deadline changes) - Emergency withdrawal functionality for non-core tokens - Human-readable order information via string formatting - Deadline enforcement to prevent stale listings

Security Model: - Uses Ownable pattern for administrative functions - Implements strict validation rules to prevent misuse - Employs SafeERC20 for secure token transfers - Enforces maximum deadline duration of 3 days

The protocol aims to replicate traditional order book functionality from centralized exchanges while maintaining the decentralized, trustless nature of DeFi applications.

Disclaimer

akronim26 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. 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

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

The findings described in this document correspond the following commit hash:

```
fdef247b2f2bd7c0f9c19310406c8e072d9ffda4
```

Scope

The audit scope includes the following contracts:

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

Roles

Protocol Owner: - Can set allowed sell tokens via `setAllowedSellToken()` - Can perform emergency withdrawals via `emergencyWithdrawERC20()` - Can withdraw accumulated fees via `withdrawFees()` - Inherits from OpenZeppelin's `Ownable` contract

Sellers: - Can create sell orders via `createSellOrder()` - Can amend their orders via `amendSellOrder()` - Can cancel their orders via `cancelSellOrder()` - Must approve tokens before creating orders - Receive USDC payments when orders are filled

Buyers: - Can purchase orders via `buyOrder()` - Must approve USDC before purchasing - Pay 3% protocol fee on all purchases - Receive tokens upon successful purchase

Viewers: - Can query order details via `getOrder()` - Can get formatted order information via `getOrderDetailsString()` - No state-changing permissions

Executive Summary

Issues found

Severity	Number of issues found
High	3
Medium	2
Low	3
Info	1
Gas Optimizations	3
Total	12

Findings

High

[H-1] Use of a Single Owner in `OrderBook` contract introduces Centralization Risks

Description:
The `OrderBook` contract currently employs a single owner model, where a single externally owned account (EOA) is granted exclusive control over all privileged administrative functions. This design pattern is common for simplicity but introduces significant centralization risks. The owner is able to execute critical operations such as setting allowed sell tokens, performing emergency withdrawals, and withdrawing accumulated protocol fees. If the owner's private key is compromised, lost, or if the owner acts maliciously, the protocol's security, user funds, and overall trustlessness can be severely undermined. Furthermore, this model may not align with the expectations of a decentralized protocol, as users and integrators may be wary of excessive centralized control.

Impact:
The concentration of authority in a single address means that a single point of

failure exists for the protocol's most sensitive operations. If the owner's key is compromised, an attacker could disrupt the system. Even in the absence of malicious intent, mistakes or loss of access by the owner could result in the protocol becoming unmanageable. This centralization can also erode user confidence, as users may be reluctant to interact with a protocol that can be unilaterally altered or shut down by one party.

Recommended Mitigation:
To address these risks, it is strongly recommended to replace the single owner model with a multi-signature (multi-sig) wallet for all administrative and privileged actions. A multi-sig wallet requires a predefined number of trusted parties (signers) to approve any sensitive transaction, such as changing protocol parameters or withdrawing funds. This approach distributes control and greatly reduces the risk of unilateral or malicious actions, as no single party can act alone. Multi-sig solutions, such as Gnosis Safe, are widely used in the industry and provide robust security guarantees. Additionally, using a multi-sig can improve transparency and accountability, as all administrative actions require consensus among multiple stakeholders. For further reading and best practices on implementing multi-sig administration, please refer to the following article:

- https://blog.openzeppelin.com/admin-accounts-and-multisigs

[H-2] Precision Loss in `buyOrder` Function in the `OrderBook` Contract

Description:
The `buyOrder` function in the `OrderBook` contract calculates the `protocolFee` using integer division, which can result in a loss of precision. In Solidity, all arithmetic is performed using integers, and any division between two integers will always round down (truncate) the result, discarding any fractional component. This means that when calculating a percentage-based fee, such as a 3% protocol fee, the actual fee collected may be slightly less than the intended amount, especially for small order values or when the fee rate is low. For example, if the order price is 10 USDC, then `(10 * 3)/ 100` will yield 0, not 0.3, resulting in no fee being collected for that transaction. Over many transactions, these small discrepancies can add up.

