FAILSAFE

Prepared for
dbook

Prepared by
Failsafe

17 December 2025

dbook

Smart Contract Audit Report

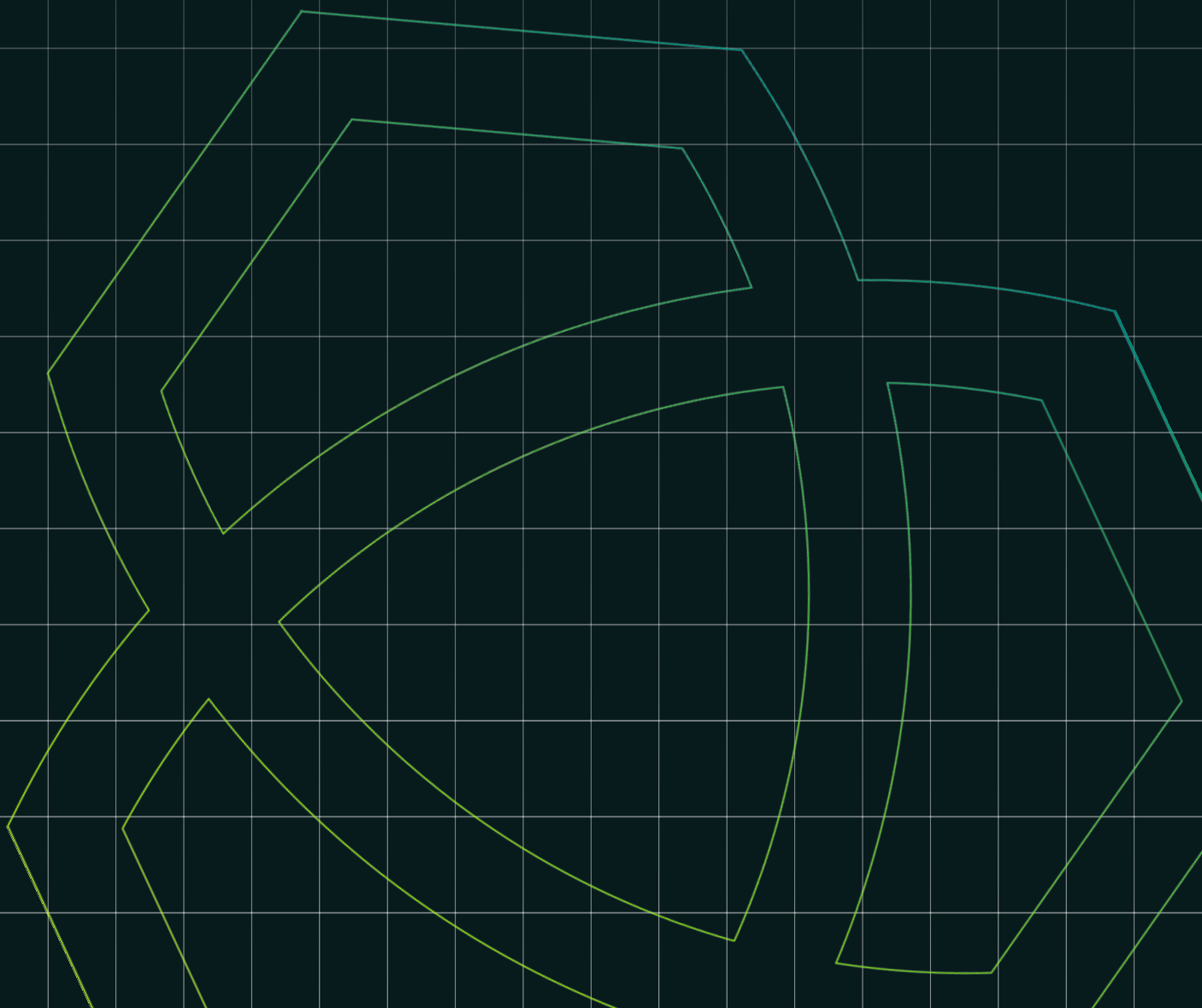

Table of Contents

Project Details

Project	dbook
Repository	https://github.com/0xnerdz/dbook-contracts
Blockchain	MegaEth
Initial Commit	195ef273af27598e7f175a4569a09f3c65412bab
Final Commit	c122647b15cede5d532c0c15525a328414314ff4 e1eff36d2505d7da8e9139f4b2c069c65b0dc3d6 58f9f45f5f98c99b98ecf5481de3337fb27b7d19
Timeline	24 November 2025 - 17 December 2025 Final Report: 17 December 2025

Structure & Organization of The Security Report

Issues are tagged as “Open”, “Acknowledged”, “Partially Resolved”, “Resolved” or “Closed” depending on whether they have been fixed or addressed.

- Open: The issue has been reported and is awaiting remediation from developer team.
- Acknowledged: Reviewed by the development team and accepted based on design intent and risk assessment.
- Partially Resolved: Mitigations have been applied, yet some risks or gaps still remain.
- Resolved: The issue has been fully addressed and no further work is necessary.
- Closed: The issue is deemed no longer pertinent or actionable.

Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

Critical	The issue affects the platform in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.
High	The issue affects the ability of the platform to compile or operate in a significant way.
Medium	The issue affects the ability of the platform to operate in a way that doesn’t significantly hinder its behavior.
Low	The issue has minimal impact on the platform’s ability to operate.
Info	The issue is informational in nature and does not pose any direct risk to the platform’s operation.

Executive Summary

FailSafe was engaged to conduct a comprehensive security audit of dbook's decentralized order book smart contract system, focusing on identifying critical vulnerabilities and security weaknesses that could impact user funds, protocol integrity, and market fairness. Our security team performed an in-depth analysis combining extensive manual code review with automated tools and threat modeling to ensure thorough coverage of potential attack vectors across the core trading infrastructure.

The audit examined the primary in-scope contracts including OrderBook.sol, OrderBookExtended.sol, Manager.sol, OrderBookFactory.sol, and supporting libraries (OrderBookAux.sol, SafeTokenTransfer.sol, PackedFenwick64.sol). The analysis uncovered thirteen significant vulnerabilities, including one critical finding and eight high-severity findings that pose substantial risks to protocol solvency, user funds, and market fairness.

Within the in-scope contracts, the analysis identified that partially filled buy limit orders with price improvement cause permanent fund loss. When a buy order partially matches at a better price and the remainder rests on the book, the contract pulls the full quote amount for the original order size but never refunds the surplus after accounting for the partial fill and resting remainder. This surplus remains permanently stranded in the contract, breaking the conservation invariant and causing unrecoverable fund loss for users placing limit buy orders.

The OrderBook contract assumes exact ERC20 transfer semantics throughout its collateral handling and matching flows. When fee-on-transfer tokens are used as either BASE_TOKEN or QUOTE_TOKEN, the contract receives less than the nominal transfer amount, but accounting proceeds as if the full amount was received. This creates an immediate discrepancy between recorded balances and actual token holdings, causing protocol insolvency after the first trade and enabling systematic value extraction where attackers can withdraw more tokens than the contract actually holds. The Manager contract's registration process enforces bytecode normalization and parameter consistency but performs no semantic validation of token mint controls, allowing freely mintable tokens to be registered. Once registered, attackers can mint unlimited tokens at zero cost and drain all scarce assets from makers, leaving them with worthless minted tokens.

The audit also identified significant risks in privileged operations. The Manager contract allows ADMIN_ROLE to atomically change the fee collector address without any timelock mechanism, enabling a compromised or malicious admin to redirect all accumulated protocol fees to an attacker-controlled address in a single transaction. Additionally, ADMIN_ROLE can overwrite the canonical OrderBook pointer for existing trading pairs without migrating user state, causing users to lose UI access to their holdings while funds remain locked in the old OrderBook contract.

Order matching and priority mechanisms contain several vulnerabilities that undermine market fairness. Attackers can place large orders to capture early FIFO priority positions, then reduce order size via decreaseOrderSize(), maintaining their priority position while freeing up most of their locked capital. This enables unfair advantage over honest makers at the same price level. An attacker can also block legitimate market makers from placing post-only orders at specific price levels by placing minimal-sized dust orders, creating severe economic asymmetry where an attacker can block unlimited amounts of professional market maker liquidity for minimal cost.

Additional findings include rebasing token accounting breaks leading to fund lockup, order history view denial of service that progressively worsens over time, minQuoteAmount bypasses enabling dust spam, accounting precision issues causing systematic fee under-collection, extended flow contracts inheriting token risk, and slippage guard semantic mismatches. These issues present significant risks to user funds, protocol solvency, market fairness, and operational reliability. Remediation strategies follow industry-standard patterns including balance-delta validation for fee-on-transfer detection, token whitelisting with semantic validation, timelock governance for privileged operations, priority reset mechanisms, and mathematical precision improvements.

Summary of Findings

Severity	Total	Open	Acknowledged	Partially Resolved	Resolved	Closed
Critical	1	-	-	-	1	-
High	7	-	2	-	5	-
Medium	4	-	1	-	3	-
Low	1	-	-	-	1	-
Total	13	-	3	-	10	-

#	Findings	Severity	Status
1	Quote Over Collateralization On Partially Filled Limit Buys	Critical	Resolved
2	Lack Of Protection Against Malicious Tokens	High	Resolved
3	Fee On Transfer Token Accounting Mismatch	High	Resolved
4	Fifo Priority Gaming Via Overpost Then Shrink	High	Resolved
5	Fee Collector Address Swap To Drain Accumulated Protocol Fees	High	Acknowledged
6	Orderbook Pointer Replacement To Strand User Liquidity	High	Acknowledged
7	Rebasing Token Accounting Break	High	Resolved
8	Order History View Denial Of Service	High	Resolved
9	Liquidity Provision Censorship Via Post Only Griefing	Medium	Acknowledged
10	Minquoteamount Bypass Via Fill First	Medium	Resolved
11	Taker Fee Rounding To Zero	Medium	Resolved
12	Extended Flows Inherit Token Risk	Medium	Resolved
13	Min Out Check Uses Gross Fill	Low	Resolved

Finding 1: Quote Over Collateralization On Partially Filled Limit Buys

Severity: CRITICAL
Status: RESOLVED

Description

When a buy limit order partially matches as taker at a better price and the remainder rests on the book, the contract pulls the full quote amount needed for the original order size but never reduces the locked quote down to only what the resting remainder requires. The refund path only executes if the remainder is zero; otherwise no adjustment is made, stranding surplus quote tokens in the contract that are never refunded and cannot be withdrawn.

The surplus equals the difference between the quote pulled for the original order size and the sum of the quote used for the partial fill plus the quote needed for the resting remainder. This surplus remains in the contract untracked, breaking the conservation invariant and causing permanent fund loss for users.

Source

OrderBook.sol - `_createOrder` function:

`contracts/OrderBook.sol:1392-1515` - Core logic for creating orders:

```
function _createOrder(uint256 baseAmount, uint256 price, bool postOnly, ExecutionType executionType,
    bool isBuy)
    internal
    returns (uint64 orderId, uint256 finalBaseFilled, uint256 finalQuoteFilled, uint256 takerFee)
{
    (bool requiresFillToSurvive, uint256 quoteAmountToLock) =
        _validateOrderInputInline(baseAmount, price, postOnly, isBuy);
    if (isBuy) {
        _handleBuyOrderCollateral(quoteAmountToLock);
    } else {
        _handleSellOrderCollateral(msg.sender, baseAmount);
    }

    uint256 baseAmountRemaining = baseAmount;
    finalBaseFilled = 0;
    finalQuoteFilled = 0;
    takerFee = 0;

    if (!postOnly) {
        (finalBaseFilled, finalQuoteFilled, takerFee) = _matchOrderEphemeral(msg.sender, isBuy, price,
 baseAmount);
        baseAmountRemaining = baseAmount > finalBaseFilled ? baseAmount - finalBaseFilled : 0;

        // ... FOK and IOC handling ...

        if (baseAmountRemaining == 0) {
            if (isBuy && quoteAmountToLock > finalQuoteFilled) {
                withdrawableQuote[msg.sender] += quoteAmountToLock - finalQuoteFilled;
            }
```

```
28                 return (0, finalBaseFilled, finalQuoteFilled, takerFee);
29             }
30
31             if (requiresFillToSurvive && finalBaseFilled == 0) {
32                 revert OrderBook__QuoteTooLow();
33             }
34         } else {
35             baseAmountRemaining = baseAmount;
36         }
37
38         // ... rest of order creation ...
39     }
```

`contracts/OrderBook.sol:2541-2558` - `_handleBuyOrderCollateral()` is called with the full `quoteAmountToLock` for the original `baseAmount`:

```
1   function _handleBuyOrderCollateral(uint256 quoteAmountToLock) internal {
2       uint256 availableQuote = withdrawableQuote[msg.sender];
3       if (availableQuote != 0) {
4           if (availableQuote >= quoteAmountToLock) {
5               unchecked {
6                   withdrawableQuote[msg.sender] = availableQuote - quoteAmountToLock;
7               }
8               return;
9           }
10
11          withdrawableQuote[msg.sender] = 0;
12          unchecked {
13              quoteAmountToLock -= availableQuote;
14          }
15      }
16
17      IERC20(address(QUOTE_TOKEN)).safeTransferFrom(msg.sender, address(this), quoteAmountToLock);
18  }
```

Impact

Permanent Fund Loss:

- User funds are permanently stranded in the contract (surplus never refunded)
- Per-user loss equals the surplus `S = baseFilled * (P_limit - P_exec)`, which can be significant on large orders with price improvement
- No workaround exists for users to recover stranded funds

Conservation Invariant Violation:

- Breaks core conservation invariant (Invariant 0) that all quote pulled should be accounted for
- Surplus quote accumulates in contract but is not tracked in any accounting structure
- Cancellation only refunds `calculateQuoteNeeded(remaining, price)`, not the surplus

Affected Operations:

- All partially filled buy limit orders that rest remainder on the book
- Occurs whenever a buy order gets price improvement (execution price better than limit price)
- Affects all users placing buy limit orders that partially fill, which is common in normal trading

Example Scenario:

- User places buy order: `baseAmount=10`, `limit=100`; best ask=90 with only 5 available
- Contract pulls 1000 quote for original size
- Order fills 5 base at execution price 90 (`finalQuoteFilled=450`)
- Remainder 5 base rests at limit price 100 (requires 500 quote locked)
- Surplus `S = 1000 - 450 - 500 = 50` quote stays in contract, untracked and unrecoverable

Remediation

Refund Surplus for Resting Orders:

Modify `_createOrder()` to calculate and refund surplus quote even when the order rests:

```
function _createOrder(...) internal returns (...) {
    // ... existing order creation logic ...

    if (isBuy && !postOnly) {
        _handleBuyOrderCollateral(quoteAmountToLock);
        (finalBaseFilled, finalQuoteFilled, takerFee) =
            _matchOrderEphemeral(msg.sender, isBuy, price, baseAmount);
        baseAmountRemaining = baseAmount > finalBaseFilled ? baseAmount - finalBaseFilled : 0;

        if (baseAmountRemaining > 0) {
            // Calculate quote needed for resting remainder
            uint256 quoteNeededForRemainder = calculateQuoteNeeded(baseAmountRemaining, price);

            // Calculate surplus: original quote - used quote - needed quote
            uint256 surplus = quoteAmountToLock > (finalQuoteFilled + quoteNeededForRemainder)
                ? quoteAmountToLock - finalQuoteFilled - quoteNeededForRemainder
                : 0;

            // Refund surplus immediately
            if (surplus > 0) {
                withdrawableQuote[msg.sender] += surplus;
            }
        } else {
            // Existing refund logic for fully filled orders
            if (quoteAmountToLock > finalQuoteFilled) {
                withdrawableQuote[msg.sender] += quoteAmountToLock - finalQuoteFilled;
            }
        }
    }

    // ... rest of function ...
}
```

Track Surplus in Order Structure:

Add a field to track surplus quote that can be refunded on cancellation:

```
struct Order {
    // ... existing fields ...
    uint256 surplusQuote; // Track surplus for later refund
}

// On cancellation, refund both remainder quote and surplus
```

POC

Proof of Concept Location:

```
test/poc/Quote_Over_Collateralization_on_Partially_Filled_Limit_Buys/
QuoteOverCollateralizationPoC.t.sol
```

Attack Scenario:

1. User places buy limit order: 10 base tokens at limit price 100 quote/base
2. Contract calculates and pulls `quoteAmountToLock = 1000` quote for full order
3. Best ask price is 90 quote/base with only 5 base available
4. Order partially fills: 5 base at execution price 90
    - `finalQuoteFilled = 450` quote used
    - `baseAmountRemaining = 5` base
5. Remainder 5 base rests at limit price 100
    - Quote needed for remainder: `calculateQuoteNeeded(5, 100) = 500` quote
6. Surplus calculation:
    - Original quote pulled: 1000
    - Quote used for fill: 450
    - Quote needed for remainder: 500
    - Surplus: `1000 - 450 - 500 = 50` quote
7. Result: 50 quote remains in contract, untracked and unrecoverable

Code Evidence:

The vulnerability is demonstrated by `_handleBuyOrderCollateral()` pulling the full `quoteAmountToLock` amount, `_matchOrderEphemeral()` executing partial fills at better prices, and the refund logic at lines 1463-1467

only executing when `baseAmountRemaining == 0`. When an order partially fills and rests, no refund occurs for the surplus quote, which equals `quoteAmountToLock - finalQuoteFilled - calculateQuoteNeeded(baseAmountRemaining, price)`.

How to Run:

```
cd <dbook-contracts-repo>
forge test --match-path test/poc/Quote_Over_Collateralization_on_Partially_Filled_Limit_Buys/QuoteOverCollateralizationPoC.t.sol -vv
```

Finding 2: Lack Of Protection Against Malicious Tokens

Severity: HIGH
Status: RESOLVED

Description

The Manager contract has no token whitelist or approval mechanism. Any ERC20-compatible address can be used to register an OrderBook without validation or review. This allows malicious tokens to be registered and used in the protocol, enabling arbitrary value extraction from OrderBook liquidity.

A malicious token contract can implement arbitrary logic in any function, including draining balances, upgrading to malicious implementations, minting unlimited supply inside transfer calls, or any other adversarial behavior. Programmatic detection of malicious code is challenging as adversarial contracts can hide malicious logic in unexpected locations.

The factory pattern amplifies this risk: with potentially hundreds of OrderBook deployments over the protocol lifetime, a single malicious token registration can result in significant value loss before detection and response. A token whitelist requiring explicit admin approval provides defense-in-depth by allowing human review of token contracts, team reputation, and market legitimacy before registration.

Primary Attack Vector - Unrestricted Mint:

The most straightforward exploitation occurs when a freely mintable token (e.g., PublicMintStandardToken) is registered. Once registered, any external user can mint unlimited tokens at zero cost and drain all scarce assets from makers, leaving them with worthless minted tokens. The Manager.registerOrderBook function validates bytecode hash matching, token addresses, decimals, and parameter consistency, but does not check whether tokens have unrestricted mint functions, access controls on minting, or any other semantic properties.

Source

Manager.sol - No Whitelist Exists:

`contracts/Manager.sol:70-140` - Manager state variables contain no token approval system:

```
// Manager.sol state (lines 70-140)

bytes32 public override expectedNormalizedBytecodeHash;
address[] public allOrderBooks;
mapping(bytes32 => address) public override orderBooks;
mapping(address => uint256) public override quoteTokenMinAmounts;

```

```
// No token whitelist
// No token approval mapping
// No token validation
```

`contracts/Manager.sol:187` - Constructor explicitly documents removal of whitelist:

```
/**

 * @notice Initializes the Manager contract, setting default fees and roles.
 * @dev ...
 * Quote token whitelist controls have been removed; only bytecode and basic parameter sanity checks apply.
 * ...
 */
constructor(address _defaultAdmin, address _feeCollector, uint256 _initialTakerFee) {
    // ...
}
```

`contracts/Manager.sol:404-500` - `registerOrderBook()` accepts any ERC20-compatible address:

```
function registerOrderBook(address baseToken, address quoteToken, address orderBookAddress)
    external
    payable
    virtual
    override
    nonReentrant
{
    bool isAdmin = _hasOrderBookRoleInternal(ADMIN_ROLE, msg.sender);
    bool isFactory = hasRole(FACTORY_ROLE, msg.sender);

    // Access control: only admin or trusted factory
    if (!isAdmin && !isFactory) {
        revert AccessControlUnauthorizedAccount(msg.sender, ADMIN_ROLE);
    }

    // Reject stray ETH to avoid trapping funds.
    if (msg.value != 0) revert Manager__UnexpectedPayment();

    // --- Input Validation ---
    if (baseToken == address(0) || quoteToken == address(0)) revert Manager__ZeroAddressToken();
    if (baseToken == quoteToken) revert Manager__TokensCannotBeSame();
    if (orderBookAddress == address(0)) revert Manager__InvalidOrderBookAddress();

    bytes32 pairHash = getPairHash(baseToken, quoteToken);
    // Only prevent duplicates for non-admin users (allows admin to register OrderBook v2)
    if (orderBooks[pairHash] != address(0) && !isAdmin) {
        revert Manager__PairAlreadyRegistered();
    }

    // 1. Bytecode Check
    // ... bytecode validation ...

    // 2. Parameter Checks
    IOrderBookCore candidateBook = IOrderBookCore(orderBookAddress);
    IOrderBook.Params memory params = candidateBook.getParams();
    if (address(params.baseToken) != baseToken) revert Manager__ParameterMismatch("BASE_TOKEN");
    if (address(params.quoteToken) != quoteToken) revert Manager__ParameterMismatch("QUOTE_TOKEN");
    // ... other parameter checks ...

    // 3. Store if all checks pass (BEFORE external calls to prevent reentrancy)
    // Update canonical pointer (keeps current semantics: latest wins)
    orderBooks[pairHash] = orderBookAddress;
    // ... registry updates ...

    // No token validation
    // Any address accepted
}
```

OrderBook.sol:

`contracts/OrderBook.sol:725-734 - marketBuy()` executes market buy orders without any token semantic validation:

```
function marketBuy(uint256 maxQuoteIn, uint256 minBaseOut)
    external
    virtual
    override
    nonReentrant
    whenNotPaused
    returns (uint256 baseFilled, uint256 quoteUsed, uint256 takerFee)
{
    return _marketBuyInternal(maxQuoteIn, minBaseOut);
}
```

`contracts/OrderBook.sol:2589-2608 - _collectQuoteCollateral()` collects quote collateral via `safeTransferFrom` without mintability checks:

```
function _collectQuoteCollateral(address trader, uint256 amountNeeded) internal {
    if (amountNeeded == 0) return;

    uint256 availableQuote = withdrawableQuote[trader];
    if (availableQuote != 0) {
        if (availableQuote >= amountNeeded) {
            unchecked {
                withdrawableQuote[trader] = availableQuote - amountNeeded;
            }
            return;
        }

        withdrawableQuote[trader] = 0;
        unchecked {
            amountNeeded -= availableQuote;
        }
    }

    IERC20(address(QUOTE_TOKEN)).safeTransferFrom(trader, address(this), amountNeeded);
}
```

Impact

Arbitrary Value Extraction:

Malicious tokens can drain OrderBook balances through any function. The attack surface includes all ERC20 functions (transfer, approve, transferFrom) and any custom logic implemented by the token contract. Detection requires manual contract review, which is impractical for every token registration.

Unrestricted Mint Attack Example:

When an OrderBook is registered with a malicious token having unrestricted mint functionality, attackers can mint unlimited tokens at zero cost and drain all valuable assets from the OrderBook. Legitimate makers receive worthless minted tokens instead of scarce assets, resulting in complete liquidity theft. Example: OrderBook with 150

WETH in sell orders can be drained for approximately $375,000 at current prices, with attack cost of only gas fees (approximately $50-100).

Factory Pattern Risk Amplification:

With hundreds of potential OrderBook deployments over the protocol lifetime, a single malicious token registration enables value extraction. The probability of malicious registration increases over time as more OrderBooks are deployed. Validation at registration time reduces the attack window and provides defense-in-depth.

No Economic Barrier:

Malicious token deployment cost is minimal (contract deployment gas only). Attack profitability scales with OrderBook liquidity. Attackers face no capital requirements beyond gas costs, making attacks infinitely profitable relative to gas expenses.

Total Value at Risk:

100% of orderbook liquidity on the affected token side. Attack requires no capital investment, and minted tokens have zero value. The attack scales with orderbook size as attackers can mint as much as needed.