This issue is particularly relevant in DeFi protocols where fee calculations are expected to be precise and predictable. Users may also notice inconsistencies in the fees charged, which could lead to confusion or a perception of unfairness. Furthermore, attackers or sophisticated users could deliberately structure their trades to minimize fees by exploiting this rounding behavior.

Impact:
Precision loss in fee calculation can lead to the protocol collecting less in fees than expected over time. This discrepancy may accumulate, resulting in a significant loss in protocol revenue, especially as the number of small-value trades increases.

Additionally, it can create inconsistencies in fee distribution, making it difficult to accurately account for protocol earnings or to distribute fees to stakeholders. In some cases, users may be able to exploit this behavior by splitting large trades into multiple small ones to avoid paying fees altogether, further reducing protocol income and potentially undermining the intended economic model.

Proof of Concept:
Add the following test into the `TestOrderBook.t.sol` to demonstrate how the protocol fee can be zero for small order values due to integer division truncation:

```
function test_protocolFeePrecisionLoss() public {
        uint256 priceInUSDC = 10;
        uint256 expectedFee = (priceInUSDC * 3) / 100;

        vm.startPrank(bob);
        weth.approve(address(book), 2);
        uint256 orderId = book.createSellOrder(
            address(weth),
            1,
            priceInUSDC,
            1 days
        );
        vm.stopPrank();

        address buyer = address(0x5678);
        usdc.mint(buyer, priceInUSDC);
        vm.startPrank(buyer);
        usdc.approve(address(book), priceInUSDC);
        book.buyOrder(orderId);
        vm.stopPrank();

        assertEq(book.totalFees(), expectedFee);
        assertEq(expectedFee, 0);
    }
```

Recommended Mitigation:
There are two main strategies to address this issue, both of which can help ensure that the protocol fee is calculated more accurately and fairly for all users:

- **Implement a rounding-up strategy:** When calculating the `protocolFee`, add a check to round up any fractional result. This can be done by adding the denominator minus one to the numerator before performing the division, or by simply adding +1 to the result if there is a remainder. This ensures that even the smallest non-zero fee is collected, and that the protocol does not systematically undercharge on small transactions. For example, the calculation could be modified as follows:

  ```
  uint256 protocolFee = (priceInUSDC * 3 + 99) / 100;
  ```

  This approach ensures that any fractional part is rounded up, so the protocol always collects at least the minimum intended fee.

- **Use a FixedPoint math library:** Integrate an external library (such as Open-

Zeppelin's FixedPoint or PRBMath) or implement your own fixed-point arithmetic to handle decimal calculations with greater precision. Fixed-point math allows you to represent fractional values accurately by scaling integers, which is a common practice in Solidity to avoid the pitfalls of integer division. This method is especially useful if the protocol may need to support more complex fee structures or calculations in the future.

Both strategies have their trade-offs: rounding up is simple and gas-efficient, while fixed-point math provides more flexibility and precision at the cost of slightly higher complexity and gas usage. The choice depends on the protocol's requirements and the desired user experience.

For further information and best practices on handling precision and rounding in Solidity, you may refer to this article, which discusses common pitfalls and recommended solutions for precision loss in smart contract arithmetic.

[H-3] Missing Deadline Check in `cancelSellOrder` Function in the `OrderBook` Contract

Description:
The `cancelSellOrder` function in the `OrderBook` contract currently lacks a check to determine whether the order's deadline has already passed before allowing a cancellation. This means that a seller is able to cancel their order even after the order has expired, which may not be consistent with the intended business logic of the protocol. In a typical order book system, expired orders are expected to be handled differently (e.g., marked as expired or reclaimed by the protocol), and allowing post-deadline cancellations could result in ambiguous or conflicting order states. Furthermore, this omission could make it more difficult for off-chain indexers, dApps, or users to reliably interpret the status of orders, as an order that is both expired and cancelled may not be clearly distinguishable from one that was cancelled before expiration.

Impact:
Permitting the cancellation of orders after their deadline has passed can undermine the integrity and predictability of the order book. It may allow sellers to bypass intended protocol restrictions, potentially leading to edge cases where expired orders are incorrectly marked as cancelled rather than expired. This can create confusion for users, complicate the logic for frontends and indexers, and may even open up subtle attack vectors if other protocol logic assumes that expired orders cannot be cancelled. In addition, it could make dispute resolution and auditing more difficult, as the true lifecycle of an order becomes less clear.