Why Not CRITICAL:

- Requires specific malicious token registration (operational mistake or intentional attack)
- Scoped to individual OrderBook instance, not protocol-wide
- Can be mitigated with Manager-level token whitelist
- Factory pattern increases probability but does not guarantee exploitation

Why HIGH Severity:

- Complete value extraction possible via malicious token
- Factory pattern increases probability of malicious registration over time
- Current architecture lacks preventive controls
- Whitelist implementation is straightforward and industry-standard
- Real-world examples exist (freely mintable tokens, malicious ERC20 implementations)

Remediation

Implement Token Whitelist:

Add approval system requiring explicit validation:

```
// contracts/Manager.sol

mapping(address => bool) public approvedTokens;

event TokenApproved(address indexed token);
event TokenRevoked(address indexed token);

function approveToken(address token) external onlyRole(ADMIN_ROLE) {
    require(!approvedTokens[token], "Already approved");
    approvedTokens[token] = true;
    emit TokenApproved(token);
}

function revokeToken(address token) external onlyRole(ADMIN_ROLE) {
    require(approvedTokens[token], "Not approved");
    approvedTokens[token] = false;
    emit TokenRevoked(token);
}

function registerOrderBook(address baseToken, address quoteToken, address orderBookAddress)
    external
    payable
    virtual
    override
    nonReentrant
{
    require(approvedTokens[baseToken], "Base token not approved");
    require(approvedTokens[quoteToken], "Quote token not approved");

    // ... existing validation ...
}
```

Token Approval Process:

Required checks before approval:

1. Verify token address matches canonical source (CoinGecko, official documentation)
2. Review contract code for malicious logic (mint, upgrade patterns, hidden drains)
3. Confirm token has established market legitimacy and DEX liquidity
4. Verify token team reputation and audit history
5. Document approval decision and identified risks

Reject criteria:

- Unverified or obfuscated source code
- Upgradeable proxy without transparent governance

- Unrestricted mint/burn functions
- Recently deployed without audit history
- Low liquidity or market cap

Mintability Detection:

Add checks during registration to detect unrestricted mint functions:

```
function hasUnrestrictedMint(address token) internal view returns (bool) {
    // Check if token has public mint function without access control
    // Attempt to detect mint function signature and access modifiers
    // Reject tokens with unrestricted minting
}
```

Operational Policy:

Document that only verified tokens with proper access controls should be registered. UIs and factories should restrict selectable tokens to curated lists of approved tokens.

POC

Proof of Concept Location:

```
test/poc/Freely_Mintable_Token_Liquidity_Drainage/
FreelyMintableTokenDrainagePoC.t.sol
```

Attack Scenario - Unrestricted Mint:

1. OrderBook registered with PublicMintStandardToken as QUOTE_TOKEN and WETH as BASE_TOKEN
2. Makers place sell orders: 150 WETH total at price 2000 PMT/WETH
3. Attacker mints 1,000,000 PublicMintStandardToken tokens (zero cost, unlimited supply)
4. Attacker approves OrderBook to spend PublicMintStandardToken
5. Attacker calls marketBuy(maxQuoteIn=330,000 PMT, minBaseOut=0) to sweep all WETH
6. Result:
    - Attacker receives 149 WETH (150 WETH minus taker fee)
    - Contract holds 300,000 PMT (minted tokens that cost nothing)
    - Makers claim their proceeds: 300,000 worthless PublicMintStandardToken tokens
    - Total loss to makers: 150 WETH (approximately $375,000 at $2,500/WETH)
    - Attacker cost: Only gas fees (approximately $50-100)

Test Cases:

1. **`test_ManagerAllowsMintableTokenRegistration()`**: Demonstrates that Manager.registerOrderBook() successfully registers an OrderBook with a freely mintable token (PublicMintStandardToken) without any mintability checks. The test verifies that the OrderBook is registered and that the quote token has an unrestricted mint function.

2. **`test_AttackerDrainsAllLiquidityWithMintedTokens()`**: This is the core attack demonstration. The test shows:

   - Makers place sell orders: 100 WETH from maker1 and 50 WETH from maker2 at price 2000 PMT/WETH
   - Attacker mints 1,000,000 PublicMintStandardToken tokens (zero cost, unlimited supply)
   - Attacker calls marketBuy() to sweep all 150 WETH from makers
   - Result: Attacker receives 149 WETH (minus taker fee), contract holds 300,000 PMT (minted tokens), makers claim worthless PMT tokens
   - Total loss to makers: 150 WETH
   - Attacker cost: Only gas fees

3. **`test_ReverseAttack_MintableBaseToken()`**: Demonstrates the reverse attack scenario where a mintable token is used as BASE_TOKEN instead of QUOTE_TOKEN, showing the vulnerability works in both directions.

4. **`test_ManagerNoMintabilityChecks()`**: Analyzes the Manager.registerOrderBook() function to show what validations are performed (bytecode normalization, parameter consistency) and what is missing (mintability checks, token whitelist).

Code Evidence:

The vulnerability is demonstrated by Manager.registerOrderBook() having no token whitelist or approval mechanism, accepting any ERC20-compatible address without validation. The constructor explicitly documents that whitelist controls have been removed. PublicMintStandardToken.mint() is an unrestricted external function with no access control, and Manager.registerOrderBook() has no mintability checks, only validating bytecode and parameters. OrderBook.marketBuy() accepts any ERC20 token without semantic validation. The attack path shows that minted tokens satisfy transferFrom requirements, allowing attackers to acquire scarce assets using zero-value tokens.

How to Run:

```
cd <dbook-contracts-repo>
forge test --match-path test/poc/Freely_Mintable_Token_Liquidity_Drainage/
    FreelyMintableTokenDrainagePoC.t.sol -vvv
```

Discussion

Developer: Permissionless listings are NOT supported. Listings will be done by team (initially) and governance (later). Non-standard tokens will not be listed. Implemented token whitelisting in Manager.sol contract.

Auditor: Resolved. Token whitelist was properly implemented in commit `d682a831`, which enforces that both base and quote tokens must be explicitly whitelisted before OrderBook registration. The implementation includes `whitelistToken()` and `revokeToken()` functions with proper ADMIN_ROLE access control, enforcement check in `registerOrderBook()`, and appropriate events. This matches the recommended remediation exactly.

Finding 3: Fee On Transfer Token Accounting Mismatch

Severity: HIGH
Status: RESOLVED

Description

The OrderBook contract assumes exact ERC20 transfer semantics throughout its collateral handling and matching flows. When a fee-on-transfer token is used as either BASE_TOKEN or QUOTE_TOKEN, the contract receives less than the nominal transfer amount, but accounting proceeds as if the full amount was received. This creates an immediate discrepancy between recorded balances and actual token holdings, causing protocol insolvency and enabling systematic value extraction.

The vulnerability manifests in two primary attack vectors:

1. **BASE_TOKEN as fee-on-transfer (sell-side insolvency)**: When users place sell orders with a fee-on-transfer base token, the contract receives less than the nominal amount transferred, but orders are recorded at full nominal value. When matched, buyers receive credits for the full amount, causing the contract to become insolvent on the base side.

2. **QUOTE_TOKEN as fee-on-transfer (buy-side insolvency)**: When users place buy orders or execute market buys with a fee-on-transfer quote token, the contract receives less quote than transferred, but matching logic credits sellers with full nominal amounts, causing quote-side insolvency.

The core issue is that SafeTokenTransfer.safeTransferFrom only validates that the transfer call succeeded and returned true, but never compares pre/post transfer balances. The OrderBook then uses nominal amounts throughout all accounting operations without any balance reconciliation.

Why This Is High Severity Despite "We Won't Use FOT Tokens" Rebuttal:

This vulnerability cannot be dismissed with a simple "we won't use FOT tokens" policy due to three critical risk factors:

1. Factory Pattern Operational Risk:

The protocol uses an OrderBookFactory pattern that allows deployment of multiple OrderBook instances. With potentially hundreds of OrderBooks deployed across different trading pairs, operational mistakes become likely rather than theoretical:

- Human error during deployment (e.g., deploying with USDT thinking it's "safe")

- No centralized review of every factory deployment
- One mistake = entire OrderBook becomes insolvent
- Attack surface is multiplicative across all deployments

2. Token Upgrade Threat - Real-World Examples:

Many widely-used tokens have upgrade mechanisms that can add FOT behavior post-deployment:

USDT (Tether) - ACTIVE RISK:

- USDT contract contains `setParams(uint newBasisPoints, uint newMaxFee)` function controlled by owner
- Currently `basisPointsRate = 0` (no fee), but can be changed at ANY time via admin call
- If enabled, all USDT OrderBooks would become instantly insolvent

PAXG (Pax Gold) - CONFIRMED CASE:

- Initially deployed with no transfer fees
- After contract upgrade: Added optional fees (currently ~0.02% for certain scenarios)
- Demonstrates this threat model is real and has occurred in production

Proxy-Upgradeable Tokens - BROAD RISK: Most modern tokens use upgradeable proxy patterns (UUPS, Transparent Proxy, Beacon Proxy):

- USDC (Centre Consortium) - Proxy upgradeable
- TUSD - Proxy upgradeable

- WBTC - Multisig controlled, can upgrade
- Many DeFi governance tokens (UNI, COMP, MKR-like) - Can vote to add fees

Governance-Controlled Tokens: DAO-governed tokens can vote to:

- Add trading fees to fund treasury
- Implement “burn-on-transfer” mechanisms
- Add liquidity provision fees

Real-World Precedent:

- SAFEMOON: Started with 10% fee, changed via contract updates
- Various DeFi 2.0 tokens: Added dynamic fee mechanisms post-launch

3. Defense-in-Depth Engineering:

Even if the protocol intends to avoid FOT tokens, implementing balance-delta verification provides:

- Protection against operational mistakes (factory pattern)
- Protection against future token upgrades (USDT, USDC, PAXG risk)
- Robust accounting practices (defensive programming)
- Minimal gas overhead (~5,000 gas per transfer, ~3-5% increase)

Summary of Upgrade Risk:

Token	Upgrade Mechanism	FOT Status	Risk Level
USDT	Admin function (setParams)	Fee mechanism exists, rate=0	**HIGH**
USDC	Proxy upgradeable (Centre Consortium)	No fee currently	**MEDIUM**
PAXG	Upgradeable (verified upgrade history)	~0.02% fee added	**CONFIRMED**
TUSD	Proxy upgradeable	No fee currently	**MEDIUM**
WBTC	Multisig controlled	No fee currently	**MEDIUM**

Source

OrderBook.sol - Collateral Handling:

`contracts/OrderBook.sol:2564-2582 - _handleSellOrderCollateral()` executes `safeTransferFrom` for full `baseAmount` without checking actual received balance:

```
function _handleSellOrderCollateral(address trader, uint256 baseAmount) internal {
    // First consume any withdrawable base balance for the user
    uint256 availableBase = withdrawableBase[trader];
    if (availableBase != 0) {
        if (availableBase >= baseAmount) {
            unchecked {
                withdrawableBase[trader] = availableBase - baseAmount;
            }
            return;
        }

        withdrawableBase[trader] = 0;
        unchecked {
```

```
            baseAmount -= availableBase;
        }
    }

    IERC20(address(BASE_TOKEN)).safeTransferFrom(trader, address(this), baseAmount);
}
```

`contracts/OrderBook.sol:2541-2558` - `_handleBuyOrderCollateral()` transfers `quoteAmountToLock` without balance reconciliation:

```
function _handleBuyOrderCollateral(uint256 quoteAmountToLock) internal {
    uint256 availableQuote = withdrawableQuote[msg.sender];
    if (availableQuote != 0) {
        if (availableQuote >= quoteAmountToLock) {
            unchecked {
                withdrawableQuote[msg.sender] = availableQuote - quoteAmountToLock;
            }
            return;
        }

        withdrawableQuote[msg.sender] = 0;
        unchecked {
            quoteAmountToLock -= availableQuote;
        }
    }

    IERC20(address(QUOTE_TOKEN)).safeTransferFrom(msg.sender, address(this), quoteAmountToLock);
}
```

`contracts/OrderBook.sol:2589-2608` - `_collectQuoteCollateral()` collects quote collateral without verifying actual received amount:

```
function _collectQuoteCollateral(address trader, uint256 amountNeeded) internal {
    if (amountNeeded == 0) return;

    uint256 availableQuote = withdrawableQuote[trader];
    if (availableQuote != 0) {
        if (availableQuote >= amountNeeded) {
            unchecked {
                withdrawableQuote[trader] = availableQuote - amountNeeded;
            }
            return;
        }

        withdrawableQuote[trader] = 0;
        unchecked {
            amountNeeded -= availableQuote;
        }
    }

    IERC20(address(QUOTE_TOKEN)).safeTransferFrom(trader, address(this), amountNeeded);
}
```

OrderBook.sol - Matching and Accounting:

`contracts/OrderBook.sol:1671-1779` - `_matchAtPriceLevel()` uses nominal `baseFilledThisLevel` and computed `quoteValue` for all accounting updates without checking actual token balances:

```
function _matchAtPriceLevel(
    address takerAddress,
    uint256 priceLevel,
    uint256 maxTakerAmountToFill,
```

```
    bool isTakerBuy,
    uint256 takerFeeRate,
    MatchAccumulator memory acc
) internal returns (uint256 baseFilledThisLevel, uint256 remainingVolumeAfter) {
    // ... volume calculation ...

    uint256 quoteValue = Math.mulDiv(baseFilledThisLevel, priceQ, BASE_PRICE_PRECISION_DENOM);

    // Update accumulator
    unchecked {
        acc.baseFilled += baseFilledThisLevel;
        acc.quoteFilled += quoteValue;
    }

    // ... fee calculation and volume updates ...
}
```

`contracts/OrderBook.sol:1075-1083 - calculateQuoteNeeded()` computes required quote amounts using nominal `baseAmount` without consideration for transfer fees:

```
function calculateQuoteNeeded(uint256 baseAmount, uint256 price)
    public
    view
    override
    returns (uint256 quoteAmount)
{
    if (price == 0 || baseAmount == 0) return 0;
    quoteAmount = Math.mulDiv(baseAmount, price * QUOTE_SCALE, BASE_PRICE_PRECISION_DENOM);
}
```

`contracts/OrderBook.sol:1625-1657 - _applyMatchResults()` credits takers and makers based on accumulated nominal fills:

```
function _applyMatchResults(address takerAddress, bool isTakerBuy, MatchAccumulator memory acc)
    internal
    returns (uint256 takerFeePaid)
{
    if (acc.baseFilled == 0) return 0;

    takerFeePaid = isTakerBuy ? acc.takerBaseFee : acc.takerQuoteFee;

    uint256 wBase = acc.wBase;
    if (wBase > 0) {
        withdrawableBase[takerAddress] += wBase;
    }

    uint256 wQuote = acc.wQuote;
    if (wQuote > 0) {
        withdrawableQuote[takerAddress] += wQuote;
    }

    // ... fee accumulation ...
}
```

SafeTokenTransfer.sol:

`contracts/libraries/SafeTokenTransfer.sol:44-63 - safeTransferFrom()` only validates that transfer call succeeded and optionally returned true. Does not compare balances before/after to detect fee-on-transfer behavior:

```
function safeTransferFrom(IERC20 token, address from, address to, uint256 amount) internal {
```

```
    if (amount == 0) return;

    uint256 success;
    assembly {
        let ptr := mload(0x40)
        mstore(ptr, 0x23b872dd00000000000000000000000000000000000000000000000000000000)
        mstore(add(ptr, 4), from)
        mstore(add(ptr, 36), to)
        mstore(add(ptr, 68), amount)

        success := call(gas(), token, 0, ptr, 100, 0, 32)

        if gt(returndatasize(), 0) {
            success := and(success, iszero(iszero(mload(0x00))))
        }
    }

    if (success == 0) revert SafeTokenTransfer__TransferFromFailed();
}
```

Manager.sol:

`contracts/Manager.sol:404-500` - `registerOrderBook()` validates bytecode normalization and parameter consistency but does not screen for fee-on-transfer semantics or enforce any transfer behavior checks:

```
function registerOrderBook(address baseToken, address quoteToken, address orderBookAddress)
    external
    payable
    virtual
    override
    nonReentrant
{
    // ... access control and input validation ...

    // 1. Bytecode Check
    // ... bytecode validation ...

    // 2. Parameter Checks
    IOrderBookCore candidateBook = IOrderBookCore(orderBookAddress);
    IOrderBook.Params memory params = candidateBook.getParams();
    if (address(params.baseToken) != baseToken) revert Manager__ParameterMismatch("BASE_TOKEN");
    if (address(params.quoteToken) != quoteToken) revert Manager__ParameterMismatch("QUOTE_TOKEN");
    // ... other parameter checks ...

    // No fee-on-transfer detection or validation
}
```

Impact

Immediate Protocol Insolvency:

- When a fee-on-transfer token is used, the contract becomes insolvent after the first trade
- Total withdrawable balances exceed actual token holdings
- Withdrawal DoS for legitimate makers as later withdrawals revert due to insufficient balance

Systematic Value Extraction:

- Attackers can place orders with fee-on-transfer tokens, match against their own orders, and withdraw more tokens than the contract actually holds
- With a 2% fee-on-transfer token and 0.3% taker fee, attackers can extract approximately 1.7% profit per cycle
- Attack scales linearly with volume and is repeatable until liquidity depletes
- Minimal capital required as attackers can recycle withdrawable balances

Total Value at Risk:

- 100% of resting liquidity on the affected token side
- All makers with pending claims on the affected side lose access to their funds
- Protocol becomes unusable for the affected trading pair

Why Not CRITICAL:

- Requires specific token choice (operational mistake) OR future token upgrade (external dependency)
- Scoped to individual OrderBook instance, not protocol-wide
- Can be mitigated with Manager-level validation and documentation
- Industry practice: Most AMMs have similar limitations

Why HIGH Severity:

- Factory pattern makes operational mistakes likely across multiple deployments
- USDT upgrade risk is real and impacts many potential deployments
- Results in complete insolvency (100% loss on that OrderBook)
- Affects legitimate, active OrderBooks (not just new deployments)
- Real-world precedent: PAXG confirmed case of fees added post-deployment

Remediation

Implement Balance-Delta Verification:

Modify SafeTokenTransfer.safeTransferFrom to compare balances before and after transfer:

```
function safeTransferFrom(
    address token,
    address from,
    address to,
    uint256 amount
) internal {
    uint256 balanceBefore = IERC20(token).balanceOf(to);
    (bool success, bytes memory data) = token.call(
        abi.encodeWithSelector(IERC20.transferFrom.selector, from, to, amount)
    );
    require(success && (data.length == 0 || abi.decode(data, (bool))), "SafeTokenTransfer: transfer
    failed");

    uint256 balanceAfter = IERC20(token).balanceOf(to);
    require(balanceAfter >= balanceBefore + amount, "SafeTokenTransfer: fee-on-transfer detected");
}
```

Token Semantic Validation:

Add fee-on-transfer detection during Manager.registerOrderBook:

```
function registerOrderBook(address baseToken, address quoteToken, address orderBookAddress)
    external payable virtual override nonReentrant
{
    // ... existing validation ...

    // Test transfer to detect fee-on-transfer
    uint256 testAmount = 1000 * 10**IERC20Metadata(baseToken).decimals();
    uint256 balanceBefore = IERC20(baseToken).balanceOf(address(this));
    IERC20(baseToken).transferFrom(msg.sender, address(this), testAmount);
    uint256 balanceAfter = IERC20(baseToken).balanceOf(address(this));
    require(balanceAfter == balanceBefore + testAmount, "Manager: fee-on-transfer token detected");

    // Repeat for quoteToken
    // ...
}
```

Alternative: Token Whitelist:

Maintain a Manager-level whitelist of approved tokens with verified transfer semantics. Only allow registration of tokens that have been verified to use exact-transfer semantics.

Gas Cost Analysis:

Balance-delta verification adds approximately 5,000 gas per transfer (~3-5% overhead):

- `balanceOf()` SLOAD: ~2,100 gas (warm) / 2,600 gas (cold)
- Two calls per transfer = ~4,200-5,200 gas total
- At 30 gwei: ~$0.02 per trade additional cost
- **Verdict: Gas cost is acceptable** for protection against operational mistakes and token upgrades

POC

Proof of Concept Location:

```
test/poc/Fee_On_Transfer_Token_Accounting_Mismatch/
FeeOnTransferPoC.t.sol
```

Test Cases:

1. **`test_AccountingMismatchOnFotTransfer()`**: Demonstrates that when a fee-on-transfer token is used, the contract receives less tokens than recorded. A 2% FOT token transfer of 1000 tokens results in only 980 tokens received, creating an immediate 20 token deficit.

2. **`test_ValueExtractionViaMarketBuy()`**: Shows an attacker can extract value by executing marketBuy with a FOT quote token. The contract records the full quoteUsed amount but actually receives less due to the transfer fee, enabling the attacker to acquire base tokens while paying less quote than accounted.

3. **`test_WithdrawalFailureDueToInsolvency()`**: Demonstrates that after multiple trades with FOT tokens, the contract becomes insolvent. Total withdrawable balances exceed actual token holdings, causing withdrawal failures for legitimate makers.

4. **`test_ValueExtractionViaCreateBuyOrder()`**: Shows the same vulnerability exists in limit orders. When a buy order is placed with a FOT quote token, the contract locks the full nominal amount but receives less, creating an accounting mismatch.

5. **`test_AttackViaCreateBuyOrder()`**: Additional test demonstrating the attack via limit orders, showing the same vulnerability exists in both market and limit order flows.

6. **`test_CumulativeDeficitCompounds()`**: Demonstrates how the deficit compounds across multiple trades, showing that the vulnerability scales with trading volume.

Attack Flow:

1. OrderBook is deployed with a fee-on-transfer token as QUOTE_TOKEN (2% fee)
2. Maker places sell order: 100 BASE at price 100 QUOTE/BASE
3. Attacker executes marketBuy to buy 100 BASE
    - Contract computes quoteUsed = 10,000 QUOTE
    - Contract calls _collectQuoteCollateral(10,000 QUOTE)
    - FOT token transfers only 9,800 QUOTE (2% fee = 200 QUOTE deducted)
    - Contract records 10,000 QUOTE received

- Attacker receives approximately 100 BASE (minus taker fee)

4. Result: Attacker paid 9,800 QUOTE but contract recorded 10,000 QUOTE, creating a 200 QUOTE deficit per trade. The contract becomes insolvent after multiple trades, causing withdrawal failures.

How to Run:

```
cd <dbook-contracts-repo>
forge test --match-path test/poc/Fee_On_Transfer_Token_Accounting_Mismatch/FeeOnTransferPoC.t.sol -vvv
```

Finding 4: Fifo Priority Gaming Via Overpost Then Shrink

Severity: HIGH
Status: RESOLVED

Description

Attackers can place large orders to capture early FIFO priority positions (low `claimRangeStart` values), then reduce order size via `decreaseOrderSize()`, maintaining their priority position while freeing up most of their locked capital. This enables unfair advantage over honest makers at the same price level, as the attacker's reduced-size order will fill before larger orders placed later.

The vulnerability exists because `claimRangeStart` is assigned at order placement time based on the current frontier and is immutable thereafter, while `decreaseOrderSize()` only modifies `claimRangeEnd` and `order.amount`, preserving the original priority position. Additionally, the `minQuoteAmount` validation check only applies to orders that have not yet received any fills, allowing filled orders to be reduced below the minimum threshold.

Source

OrderBook.sol:

`contracts/OrderBook.sol:1799-1911` - `_addOrderToBook()` assigns `claimRangeStart` based on current queue tail, establishing FIFO priority:

```
function _addOrderToBook(uint64 orderId, uint256 price, uint256 baseAmountRemaining, bool isBuy)
    internal {
    // ... order creation event ...

    // === CLAIM RANGE ASSIGNMENT (O(1) GAS OPTIMIZATION) ===
    // Assign this order its position in the global frontier coordinate system.
    IOrderBook.QueuePosition storage queuePos = isBuy ? buyQueuePos[price] : sellQueuePos[price];
    uint128 tail = queuePos.position;
    uint256 frontierBase = queuePos.reserved;
    uint256 localFrontier = cumulative + cancelPrefix;
    localFrontier = _roundDownToQueue(localFrontier);
    uint256 frontier = frontierBase + localFrontier;

    uint256 start256 = frontier > tail ? frontier : uint256(tail);
    uint256 end256 = start256 + baseAmountRemaining;

    uint128 claimRangeStart = uint128(start256);
    uint128 claimRangeEnd = uint128(end256);

    // Packed writes: slot1 (amount + claimRangeStart) and slot2 (claimRangeEnd + isBuy + status +
    cancelIndex)
    _setOrderSlot1Packed(newOrder, claimRangeStart, uint128(baseAmountRemaining));
    _setOrderSlot2Packed(newOrder, claimRangeEnd, isBuy, IOrderBook.OrderStatus.Active, idx, gen);
}
```

OrderBookExtended.sol:

`contracts/OrderBookExtended.sol:97-163 - _decreaseOrderSizeInternal()` reduces `claimRangeEnd` and `order.amount` but preserves `claimRangeStart`, maintaining priority position:

```
function _decreaseOrderSizeInternal(uint64 orderId, uint256 reduceAmount, address trader)
    internal
    returns (uint256 newTotalSize)
{
    // ... validation ...

    bool hasSeenFill = totalFilled > 0;
    uint256 newTotalOrderSize = totalOrderSize - reduceAmount;
    if (minQuoteAmount > 0 && !hasSeenFill) {
        uint256 newQuoteValue = calculateQuoteNeeded(newTotalOrderSize, price);
        if (newQuoteValue < minQuoteAmount) {
            revert OrderBook__QuoteTooLow();
        }
    }

    _recordCancelledRemainder(order, orderId, price, isBuyOrder, claimRangeEnd, reduceAmount);

    uint256 newClaimRangeEnd = claimRangeEnd - reduceAmount;
    order.claimRangeEnd = uint128(newClaimRangeEnd);

    uint256 newAmount = amountBefore - reduceAmount;
    order.amount = uint128(newAmount);

    // Note: claimRangeStart is NOT modified, preserving original priority position
}
```

Impact

Unfair Priority Gaming:

- Legitimate makers are disadvantaged at affected price levels
- Attackers can maintain priority with minimal capital after initial placement
- Undermines FIFO fairness, which is core to the exchange value proposition

Material Maker Revenue Loss:

- Honest makers lose expected fill volume to priority-gaming attackers
- If widely adopted, creates a priority manipulation race where all makers must engage in the same behavior
- Significant revenue impact at popular price levels with high trading volume

Capital Efficiency Exploitation:

- Attackers achieve approximately 100x capital efficiency (1000 BASE briefly locked, reduced to 10 BASE for same priority)

- Attack is repeatable continuously with minimal capital
- No special permissions required, any unprivileged user can execute

Remediation

Reset Priority on Size Reduction:

Modify `decreaseOrderSize()` to recalculate `claimRangeStart` when order size is reduced below a threshold, or invalidate the priority position entirely:

```
function decreaseOrderSize(uint64 orderId, uint256 reduceAmount) external {
    // ... existing validation ...

    // If order size reduction is significant, reset priority
    uint256 newSize = order.amount - reduceAmount;
    if (newSize < order.amount / 2) { // Threshold check
        // Recalculate claimRangeStart based on current frontier
        // This removes the unfair priority advantage
    }

    // ... rest of function ...
}
```

Enforce Minimum Size for Priority Preservation:

Require that orders maintain a minimum size relative to their original size to preserve priority. If reduced below this threshold, recalculate priority:

```
uint256 minSizeForPriority = (order.originalAmount * MIN_PRIORITY_RATIO) / 100;
if (newSize < minSizeForPriority) {
    // Reset claimRangeStart to current frontier
}
```

Apply minQuoteAmount to All Size Reductions:

Remove the `!hasSeenFill` condition from the `minQuoteAmount` check in `decreaseOrderSize()`, ensuring all orders maintain minimum economic significance:

```
// Remove: if (!hasSeenFill) { ... }
// Always check minQuoteAmount regardless of fill status
```

POC

Proof of Concept Location:

```
test/poc/FIFO_Priority_Gaming_via_Overpost_Then_Shrink/
FIFOPriorityGamingPoC.t.sol
```

Attack Scenario (Using Testnet Configuration):

Testnet Configuration (Verified):

- Contract Address: `0xf73C123fc8392c84A68f5F048d64e8bC9CB8Cf4c`
- Network: MegaETH Testnet
- `minQuoteAmount`: 10,000,000 (10 USDC)

Attack Execution:

1. Attacker observes competitive price level with spread opportunity
2. Attacker places 1000 BASE order at price P (must meet `minQuoteAmount = $10 USDC`), securing `claimRangeStart=1000` (assuming frontier=1000)
3. Honest maker places 500 BASE order at same price, receiving `claimRangeStart=2000`
4. Attacker receives a partial fill (any amount > 0), setting `hasSeenFill=true`
5. Attacker calls `decreaseOrderSize(orderId, reduceAmount)`, reducing order to minimal size (targets 10 BASE, but may be ~10-20 BASE due to QUEUE_UNIT_SIZE rounding)
    - Because `hasSeenFill=true`, the `minQuoteAmount` check is bypassed (lines 126-131)
    - Order can shrink below $10 USDC threshold
6. Contract updates `claimRangeEnd` but preserves `claimRangeStart` (priority maintained)
7. Attacker receives refund of approximately 98% of locked collateral while maintaining priority
8. When taker executes marketBuy for 400 BASE, attacker's reduced order (10-20 BASE) fills entirely before honest maker's 500 BASE

Economic Analysis:

- Initial capital: Must meet `minQuoteAmount = $10 USDC` for initial order placement
- After partial fill: Can reduce to minimal size (below $10 USDC) while maintaining priority
- Capital efficiency: Approximately 50x (1000 BASE briefly locked, reduced to ~20 BASE for same priority, accounting for QUEUE_UNIT_SIZE rounding)
- Attack cost: Gas fees only (capital is recoverable via size reduction)
- Benefit: Priority position maintained with minimal locked capital

Code Evidence:

The vulnerability is demonstrated by `_addOrderToBook()` assigning `claimRangeStart` at placement time based on the current frontier, `decreaseOrderSize()` preserving this value while reducing order size, and

`_matchAtPriceLevel()` filling orders strictly in `claimRangeStart` order. The `minQuoteAmount` bypass for filled orders (lines 126-131 in OrderBookExtended.sol) makes this attack easier to exploit by allowing orders to shrink below the $10 USDC minimum threshold after receiving any fill, as verified on the testnet deployment.

How to Run:

```
cd <dbook-contracts-repo>
forge test --match-path test/poc/FIFO_Priority_Gaming_via_Overpost_Then_Shrink/FIFOPriorityGamingPoC.t.sol -vv
```

Discussion

Developer: Implemented unconditional minQuoteAmount enforcement in commit `621e3c6c`. Orders can no longer shrink below economic minimums regardless of fill status. The behavior of preserving time priority on size reduction follows standard price-time priority conventions used by major CLOBs. Reducing order size does not reset queue position because the trader is not improving their price or gaining an execution advantage beyond what their original arrival time justified. Recalculating priority on size reduction would break alignment with standard CLOB behavior.

Auditor: Resolved. The main exploit vector (shrinking to dust after partial fill by bypassing minQuoteAmount) is fixed via unconditional enforcement in commit `621e3c6c`. The remaining behavior (size reduction preserves priority) is a legitimate design decision consistent with many traditional CLOBs. The capital efficiency optimization is now bounded by minQuoteAmount.

Finding 5: Fee Collector Address Swap To Drain Accumulated Protocol Fees

Severity: HIGH **Status:** ACKNOWLEDGED

Description

The Manager contract allows ADMIN_ROLE to atomically change the fee collector address without any timelock or delay mechanism. Combined with the OrderBook's dynamic fee collector lookup at claim time, this enables a compromised or malicious ADMIN to redirect all accumulated protocol fees to an attacker-controlled address in a single transaction.

The attack flow is straightforward:

1. ADMIN calls `Manager.setFeeCollector(attackerAddress)` to immediately update the fee collector
2. ADMIN calls `OrderBook.claimFees()` on all registered OrderBooks
3. All accumulated `pendingBaseFees` and `pendingQuoteFees` are transferred to the attacker address

No timelock or recovery mechanism exists to prevent this atomic swap and drain attack. The attack can extract weeks or months of accumulated protocol revenue, representing a significant portion of annual protocol income.

Source

Manager.setFeeCollector:

`contracts/Manager.sol:377-384` - Function protected only by `onlyAdminRole` modifier with no timelock. Immediately updates `feeCollector` storage variable upon execution:

```
function setFeeCollector(address _feeCollector) external override onlyAdminRole {
    if (_feeCollector == address(0)) revert Manager__InvalidFeeCollector();
    if (_feeCollector != feeCollector) {
        address oldFeeCollector = feeCollector;
        feeCollector = _feeCollector;
        emit FeeCollectorUpdated(oldFeeCollector, _feeCollector);
    }
}
```

OrderBook.claimFees:

`contracts/OrderBook.sol:981-1011` - Function restricted to `ADMIN_ROLE` via `onlyManagerRole(ADMIN_ROLE)`. Reads `feeCollector` dynamically from `manager.feeCollector()` at claim time, not cached at fee accrual:

```
function claimFees()
    external
    override
    nonReentrant
    onlyManagerRole(ADMIN_ROLE)
    returns (bool baseSuccess, bool quoteSuccess)
{
    address feeCollector = manager.feeCollector();

    bytes32 pendingBaseSlot;
    bytes32 pendingQuoteSlot;
    assembly {
        pendingBaseSlot := pendingBaseFees.slot
        pendingQuoteSlot := pendingQuoteFees.slot
    }

    uint256 baseToClaim;
    uint256 quoteToClaim;
    (baseSuccess, quoteSuccess, baseToClaim, quoteToClaim) = OrderBookAux.claimFees(
        pendingBaseFees,
        pendingQuoteFees,
        address(BASE_TOKEN),
        address(QUOTE_TOKEN),
        feeCollector,
        pendingBaseSlot,
        pendingQuoteSlot
    );

    emit FeesClaimed(feeCollector, baseToClaim, quoteToClaim, baseSuccess, quoteSuccess);
    return (baseSuccess, quoteSuccess);
}
```

OrderBookAux.claimFees:

`contracts/libraries/OrderBookAux.sol:227-280` - Library function performs optimistic storage updates (zeroing pending fees) then attempts token transfers to the provided `feeCollector` address. If transfers succeed, fees are swept:

```
function claimFees(
    uint256 pendingBaseFeesValue,
    uint256 pendingQuoteFeesValue,
    address baseToken,
    address quoteToken,
    address feeCollector,
    bytes32 pendingBaseFeesSlot,
    bytes32 pendingQuoteFeesSlot
)
    external
    returns (
        bool baseSuccess,
        bool quoteSuccess,
        uint256 baseToClaim,
        uint256 quoteToClaim
    )
{
    uint256 basePending = pendingBaseFeesValue;
    uint256 quotePending = pendingQuoteFeesValue;

    baseToClaim = basePending;
    quoteToClaim = quotePending;
    if (baseToClaim == 0 && quoteToClaim == 0) revert IOrderBook.OrderBook__NoFeesToClaim();

    if (baseToClaim > 0) {
        uint256 updatedBase = basePending - baseToClaim;
        assembly {
```

```
28                 sstore(pendingBaseFeesSlot, updatedBase)
29             }
30             baseSuccess = _trySafeTransfer(baseToken, feeCollector, baseToClaim);
31             if (!baseSuccess) {
32                 assembly {
33                     sstore(pendingBaseFeesSlot, basePending)
34                 }
35             }
36         } else {
37             baseSuccess = true;
38         }
39
40         if (quoteToClaim > 0) {
41             uint256 updatedQuote = quotePending - quoteToClaim;
42             assembly {
43                 sstore(pendingQuoteFeesSlot, updatedQuote)
44             }
45             quoteSuccess = _trySafeTransfer(quoteToken, feeCollector, quoteToClaim);
46             if (!quoteSuccess) {
47                 assembly {
48                     sstore(pendingQuoteFeesSlot, quotePending)
49                 }
50             }
51         } else {
52             quoteSuccess = true;
53         }
54     }
```

Impact

Direct Fund Loss:

- All accumulated `pendingBaseFees` and `pendingQuoteFees` across all registered OrderBooks can be drained in a single transaction
- On a mature protocol with active trading, fees accumulate continuously. Weeks of accumulation could represent 5-10% of annual protocol revenue
- Example: Protocol with 10 active markets, 0.05% taker fees on 10,000 ETH weekly volume accumulates approximately 5 ETH per week in fees
- Total value at risk scales with protocol maturity and trading volume, potentially exceeding $100,000 USD on established deployments

Protocol Revenue Impact:

- Protocol loses fee revenue needed to fund operations and development
- No recovery mechanism exists after successful attack
- Loss of user trust and potential capital flight from the protocol

Attack Characteristics:

- Requires only ADMIN_ROLE key compromise (phishing, insider threat, or key management vulnerability)
- Minimal execution cost: 0.1-0.5 ETH gas for one `setFeeCollector` call plus multiple `claimFees` calls
- Attack duration: Single transaction batch (under 1 minute)
- Profit margin: Extremely high ROI (theft of 10 ETH costs ~0.5 ETH gas = 2000% ROI)
- Optional stealth: Attacker can restore legitimate fee collector after theft to temporarily obscure the attack

Remediation

Implement Timelock Governance:

The only effective remediation for this vulnerability is to implement timelock governance for `Manager.setFeeCollector`. This ensures all fee collector changes are observable, delayable, and cancelable, providing the community with sufficient time to detect and respond to malicious actions.