Recommended Mitigation:
To address this issue, it is recommended to add a deadline check in the `cancelSellOrder` function. Specifically, the function should revert if the current block timestamp is greater than or equal to the order's `deadlineTimestamp`,

thereby preventing cancellation of orders that have already expired. This ensures that only active, non-expired orders can be cancelled by the seller, maintaining a clear and consistent order state throughout the protocol. Additionally, consider updating documentation and user interfaces to reflect this behavior, so that users are aware that expired orders must be handled differently from cancelled ones.

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

        if (order.seller == address(0)) revert OrderNotFound();
        if (order.seller != msg.sender) revert NotOrderSeller();
        if (!order.isActive) revert OrderAlreadyInactive();
+       if (block.timestamp >= order.deadlineTimestamp) revert OrderExpired
    ();

        order.isActive = false;

        IERC20(order.tokenToSell).safeTransfer(
            order.seller,
            order.amountToSell
        );

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

Medium

[M-1] Missing Check for Zero Address in the `createSellOrder` function in the `OrderBook` contract can lead to an edge case where the seller of the order is `address(0)`

Description:
The `createSellOrder` function in the `OrderBook` contract currently does not include a check to ensure that the seller address is not the zero address (`address(0)`). While this may seem like a minor omission, it can introduce subtle vulnerabilities and edge cases into the protocol. In Ethereum, `address(0)` is a special value that is often used to represent the absence of an address or as a sentinel value. Allowing an order to be created with `address(0)` as the seller can result in orders that are effectively orphaned and cannot be interacted with in a meaningful way.

This issue is particularly relevant in scenarios where the function is called via a contract using `delegatecall`, or if there is a misconfiguration in the calling context. In such cases, `msg.sender` could be set to `address(0)`, even though this is not possible through direct user interaction. If the protocol does not explicitly prevent this, it is possible for an order to be created with an invalid seller, which can disrupt the normal operation of the order book and complicate order management.

Impact:
The absence of a zero address check for the seller can disrupt the protocol's functionality in several ways. First, any order created with `address(0)` as the seller will

be unmanageable: it cannot be cancelled, amended, or filled, since all subsequent functions that interact with the order will check for a valid seller and revert if the seller is `address(0)`. This can lead to stuck or unusable orders in the system, which may clutter the order book and create confusion for users and off-chain indexers. Additionally, the presence of such invalid orders could complicate auditing, analytics, and user interface logic, as special handling would be required to filter out or ignore these orders.

Proof of Concept:
The following scenario demonstrates how this vulnerability could manifest:

1. Although a regular user cannot directly call a function with `msg.sender` set to `address(0)` on Ethereum, there are certain edge cases—such as when a contract uses `delegatecall` or is misconfigured—where `msg.sender` could be set to `address(0)`. If the `createSellOrder` function does not check for this condition, it is theoretically possible for an order to be created with `seller` set to `address(0)`.
2. Once such an order exists, any attempt to interact with it (e.g., to buy, cancel, or amend the order) would always fail, because the following check would revert:

```
if (order.seller == address(0)) revert OrderNotFound();
```

Recommended Mitigation: Add a check to ensure the seller is not the zero address.

```
        function createSellOrder(
            address _tokenToSell,
            uint256 _amountToSell,
            uint256 _priceInUSDC,
            uint256 _deadlineDuration
        ) public returns (uint256) {

+           require(msg.sender != address(0), "Zero address");
            if (!allowedSellToken[_tokenToSell]) revert InvalidToken();
            if (_amountToSell == 0) revert InvalidAmount();
            .
            .
            .
        }
```

[M-2] Use of `abi.encodePacked()` instead of `abi.encode()` in `getOrderDetailsString` function in the `OrderBook` contract can lead to String Collisions

Description:
The `getOrderDetailsString` function in the `OrderBook` contract is responsible for returning a human-readable string representation of an order's details. In its current implementation, this function uses `abi.encodePacked()` to concatenate and

encode various order fields into a single string. While `abi.encodePacked()` is a convenient way to tightly pack multiple values together, it is not collision-resistant when used for string concatenation. Specifically, `abi.encodePacked()` performs tight packing of the input data, which means that different combinations of input values can produce the same output bytes if their boundaries are not clearly defined. This can result in collisions, where two distinct sets of order details are encoded into identical byte sequences, potentially leading to incorrect order information being returned to users or external systems that rely on this function.