A comprehensive timelock architecture is detailed in **Appendix A: Timelock Governance Architecture**. The implementation should:

1. Deploy a `TimelockController` with a 48-hour minimum delay
2. Transfer `ADMIN_ROLE` from EOA/multisig to the timelock contract
3. Route all `Manager.setFeeCollector()` calls through `timelock.schedule()` and `timelock.execute()`
4. Implement monitoring infrastructure to alert on `CallScheduled` events for fee collector changes

This approach addresses the root cause by making privileged operations transparent and providing a mandatory delay period before execution. During the delay, other members of the multisig quorum can review the proposed change and have sufficient time to respond. If a malicious operation is detected, guardians or other multisig signers can cancel the scheduled transaction before it executes.

See **Appendix A** for complete implementation details, monitoring examples, and operational procedures.

Code

Manager.setFeeCollector (Manager.sol:227-233):

```
function setFeeCollector(address _feeCollector) external override onlyAdminRole {
    if (_feeCollector == address(0)) revert Manager__InvalidFeeCollector();
    if (_feeCollector != feeCollector) {
        address oldFeeCollector = feeCollector;
        feeCollector = _feeCollector;
        emit FeeCollectorUpdated(oldFeeCollector, _feeCollector);
    }
}
```

OrderBook.claimFees (OrderBook.sol:519-540):

```
function claimFees()
    external
    override
    nonReentrant
    onlyManagerRole(ADMIN_ROLE)
    returns (bool baseSuccess, bool quoteSuccess)
{
    address feeCollector = manager.feeCollector();
    bytes32 pendingBaseSlot;
    bytes32 pendingQuoteSlot;
    assembly {
        pendingBaseSlot := pendingBaseFees.slot
        pendingQuoteSlot := pendingQuoteFees.slot
    }
    uint256 baseToClaim;
    uint256 quoteToClaim;
    (baseSuccess, quoteSuccess, baseToClaim, quoteToClaim) = OrderBookAux.claimFees(
        pendingBaseFees,
        pendingQuoteFees,
        address(BASE_TOKEN),
        address(QUOTE_TOKEN),
        feeCollector,
        pendingBaseSlot,
        pendingQuoteSlot
    );
    emit FeesClaimed(feeCollector, baseToClaim, quoteToClaim, baseSuccess, quoteSuccess);
    return (baseSuccess, quoteSuccess);
}
```

OrderBookAux.claimFees (OrderBookAux.sol:146-190):

```
function claimFees(
    uint256 pendingBaseFeesValue,
    uint256 pendingQuoteFeesValue,
    address baseToken,
    address quoteToken,
    address feeCollector,
    bytes32 pendingBaseFeesSlot,
    bytes32 pendingQuoteFeesSlot
)
    external
    returns (
        bool baseSuccess,
        bool quoteSuccess,
        uint256 baseToClaim,
        uint256 quoteToClaim
    )
{
    uint256 basePending = pendingBaseFeesValue;
    uint256 quotePending = pendingQuoteFeesValue;
    baseToClaim = basePending;
    quoteToClaim = quotePending;
    if (baseToClaim == 0 && quoteToClaim == 0) revert IOrderBook.OrderBook__NoFeesToClaim();
    if (baseToClaim > 0) {
        uint256 updatedBase = basePending - baseToClaim;
        assembly {
            sstore(pendingBaseFeesSlot, updatedBase)
        }
        baseSuccess = _trySafeTransfer(baseToken, feeCollector, baseToClaim);
        if (!baseSuccess) {
            assembly {
                sstore(pendingBaseFeesSlot, basePending)
            }
        }
    } else {
        baseSuccess = true;
    }
```

```
37      if (quoteToClaim > 0) {
38          uint256 updatedQuote = quotePending - quoteToClaim;
39          assembly {
40              sstore(pendingQuoteFeesSlot, updatedQuote)
41          }
42          quoteSuccess = _trySafeTransfer(quoteToken, feeCollector, quoteToClaim);
43          if (!quoteSuccess) {
44              assembly {
45                  sstore(pendingQuoteFeesSlot, quotePending)
46              }
47          }
48      } else {
49          quoteSuccess = true;
50      }
51  }
```

POC

Attack Execution:

1. **Precondition:** ADMIN_ROLE key is compromised (phishing, insider threat, or key management vulnerability)

2. **Monitor fee accumulation:** Query `pendingBaseFees` and `pendingQuoteFees` for all OrderBooks to determine total value at risk

3. **Execute atomic swap and drain:**

   - Call `Manager.setFeeCollector(attackerAddress)` - updates fee collector immediately with no delay
   - Enumerate all registered OrderBooks via `Manager.allOrderBooks` or `Manager.getPaginatedOrderBookAddresses`
   - For each OrderBook, call `OrderBook.claimFees()`
   - Each `claimFees` call reads `manager.feeCollector()` dynamically, which now returns `attackerAddress`
   - `OrderBookAux.claimFees` zeros pending fee accumulators and transfers tokens to `attackerAddress`
   - All pending fees across all markets flow to attacker address in one transaction or rapid sequence

4. **Optional stealth step:** Call `Manager.setFeeCollector(legitimateCollector)` to restore original address and temporarily obscure the theft

Attack Characteristics:

- Capital needed: 0 ETH (requires only ADMIN key)
- Execution cost: 0.1-0.5 ETH gas (1 `setFeeCollector` + N `claimFees` calls)

- Potential profit: 1-100+ ETH depending on protocol maturity and trading volume
- Attack duration: Single transaction batch (under 1 minute)
- Profit margin: Extremely high ROI (theft of 10 ETH costs ~0.5 ETH gas = 2000% ROI)

Detection:

- `FeeCollectorUpdated` and `FeesClaimed` events are emitted, providing on-chain evidence
- If executed during off-hours, detection may be delayed
- Monitoring of Manager events could detect the attack, but response time may be insufficient to prevent fund loss

Discussion

Developer: In practice, before self-governance, the feeCollector is the same wallet as the main admin wallet, so this is not an issue. During periodic fee claims, protocol funds are processed and sent to a treasury cold wallet for more secure storage. When the protocol is self-governing, the admin role will already be a timelock contract, which is the solution proposed by the auditor. The timelock contract is out of the scope of this audit.

Auditor: Acknowledged. The current operational setup (same wallet for admin and feeCollector) mitigates the risk in practice, and the planned timelock governance aligns with the recommended remediation. The underlying code vulnerability remains, but the operational and future governance mitigations are reasonable.

Finding 6: Orderbook Pointer Replacement To Strand User Liquidity

Severity: HIGH
Status: ACKNOWLEDGED

Description

The Manager contract allows ADMIN_ROLE to overwrite the canonical `orderBooks[pairHash]` pointer for an existing trading pair without migrating user state. This redirects all integrations and frontends to a new OrderBook address while user funds, open orders, and pending fees remain locked in the old OrderBook.

The vulnerability stems from an intentional ADMIN bypass in the duplicate pair registration check. When ADMIN calls `Manager.registerOrderBook` for an already-registered pair, the function bypasses the duplicate check and unconditionally overwrites the canonical pointer. No code exists to migrate user state (orders, withdrawable balances, pending fees) from the old OrderBook to the new one.

This causes severe user impact: users who deposited in the old OrderBook lose UI access to their holdings. Frontends and integrations calling `Manager.getOrderBookAddress()` now receive the new OrderBook address, displaying empty balances for affected users. Users must manually interact with the old contract address to recover funds, which is inaccessible to non-technical users.

Source

Manager.registerOrderBook:

`contracts/Manager.sol:404-500` - Allows `ADMIN_ROLE` to overwrite `orderBooks[pairHash]` with a new OrderBook address:

```
function registerOrderBook(address baseToken, address quoteToken, address orderBookAddress)
    external
    payable
    virtual
    override
    nonReentrant
{
    bool isAdmin = _hasOrderBookRoleInternal(ADMIN_ROLE, msg.sender);
    bool isFactory = hasRole(FACTORY_ROLE, msg.sender);

    // Access control: only admin or trusted factory
    if (!isAdmin && !isFactory) {
        revert AccessControlUnauthorizedAccount(msg.sender, ADMIN_ROLE);
    }

    // ... input validation ...

    bytes32 pairHash = getPairHash(baseToken, quoteToken);
    // Only prevent duplicates for non-admin users (allows admin to register OrderBook v2)
```

```
    if (orderBooks[pairHash] != address(0) && !isAdmin) {
        revert Manager__PairAlreadyRegistered();
    }

    // ... bytecode and parameter validation ...

    // 3. Store if all checks pass (BEFORE external calls to prevent reentrancy)
    // Update canonical pointer (keeps current semantics: latest wins)
    orderBooks[pairHash] = orderBookAddress;

    // ... registry updates ...
}
```

`contracts/Manager.sol:472` - Unconditional pointer overwrite: `orderBooks[pairHash] = orderBookAddress;`

Comment at `contracts/Manager.sol:428` explicitly confirms this is intentional: “Only prevent duplicates for non-admin users (allows admin to register OrderBook v2)”.

Manager.getOrderBookAddress:

`contracts/Manager.sol:622-624` - Returns `orderBooks[pairHash]`, which is the integration point for all frontends and aggregators:

```
function getOrderBookAddress(address tokenA, address tokenB) external view override returns (address) {
    bytes32 pairHash = getPairHash(tokenA, tokenB);
    return orderBooks[pairHash];
}
```

No State Migration:

No code exists in `Manager.sol` or `OrderBook.sol` to migrate user state (orders, withdrawable balances, pending fees) from the old OrderBook to the new one.

Impact

Direct User Impact:

- Users lose UI access to their holdings in the old OrderBook
- Frontends and integrations calling `Manager.getOrderBookAddress()` now receive the new OrderBook address
- UI displays empty balances for affected users (shows new OrderBook state, not old)
- Users must manually interact with the old contract address to recover funds
- Total value at risk: All TVL in old OrderBook (could be $100k-$10M+ on active trading pair)

Protocol Impact:

- Liquidity fragmentation across two OrderBooks for the same trading pair
- Loss of user trust if funds become invisible in UI
- Reputational damage from apparent loss of funds
- Reduced market efficiency due to fragmented liquidity

Attack Vectors:

- If new OrderBook contains backdoor (e.g., malicious admin withdrawal function), attacker can drain new deposits
- Attacker can set elevated fees on new OrderBook (up to 0.5% via `Manager.setFees`)
- Old OrderBook becomes ghost market, fragmenting liquidity and harming price discovery

Mitigating Factors:

- Requires privileged ADMIN_ROLE (not a permissionless attack)
- Events provide on-chain audit trail (`OrderBookRegistered` event emitted)
- Old OrderBook remains functional (funds not technically stolen, just UI-inaccessible)
- Monitoring could detect pointer changes

Remediation

Remove ADMIN Bypass for Duplicate Pair Registration:

Change the duplicate pair check to prevent pointer replacement entirely:

```
// Manager.sol line 277 - Remove isAdmin bypass
function registerOrderBook(
    address baseToken,
    address quoteToken,
    address orderBookAddress
) external override {
    // ... existing validation ...

    bytes32 pairHash = getPairHash(baseToken, quoteToken);
    // Remove !isAdmin condition - prevent all duplicate registrations
    if (orderBooks[pairHash] != address(0)) {
        revert Manager__PairAlreadyRegistered();
    }

    // ... rest of function ...
}
```

Alternative: Implement State Migration:

If pointer replacement is intended for upgrades, require atomic migration of all user state before allowing registration:

```
function registerOrderBookWithMigration(
    address baseToken,
    address quoteToken,
    address newOrderBookAddress,
    address oldOrderBookAddress
) external override onlyAdminRole {
    bytes32 pairHash = getPairHash(baseToken, quoteToken);
    address oldOrderBook = orderBooks[pairHash];

    if (oldOrderBook == address(0)) {
        revert Manager__PairNotRegistered();
    }
    if (oldOrderBook != oldOrderBookAddress) {
        revert Manager__InvalidOldOrderBook();
    }

    // Migrate all user state from old to new OrderBook
    _migrateOrderBookState(oldOrderBook, newOrderBookAddress);

    // Only after successful migration, update pointer
    orderBooks[pairHash] = newOrderBookAddress;
    emit OrderBookReplaced(pairHash, oldOrderBook, newOrderBookAddress);
}
```

Additional Safeguards:

1. **Add Timelock:** Require 24-48 hour delay before pointer replacement takes effect, giving users time to withdraw and allowing other multisig signers time to review and respond

2. **Emit Distinct Events:** Add `OrderBookReplaced` event that clearly indicates an existing pair pointer was overwritten, including old and new addresses

3. **Multi-Signature Requirement:** Require multiple ADMIN signers to approve pointer replacement

4. **Migration Flag:** Add explicit migration flag to `getPairOrderBooks` return value indicating which OrderBook in the list is currently canonical

Code

Manager.registerOrderBook - ADMIN Bypass (Manager.sol:277-287):

```
bytes32 pairHash = getPairHash(baseToken, quoteToken);
// Only prevent duplicates for non-admin users (allows admin to register OrderBook v2)
if (orderBooks[pairHash] != address(0) && !isAdmin) {
    revert Manager__PairAlreadyRegistered();
}
```

Manager.registerOrderBook - Pointer Overwrite (Manager.sol:310):

```
orderBooks[pairHash] = orderBookAddress;
```

Manager.getOrderBookAddress (Manager.sol:661-665):

```
function getOrderBookAddress(address tokenA, address tokenB) external view override returns (address)
    {
    bytes32 pairHash = getPairHash(tokenA, tokenB);
    return orderBooks[pairHash];
}
```

POC

Attack Execution:

1. **Precondition:** ADMIN_ROLE key is compromised (phishing, insider threat, or key management vulnerability)
2. **Deploy new OrderBook:** Attacker deploys a new OrderBook contract for an existing trading pair. This can be legitimate code or contain a backdoor for future exploitation
3. **Call registerOrderBook:** Attacker calls `Manager.registerOrderBook(baseToken, quoteToken, newOrderBookAddress)` with ADMIN privileges
4. **Manager overwrites pointer:**
   - Access control check at line 271: `isAdmin = _hasOrderBookRoleInternal(ADMIN_ROLE, msg.sender)` returns true
   - Pair existence check at line 277: `if (orderBooks[pairHash] != address(0) && !isAdmin)` evaluates to false because `isAdmin=true`, so NO revert occurs
   - Bytecode validation at lines 288-303 passes if attacker deploys legitimate OrderBook code
   - Canonical pointer overwrite at line 310: `orderBooks[pairHash] = newOrderBookAddress`
   - Append-only index update at lines 329-334: `_pairOrderBooks[pairHash].push(newOrderBookAddress)`
   - Event emission at line 345: `OrderBookRegistered(pairHash, baseToken, quoteToken, newOrderBookAddress)`
5. **Impact:**
   - All integrations/UIs calling `Manager.getOrderBookAddress(baseToken, quoteToken)` now receive new address

- Users' existing orders, withdrawable balances, and pending fees remain in old OrderBook
- UI shows empty balances for affected users (displays new OrderBook state, not old)
- Users must manually interact with old contract address to recover funds
- New users deposit in new OrderBook, fragmenting liquidity

Attack Characteristics:

- Capital needed: 0 ETH (requires only ADMIN key)
- Execution cost: Low gas (1 `registerOrderBook` call)
- Potential profit:
  - If new OrderBook contains backdoor, attacker can drain new deposits
  - Can set elevated fees on new OrderBook (up to 0.5% via `Manager.setFees`)
  - Old OrderBook becomes ghost market, fragmenting liquidity

Detection:

- `OrderBookRegistered` event emitted at line 345 provides on-chain evidence
- However, event does not explicitly flag this as a pointer replacement vs. new pair registration
- Most users do not monitor Manager events
- UI redirection may go unnoticed initially until users check balances

User Recovery:

- Old OrderBook remains fully functional (users can still call `withdrawTokens()` if they know the address)
- However, integrations lose the reference, making manual contract interaction necessary
- Non-technical users cannot recover funds without assistance

Discussion

Developer: Pointer replacement by ADMIN_ROLE is intentional and supports coordinated OrderBook upgrades. No funds are lost - legacy OrderBooks remain fully accessible via `getPairOrderBooks()`, and the UI shows all versions so users can cancel and withdraw normally. New liquidity flows to the upgraded OrderBook as expected. Policy is to only update pointers during planned upgrades with sufficient notice to traders and market makers, with UI tools provided to help users migrate if desired.

Auditor: Acknowledged as a design decision.

Finding 7: Rebasing Token Accounting Break

Severity: HIGH
Status: RESOLVED

Description

The OrderBook contract maintains static accounting records (`withdrawableBase`, `withdrawableQuote`, order amounts) that assume token balances remain constant between transactions. Rebasing tokens violate this invariant by automatically adjusting holder balances without any transfer occurring, causing irreconcilable divergence between recorded accounting and actual token holdings.

Unlike fee-on-transfer tokens (which can be detected at transfer time via balance-delta verification), rebasing tokens change balances between transactions through external rebase events. This makes the issue fundamentally unfixable through transfer-time validation and requires complete exclusion of rebasing token types.

The vulnerability manifests in three failure modes:

1. **Positive Rebase - Permanent Fund Lockup**: When the token's total supply increases (e.g., stETH earning staking rewards, aTokens accruing interest), the contract's actual token balance increases proportionally while all recorded accounting remains static. Surplus tokens become permanently trapped with no accounting record, and no user can claim the rebase gains as they're not tracked in state.

2. **Negative Rebase - Protocol Insolvency**: When the token's total supply decreases (e.g., AMPL negative rebase, slashing event), the contract's actual token balance decreases proportionally while all recorded accounting remains at original values. Total withdrawable balances exceed actual contract holdings, causing protocol insolvency where last withdrawers receive nothing, with no mechanism to proportionally reduce user balances.

3. **Active Order Desynchronization**: Resting maker orders record fixed amounts that become stale after rebases. For example, a maker places an order for 1,000 AMPL at price X. After AMPL rebases +10% overnight, the contract now holds 1,100 AMPL for this order, but the order still records 1,000 AMPL. When filled, the taker pays for 1,000 AMPL but 100 AMPL remains locked. The reverse scenario causes orders to promise more than available.

Factory Pattern Amplifies Risk:

The OrderBook factory pattern creates hundreds of potential deployment vectors with no centralized token vetting for each new OrderBook deployment. Human operators will inevitably deploy with problematic tokens (e.g.,

aUSDC mistaken for USDC). A single operational mistake renders that entire OrderBook permanently broken. Common confusions include: aUSDC vs USDC, stETH vs wstETH, sDAI vs DAI. Risk multiplies with every new deployment.

Token Upgrade Threat:

Many tokens can upgrade to add rebasing behavior post-deployment. Governance-controlled tokens (e.g., UNI, COMP) could vote to add yield distribution. Stablecoin upgrades could add interest-bearing features (similar to sDAI evolution). Proxy-upgradeable tokens allow complete implementation swaps. Examples include DAI to sDAI (savings variant with rebasing yield), and USDC could theoretically add yield. Unlike fee-on-transfer additions (simple parameter changes), rebasing requires fundamental contract redesign, making intentional upgrades less likely but not impossible. The risk is particularly acute for "savings" or "interest-bearing" token variants that existing projects might introduce.

Source

OrderBook.sol - Static Balance Accounting:

`contracts/OrderBook.sol:438-444` - User balance mappings assume static token holdings:

```
/**
 * @notice Mapping from user address to their withdrawable balance of base tokens.
 * @dev These balances accrue from cancelled sell orders (remaining base) or filled buy maker orders (received base).
 *      Users can claim these via `withdrawTokens()`.
 */
mapping(address => uint256) public withdrawableBase;
mapping(address => uint256) public withdrawableQuote;
```

`contracts/OrderBook.sol:455` - Order storage records fixed amounts:

```
/**
 * @notice Mapping from order ID to the Order struct containing its details.
 * @dev This is the primary storage for all order data, active or inactive.
 */
mapping(uint256 => Order) internal orders;

// Order struct (lines 350-360) contains:
struct Order {
    address trader;
    uint96 price;
    uint128 amount;           // Fixed amount, never adjusted for rebases
    uint128 claimRangeStart;
    uint128 claimRangeEnd;
    // ... status and metadata ...
}
```

`contracts/OrderBook.sol:1625-1657` - `_applyMatchResults` credits users with nominal amounts:

```
function _applyMatchResults(address takerAddress, bool isTakerBuy, MatchAccumulator memory acc)
    internal
    returns (uint256 takerFeePaid)
{
    if (acc.baseFilled == 0) return 0;

    takerFeePaid = isTakerBuy ? acc.takerBaseFee : acc.takerQuoteFee;

    uint256 wBase = acc.wBase;
    if (wBase > 0) {
        withdrawableBase[takerAddress] += wBase;  // Static addition, never adjusted
    }

    uint256 wQuote = acc.wQuote;
    if (wQuote > 0) {
        withdrawableQuote[takerAddress] += wQuote;  // Static addition
    }

    // ... fee accumulation ...
}
```

`contracts/OrderBook.sol:2564-2582` - `_handleSellOrderCollateral` locks base tokens:

```
function _handleSellOrderCollateral(address trader, uint256 baseAmount) internal {
    // First consume any withdrawable base balance for the user
    uint256 availableBase = withdrawableBase[trader];
    if (availableBase != 0) {
        if (availableBase >= baseAmount) {
            unchecked {
                withdrawableBase[trader] = availableBase - baseAmount;
            }
            return;
        }

        withdrawableBase[trader] = 0;
        unchecked {
            baseAmount -= availableBase;
        }
    }

    IERC20(address(BASE_TOKEN)).safeTransferFrom(trader, address(this), baseAmount);
    // Contract balance recorded, but will diverge if BASE_TOKEN rebases
}
```

`contracts/OrderBook.sol:900-920` - `withdrawTokens` transfers recorded amounts without balance reconciliation:

```
function withdrawTokens() external override nonReentrant {
    address user = msg.sender;
    uint256 baseAmount = withdrawableBase[user];
    uint256 quoteAmount = withdrawableQuote[user];

    if (baseAmount == 0 && quoteAmount == 0) {
        revert OrderBook__NothingToWithdraw();
    }

    if (baseAmount != 0) {
        withdrawableBase[user] = 0;
        IERC20(BASE_TOKEN).safeTransfer(user, baseAmount);  // May fail if rebase caused insolvency
    }

    if (quoteAmount != 0) {
        withdrawableQuote[user] = 0;
        IERC20(QUOTE_TOKEN).safeTransfer(user, quoteAmount);  // May fail
    }

```

```
        emit TokensClaimed(user, baseAmount, quoteAmount);
    }
```

Manager.sol - No Token Behavior Validation:

`contracts/Manager.sol:404-500` - `registerOrderBook` validates bytecode and parameters but not token behavior:

```
function registerOrderBook(address baseToken, address quoteToken, address orderBookAddress)
    external
    payable
    virtual
    override
    nonReentrant
{
    // ... access control and input validation ...

    // 1. Bytecode Check
    // ... validates OrderBook contract bytecode ...

    // 2. Parameter Checks
    IOrderBookCore candidateBook = IOrderBookCore(orderBookAddress);
    IOrderBook.Params memory params = candidateBook.getParams();
    if (address(params.baseToken) != baseToken) revert Manager__ParameterMismatch("BASE_TOKEN");
    if (address(params.quoteToken) != quoteToken) revert Manager__ParameterMismatch("QUOTE_TOKEN");
    // ... other parameter checks ...

    // No rebasing detection
    // No token whitelist check
    // No balance behavior validation

    // 3. Registration
    // ... register OrderBook in registry ...
}
```

`contracts/Manager.sol:70-100` - No token whitelist exists:

```
// Manager state variables (lines 70-100):

/// @notice Expected keccak256 hash of the NORMALIZED OrderBook runtime bytecode
bytes32 public override expectedNormalizedBytecodeHash;

/// @notice Array storing all registered OrderBook contract addresses.
address[] public allOrderBooks;

/// @notice Mapping from a unique token pair hash to the corresponding OrderBook contract address.
mapping(bytes32 => address) public override orderBooks;

// No token whitelist mapping
// No token status tracking
// No approved tokens registry
```

Impact

Permanent Fund Lockup (Positive Rebase):

When a rebasing token experiences a positive rebase, surplus tokens become permanently trapped. For example, if stETH earns 4% APR staking rewards, after one year with 1,000,000 stETH locked across all orders, the contract

balance grows to 1,040,000 stETH while total `withdrawableBase` remains 1,000,000 stETH. This results in 40,000 stETH (approximately $80M at $2,000/ETH) permanently trapped with no accounting mechanism to distribute rebase gains. Funds are irrecoverable even with contract upgrades.