This issue is particularly problematic in contexts where the string representation is used for display, logging, or off-chain processing, as it undermines the integrity and uniqueness of order data. In the worst case, a malicious actor could craft order details that intentionally collide with another order's string representation, causing confusion or misrepresentation in the order book interface or in downstream integrations.

Impact:
If two different sets of order details produce the same packed encoding, it could result in incorrect or ambiguous information being displayed to users, dApps, or off-chain services that consume the output of `getOrderDetailsString`. This spoils the integrity and reliability of the order book, as users may be misled about the true state or contents of an order. In addition, this vulnerability could be exploited in edge cases to deliberately misrepresent order data, potentially enabling phishing, spoofing, or other forms of manipulation within the protocol or its ecosystem. Over time, such issues can erode user trust and make it more difficult to audit or verify the correctness of order data.

Proof of Concept:
Add the following test to `TestOrderBook.t.sol` to demonstrate the collision:

```
function test_encodePackedCanCollide() external pure {
        assertEq(keccak256(abi.encodePacked("abc", "def")), keccak256(abi.encodePacked("abcdef")));
        assert(keccak256(abi.encode("abc", "def")) != keccak256(abi.encode("abcdef")));
    }
```

Recommended Mitigation: Replace the `abi.encodePacked()` with `abi.encode()`

```
    function getOrderDetailsString(
        uint256 _orderId
    ) public view returns (string memory details) {
        Order storage order = orders[_orderId];
        if (order.seller == address(0)) revert OrderNotFound();
        .
        .
        .
        details = string(
-           abi.encodePacked(
+           abi.encode(
                "Order ID: ",
                order.id.toString(),
                .
                .
                .
    }
```

Low

[L-1] Solidity pragma is too broad, increasing Risk of Vulnerabilities

Description: The Solidity pragma directive in the `OrderBook` contract currently permits compilation with a broad range of compiler versions. This practice can introduce security vulnerabilities, as different compiler versions may contain undiscovered bugs, security issues, or breaking changes that can affect the contract's behavior in subtle and unpredictable ways. For example, certain compiler versions may handle arithmetic operations, memory management, or other language features differently, which could result in inconsistent or unintended contract logic. Additionally, using a wide pragma range can make it more difficult to audit and maintain the contract over time, as it is unclear which compiler version was used for deployment, and future upgrades to the Solidity compiler may introduce incompatibilities or new risks.

Impact: Allowing a broad range of compiler versions increases the risk that the contract will be compiled and deployed with a version of Solidity that behaves differently than intended, or that contains known or unknown vulnerabilities. This can lead to subtle bugs, unexpected behavior, or even critical security issues that may not be immediately apparent during development or testing. Furthermore, it complicates the auditing process and can undermine the reliability and predictability of the contract, as different environments may produce different bytecode or execution results. In the worst case, this could result in loss of funds, contract malfunction, or exploitation by malicious actors who are aware of version-specific vulnerabilities.

Recommended Mitigation: To mitigate these risks, it is strongly recommended to specify an exact compiler version (e.g., `pragma solidity 0.8.24;`) or, at minimum, a tightly constrained version range that excludes known-vulnerable or deprecated versions. This ensures that the contract will always be compiled with a known, tested, and trusted compiler version, reducing the likelihood of unexpected issues and making the contract easier to audit and maintain. Additionally, documenting the chosen compiler version in the contract and deployment scripts can further enhance transparency and security.

[L-2] Use of `block.timestamp` for Deadline Checks can introduce vulnerabilities due to Miner Manipulation

Description:
The use of `block.timestamp` to enforce order deadlines in the `OrderBook` contract is not entirely safe, as block timestamps can be slightly manipulated by miners. Although Ethereum protocol rules restrict the extent of this manipulation (typically within a range of a few seconds), it is still possible for miners to influence the timestamp of a block they are producing. This subtle control can be exploited in certain scenarios, particularly in time-sensitive applications such as decentralized exchanges or order books, where the precise timing of order expiration is critical. Relying on `block.timestamp` for deadline enforcement can therefore introduce a minor, but non-negligible, attack surface that may be abused to gain an unfair advantage or disrupt the intended operation of the protocol.

Impact:
Malicious miners could manipulate the block timestamp to prematurely expire or artificially extend the validity of orders, potentially disrupting the intended functionality of the order book and enabling unfair trading advantages. For example, a miner could cause a legitimate buy transaction to fail by advancing the timestamp just enough to make an order appear expired, or conversely, delay expiration to allow a favored party to fill an order that should have already lapsed. Over time, repeated exploitation of this vector could erode user trust in the fairness and reliability of the protocol, especially in high-frequency or competitive trading environments.

Proof of Concept:
The following scenario demonstrates the vulnerability in practice:

1. A user creates a sell order with a deadline set to expire in 60 seconds (using `block.timestamp + 60`).
2. Another user submits a buy transaction to fill this order just before the deadline is reached.
3. A malicious miner, who is able to include this transaction in a block they mine, intentionally sets the `block.timestamp` forward by several seconds (within the allowed protocol range).
4. As a result, the order's deadline is considered expired, and the buy transaction reverts, even though it was submitted in time.
5. The miner or a favored user can then submit their own transaction in the next block, potentially filling the order themselves or preventing others from doing so.
6. This demonstrates how even a small manipulation of `block.timestamp` can impact the fairness and reliability of the order book, especially in scenarios where timing is critical and users expect precise deadline enforcement.
7. In more competitive or adversarial environments, this could be used to front-

run, grief, or otherwise manipulate the outcome of trades, undermining the protocol's integrity.

Recommended Mitigation:
To mitigate this risk, consider using `block.number` for deadline checks instead of `block.timestamp`, as block numbers are deterministic and far less susceptible to manipulation by miners. If precise timing is required, block numbers can be converted to approximate timestamps using the average block time, though this introduces some granularity. Additionally, ensure that critical logic does not rely on exact timestamp precision, and document any timing assumptions clearly in the contract and user documentation. Where possible, design the protocol to tolerate small timing discrepancies, or provide a buffer period to reduce the impact of minor timestamp manipulation. Regularly review and test deadline logic to ensure it remains robust against miner manipulation and other edge cases.

[L-3] The `withdrawFees`, `buyOrder`, `amendSellOrder` and `createSellOrder` functions in the `OrderBook` contract do not follow the Checks-Effects-Interactions (CEI) Pattern

Description:
The functions `withdrawFees`, `buyOrder`, and `createSellOrder` in the `OrderBook` contract currently perform external calls—such as token transfers—*before* updating the contract's internal state. This approach is contrary to the Checks-Effects-Interactions (CEI) pattern, a widely recommended best practice in smart contract development. The CEI pattern dictates that all necessary checks (such as input validation and access control) and all updates to the contract's internal state should be completed *prior* to any external interactions, including calls to other contracts or token transfers. By not adhering to this pattern, the contract increases its exposure to certain classes of vulnerabilities.

Impact:
Failure to follow the CEI pattern can expose the contract to reentrancy attacks and other unexpected behaviors. If an external contract is called before the internal state is updated, a malicious contract could exploit this by re-entering the function (or another function that depends on the same state) and manipulating the contract's state in an unintended way. This could potentially result in the loss of funds, double-withdrawals, or inconsistent contract state. Even if the contract is not directly vulnerable to reentrancy, performing external calls before state updates can make the code harder to reason about and audit, increasing the risk of subtle bugs or future vulnerabilities as the codebase evolves.

Recommended Mitigation:
It is strongly recommended to refactor the `withdrawFees`, `amendSellOrder`, and `buyOrder` functions to strictly follow the Checks-Effects-Interactions pattern, as has been suggested for the `createSellOrder` function. Specifically, ensure that all

input validation, access control, and updates to the contract's internal state are performed *before* any external calls, such as token transfers or calls to other contracts. This reduces the attack surface for reentrancy and makes the contract's behavior more predictable and secure. Additionally, consider reviewing all functions that interact with external contracts to ensure they follow this pattern, and document the rationale for the order of operations to aid future audits and maintenance. Adhering to CEI not only improves security but also enhances the clarity and maintainability of the contract code.

```
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 orderId = _nextOrderId++;

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

        orders[orderId] = Order({
            id: orderId,
            seller: msg.sender,
            tokenToSell: _tokenToSell,
            amountToSell: _amountToSell,
            priceInUSDC: _priceInUSDC,
            deadlineTimestamp: deadlineTimestamp,
            isActive: true
        });

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

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

Informational

[I-1] Lack of Natspec in `OrderBook` contract

Description:
The `OrderBook` contract currently lacks Natspec documentation for its functions. Natspec (Ethereum Natural Specification Format) is a standardized way to document Solidity code, providing structured comments that describe the purpose, parameters, return values, and potential side effects of functions. Without these comments, it can be challenging for users, integrators, and auditors to quickly understand the intent, expected behavior, and correct usage of each function. This lack of documentation can also hinder the generation of accurate developer documentation and reduce the overall maintainability of the codebase.

Impact:
The absence of Natspec documentation increases the risk of misunderstandings or misuse of the contract's functions. Developers and users may misinterpret the function's intent, leading to incorrect integrations or unexpected behaviors. Additionally, auditors may spend more time trying to infer the purpose and expected outcomes of functions, which can slow down the review process and increase the likelihood of missing subtle issues.

Recommended Mitigation:
It is recommended to add comprehensive Natspec documentation to every public and external function in the `OrderBook` contract. Each function should include a description of its purpose, detailed explanations of all input parameters, the meaning of its return values, and any important side effects or requirements (such as access control or reentrancy considerations). This will improve code readability, facilitate easier integration for third-party developers, and help ensure that the contract is used as intended.

Gas

[G-1] Functions in `OrderBook` contract that are Never Called Within the Same Contract and are marked as `public` should be marked as `external` to Save Gas

Description:
In the current implementation of the `OrderBook` contract, several functions are declared with **`public`** visibility. However, these functions are never invoked from within the contract itself, they are only intended to be called externally by users or other contracts. In Solidity, functions marked as **`public`** can be accessed both

externally and internally, while `external` functions can only be called from outside the contract. Marking such functions as `external` instead of **`public`** is a best practice when internal calls are not required.

Impact:
If left unaddressed, this inefficiency can accumulate, especially as the protocol scales and more users interact with these functions. It also sets a precedent for suboptimal gas usage, which can be avoided with a simple change in function visibility. Additionally, using the most restrictive and appropriate visibility modifiers improves code clarity and maintainability, making it easier for future developers and auditors to understand the intended usage of each function.

Recommended Mitigation:
Review all functions in the `OrderBook` contract that are currently marked as **`public`** and are not called internally. Change their visibility from **`public`** to `external` to optimize gas usage. This includes, but may not be limited to, functions such as `createSellOrder`, `amendSellOrder`, `cancelSellOrder`, `buyOrder`, `getOrder`, and `getOrderDetailsString`. Adopting this practice will help reduce gas costs for users and improve the overall efficiency of the contract.

```diff
    function createSellOrder(
            address _tokenToSell,
            uint256 _amountToSell,
            uint256 _priceInUSDC,
            uint256 _deadlineDuration
-       ) public returns (uint256) {
+       ) external returns (uint256) {
        .
        .
        .
    }
```

Similarly, this practice should be repeated for `amendSellOrder`, `cancelSellOrder`, `buyOrder`, `getOrder`, and `getOrderDetailsString` functions in the `OrderBook` contracts.

[G-2] Missing check for Sufficient Token Balance in `emergencyWithdrawERC20` function in the `OrderBook` contract

Description:
The `emergencyWithdrawERC20` function in the `OrderBook` contract currently lacks a check to verify that the contract holds a sufficient balance of the specified ERC20 token before attempting to process a withdrawal. As a result, if the contract owner tries to withdraw more tokens than are actually available in the contract, the function will proceed until it reaches the `token.safeTransfer(_to, _amount);` statement, at which point the transaction will revert due to insufficient balance. This revert occurs only after the function has already performed other checks and com-

putations, leading to unnecessary gas consumption and a less user-friendly experience.

Additionally, the absence of a pre-transfer balance check can make it harder for users and integrators to quickly diagnose why a withdrawal transaction failed, as the revert will be triggered deep within the ERC20 transfer logic rather than by a clear, descriptive error at the start of the function.

Impact:
Failing to check the contract's token balance before attempting a withdrawal can result in wasted gas for the contract owner, who may repeatedly attempt to withdraw an unavailable amount without understanding the cause of the failure. Each failed attempt incurs gas costs, and the error message may not be immediately clear, especially if the underlying ERC20 implementation does not provide detailed revert reasons. This inefficiency can be particularly frustrating in emergency situations where quick and predictable access to funds is critical. Moreover, the lack of an explicit check reduces the overall robustness and clarity of the contract's code.

Proof of Concept:
1. Suppose the contract holds 100 tokens of a particular ERC20 token. 2. The owner initiates a call to `emergencyWithdrawERC20` with `_amount = 200`. 3. The function performs its initial checks (such as verifying the token is not a core protocol token and that the recipient address is valid), but does not check the contract's token balance. 4. The function then calls `token.safeTransfer(_to, _amount);`, which reverts because the contract does not have enough tokens to fulfill the transfer. 5. The owner pays gas for the failed transaction, and the revert message may not clearly indicate that the failure was due to insufficient contract balance.

Recommended Mitigation:
To address this issue, add an explicit check at the beginning of the `emergencyWithdrawERC20` function to ensure that the contract's balance of the specified token is at least equal to the requested withdrawal amount. If the balance is insufficient, the function should revert immediately with a clear and descriptive error message, saving gas and improving the user experience. For example:

```
    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);
+       require(_amount > token.balanceOf(address(this)), "Excess
    withdrawal error")
        token.safeTransfer(_to, _amount);

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

[G-3] Unnecessary Repeated Checks for `status` in `getOrderDetailsString` Function Increase Gas Costs

Description:
The `getOrderDetailsString` function in the `OrderBook` contract currently suffers from redundant and repeated conditional checks when determining the `status` string of an order. In its present implementation, the function uses multiple overlapping `if` and `else if` statements to evaluate the state of an order, such as whether it is active, expired, or inactive. These checks often cover the same logical ground more than once, leading to unnecessary code execution paths. This redundancy not only makes the function more difficult to read and understand, but also increases the cognitive load for developers and auditors who need to reason about the correctness of the status assignment logic. Additionally, the presence of repeated logic can make future maintenance more error-prone, as changes in one part of the status logic may not be consistently reflected elsewhere.

Impact:
The impact of these redundant checks is twofold. First, from a performance perspective, every unnecessary conditional branch in a Solidity function can contribute to higher gas consumption, especially if the function is called in a context where gas costs matter (such as from another contract or via a transaction that requires on-chain execution). While `getOrderDetailsString` is a `view` function and does not modify contract state, it can still be invoked in a way that incurs gas costs, and any inefficiency in its logic is magnified with frequent use. Second, from a code quality and security standpoint, the unnecessary complexity introduced by repeated checks makes the function harder to audit and maintain. This can increase the risk of subtle bugs, such as inconsistent status reporting or missed edge cases, and can slow down the process of onboarding new developers or conducting security reviews.

Moreover, the current approach can lead to confusion for users and integrators who rely on the status string for downstream logic or user interface display. If the logic for determining status is not straightforward and centralized, it becomes more difficult to ensure that the reported status accurately reflects the true state of the order, especially as the contract evolves over time.

Recommended Mitigation:
To address these issues, it is recommended to refactor the `getOrderDetailsString` function to eliminate all unnecessary and repeated conditional checks when determining the `status` string. Instead, implement a single, well-structured conditional block (such as a clear **`if-else if-else`** ladder or a **`switch`**-like structure) that covers all possible order states in a mutually exclusive and collectively exhaustive manner. This approach will not only improve the readability and maintainability of the code, but also ensure that the function executes as efficiently as possible, minimizing gas usage for callers. Additionally, a streamlined status determination logic will make it easier to audit the function for correctness and to extend or modify it in the future if new order states are introduced. Consider documenting the logic with comments to further aid understanding and future maintenance.

```
function getOrderDetailsString(
        uint256 _orderId
    ) public view returns (string memory details) {
        Order storage order = orders[_orderId];
        if (order.seller == address(0)) revert OrderNotFound();
        .
        .
        .
        string memory status = order.isActive
            ? (
                block.timestamp < order.deadlineTimestamp
                    ? "Active"
-                   : "Expired (Active but past deadline)"
+                   : "Expired"
            )
            : "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";
-       }
        .
        .
        .
    }
```