Protocol Insolvency (Negative Rebase):

When a rebasing token experiences a negative rebase, the protocol becomes insolvent. For example, if AMPL targets $1 and is currently $1.10, a negative rebase of -9% occurs to restore the peg. If 100 users deposit 1,000,000 AMPL total on day 1, the contract balance decreases to 910,000 AMPL while total withdrawable remains 1,000,000 AMPL, creating a deficit of 90,000 AMPL. The first 91% of withdrawers succeed, but the last 9% of withdrawers lose 100% of their funds with no fair mechanism to distribute losses.

Complete OrderBook Failure:

Once a rebasing token is deployed, accounting divergence is irreversible. The issue cannot be fixed without full migration (which is impossible with active orders). The entire OrderBook must be deprecated, all users must exit positions (if solvent), and liquidity is permanently lost for that trading pair.

Factory Pattern Multiplies Risk:

With potentially hundreds of OrderBooks, the probability of operational error approaches 1 over time. Common mistakes include deploying with aUSDC thinking it's USDC, using stETH thinking it's just ETH, or using sDAI thinking it's a safe stablecoin. One mistake renders one OrderBook permanently broken, and there is no way to validate every token used across all deployments.

Total Value at Risk:

Per OrderBook: 100% of resting liquidity if negative rebase, or 100% of rebase gains if positive rebase (locked forever). Protocol-wide: Any OrderBook using rebasing tokens will eventually fail. Given the factory pattern, the likelihood of deployment with a rebasing token is high. Even if initial deployment is careful, token upgrades can introduce rebasing.

Why Not CRITICAL:

- Requires specific token choice (operational mistake) OR future token upgrade (external dependency)
- Scoped to individual OrderBook instance, not protocol-wide
- Can be mitigated with Manager-level validation and token whitelist
- Unlike fee-on-transfer, rebasing is less common and easier to detect via interface checks

Why HIGH Severity:

- Factory pattern makes operational mistakes likely across multiple deployments
- Results in complete insolvency or permanent fund lockup (100% loss on that OrderBook)
- Affects legitimate, active OrderBooks (not just new deployments)
- Irreversible and unfixable once deployed
- Real-world examples exist (stETH, aTokens, AMPL)

Remediation

Implement Token Whitelist with Validation:

Implement a mandatory token approval system in Manager that validates tokens before allowing OrderBook registration:

```
// contracts/Manager.sol

mapping(address => TokenStatus) public tokenStatus;
address[] public approvedTokens;

enum TokenStatus {
    NotApproved,
    Approved,
    Deprecated
}

function approveToken(address token) external onlyAdminRole nonReentrant {
    require(tokenStatus[token] == TokenStatus.NotApproved, "Manager: already approved or deprecated");

    _validateTokenSafety(token);

    tokenStatus[token] = TokenStatus.Approved;
    approvedTokens.push(token);

    emit TokenApproved(token, "Passed validation suite");
}

function _validateTokenSafety(address token) internal {
    require(token != address(0), "Manager: zero address");

    // Check 1: Basic ERC20 compliance
    try IERC20Metadata(token).decimals() returns (uint8 decimals) {
        require(decimals <= 18, "Manager: decimals > 18");
    } catch {
        revert("Manager: invalid ERC20");
    }

    // Check 2: Balance-delta verification (detects fee-on-transfer)
    uint256 testAmount = 1000 * 10**IERC20Metadata(token).decimals();
    uint256 balBefore = IERC20(token).balanceOf(address(this));

    IERC20(token).safeTransferFrom(msg.sender, address(this), testAmount);

    uint256 balAfter = IERC20(token).balanceOf(address(this));
    uint256 actualReceived = balAfter - balBefore;

    if (actualReceived != testAmount) {
        if (actualReceived > 0) {
            IERC20(token).safeTransfer(msg.sender, actualReceived);
        }
```

```
            revert Manager__FeeOnTransferDetected(token, testAmount, actualReceived);
        }

        // Check 3: Rebasing detection via common interfaces
        if (_hasRebasingInterface(token)) {
            IERC20(token).safeTransfer(msg.sender, testAmount);
            revert Manager__RebasingTokenDetected(token);
        }

        // Check 4: Gas cost heuristic for balanceOf
        uint256 gasBefore = gasleft();
        IERC20(token).balanceOf(address(this));
        uint256 gasUsed = gasBefore - gasleft();

        if (gasUsed > 10000) {
            IERC20(token).safeTransfer(msg.sender, testAmount);
            revert("Manager: balanceOf too expensive (possible rebasing)");
        }

        // Return test tokens
        IERC20(token).safeTransfer(msg.sender, testAmount);
    }

    function _hasRebasingInterface(address token) internal view returns (bool) {
        // Check for stETH-like (Lido)
        try IStETH(token).sharesOf(address(this)) returns (uint256) {
            return true;
        } catch {}

        try IStETH(token).getPooledEthByShares(1e18) returns (uint256) {
            return true;
        } catch {}

        // Check for aToken-like (Aave)
        try IAToken(token).scaledBalanceOf(address(this)) returns (uint256) {
            return true;
        } catch {}

        // Check for Ampleforth-like
        try IAmpleforth(token).scaledBalanceOf(address(this)) returns (uint256) {
            return true;
        } catch {}

        return false;
    }

    function registerOrderBook(
        address baseToken,
        address quoteToken,
        address orderBookAddress
    )
        external
        payable
        virtual
        override
        nonReentrant
    {
        // Enforce token whitelist
        if (tokenStatus[baseToken] != TokenStatus.Approved) {
            revert Manager__TokenNotApproved(baseToken);
        }
        if (tokenStatus[quoteToken] != TokenStatus.Approved) {
            revert Manager__TokenNotApproved(quoteToken);
        }

        // ... existing bytecode validation ...
        // ... existing parameter checks ...
        // ... existing registration logic ...
    }
```

Operational Guidelines:

Create documentation that explicitly lists rebasing tokens that must never be approved, including:

- stETH (use wstETH instead)
- aUSDC, aDAI, aTokens (use underlying tokens instead)
- sDAI (use DAI instead)
- AMPL, OHM, sOHM (reject entirely)

Monitoring Infrastructure:

Implement off-chain monitoring for token upgrades and balance drift. Monitor proxy implementation upgrades for approved tokens and alert on any changes. Monitor OrderBooks for balance drift (difference between actual token balance and recorded accounting) as an indicator of rebasing behavior.

POC

Attack Scenario - Positive Rebase (stETH Example):

1. OrderBook is deployed with BASE_TOKEN = stETH (rebasing), QUOTE_TOKEN = USDC
2. Maker places sell order: 100 stETH at 2000 USDC/stETH
3. Contract locks 100 stETH and records order.amount = 100 stETH
4. After one year of staking rewards (approximately 4% APR), stETH rebases +4%
5. Contract stETH balance increases to 104 stETH (rebased)
6. Order record remains unchanged: order.amount = 100 stETH
7. Taker executes market buy to purchase 100 stETH for 200,000 USDC
8. Matching logic uses order.amount (100 stETH) for accounting
9. Maker credited: withdrawableQuote[maker] += 200,000 USDC
10. Taker receives: withdrawableBase[taker] += 100 stETH
11. Contract state after: actualBalance = 4 stETH (leftover), withdrawableBase[anyone] = 0
12. Result: 4 stETH (approximately $8,000) permanently trapped with no accounting record

Attack Scenario - Negative Rebase (AMPL Example):

1. OrderBook is deployed with BASE_TOKEN = WETH, QUOTE_TOKEN = AMPL (rebasing)

2. Maker places buy order: Buy 10 WETH for 20,000 AMPL (price = 2000 AMPL/WETH)
3. Contract locks 20,000 AMPL and records quoteValue = 20,000 AMPL
4. AMPL negative rebase occurs: -10% to restore $1 peg
5. Contract AMPL balance decreases to 18,000 AMPL (rebased -10%)
6. Order record remains unchanged: quoteValue = 20,000 AMPL
7. Order fills: Maker receives 10 WETH, Seller credited: withdrawableQuote[seller] += 20,000 AMPL
8. Contract state: actualBalance = 18,000 AMPL, totalWithdrawable = 20,000 AMPL
9. Seller attempts withdrawal: Transfer fails due to insufficient balance
10. Result: Protocol insolvent by 2,000 AMPL, seller cannot access funds

Operational Mistake Scenario (aUSDC Example):

1. Admin wants to deploy USDC/WETH OrderBook
2. Admin sees “aUSDC” in wallet (from previous Aave deposit)
3. Admin mistakenly thinks: “aUSDC is just USDC on Aave, should be fine”
4. Admin deploys: factory.createOrderBook(aUSDC, WETH, …)
5. No validation catches the mistake (Manager.registerOrderBook has no rebasing detection)
6. After one year of lending interest (approximately 3% APR), aUSDC rebases +3%
7. Contract balance: 103,000 aUSDC (earned interest from rebasing)
8. Total withdrawable: 100,000 aUSDC (unchanged in accounting)
9. Result: 3,000 aUSDC ($3,000) trapped, compounding over time

Code Evidence:

The vulnerability is demonstrated by OrderBook maintaining static accounting records (`withdrawableBase`, `withdrawableQuote`, `order.amount`) that never adjust for rebase events. When a rebasing token experiences a positive rebase, the contract’s actual token balance increases proportionally while all recorded accounting remains static, causing surplus tokens to become permanently trapped. When a negative rebase occurs, the contract’s actual balance decreases while recorded accounting remains unchanged, causing protocol insolvency. The Manager.registerOrderBook function validates bytecode and parameters but performs no rebasing detection or token whitelist checks, allowing rebasing tokens to be registered without validation.

Discussion

Developer: dbook strictly does not support rebasing tokens. However, there is no fool-proof solution to check for rebasing logic which could be hidden in non-standard interfaces or delayed until after registration. We must rely on manual checking of the token's contract to ascertain whether it is rebasing or might rebase in the future. For upgradeable token contracts (e.g., USDT), no checks can completely remove this risk - this is inherent to most DeFi protocols interacting with such tokens. Implemented token whitelisting in Manager.sol to require explicit admin approval before any token can be used in OrderBook registration.

Auditor: Mitigated via token whitelist implementation (commit `d682a831`). The whitelist correctly enforces that both base and quote tokens must be explicitly approved by ADMIN_ROLE before OrderBook registration. This prevents accidental deployment with rebasing tokens and creates an audit trail via events. Residual risk remains for admin error or future token upgrades, but this is an industry-standard mitigation approach.

Finding 8: Order History View Denial Of Service

Severity: HIGH
Status: RESOLVED

Description

The `getUserOrderIdsPaged` function iterates through bitmap chunks from `startOrderId` to the global `lastOrderId`. When the orderbook has a high global `lastOrderId` value but a user has sparse order history, the function must scan thousands of empty chunks to find the user's orders. This unbounded iteration can exceed block gas limits, causing the view function to revert and making order history inaccessible via on-chain queries.

The vulnerability exists because the loop is bounded by the global `lastOrderId` rather than the user's actual order count. An attacker can cheaply spam order creation to inflate `lastOrderId`, forcing legitimate users with sparse order history to scan increasingly large ranges of empty chunks.

Progressive Degradation:

This issue becomes progressively worse over time as the global `lastOrderId` counter increases with normal protocol usage. As more orders are created, the gas cost for querying order history increases linearly. Once `lastOrderId` exceeds certain thresholds, the function becomes unusable for users with sparse order history, effectively causing a permanent denial of service for on-chain order history queries. The impact escalates continuously as the protocol matures and order volume grows.

Source

OrderBookAux.sol:

`contracts/libraries/OrderBookAux.sol:463-523` - `getUserOrderIdsPaged()` iterates through `userOrderBitmaps` chunks from `startChunk` to `lastChunk`:

```
function getUserOrderIdsPaged(
    address user,
    uint256 startOrderId,
    uint256 limit,
    uint64 lastOrderId,
    uint8 chunkBits,
    mapping(address => mapping(uint256 => uint256)) storage userOrderBitmaps
) public view returns (uint64[] memory orderIds) {
    if (limit == 0 || startOrderId > lastOrderId) {
        return new uint64[](0);
    }

    uint256 chunkSize = 1 << chunkBits; // e.g. 256
    uint256 chunkMask = chunkSize - 1;
```

```
    uint256 startChunk = startOrderId >> chunkBits;
    uint256 lastChunk = uint256(lastOrderId) >> chunkBits;

    orderIds = new uint64[](limit);
    uint256 found = 0;
    uint256 chunkIndex = startChunk;
    uint256 bitOffset = startOrderId & chunkMask;

    while (chunkIndex <= lastChunk && found < limit) {
        uint256 word = userOrderBitmaps[user][chunkIndex];
        // ... bit scanning logic ...
        unchecked {
            chunkIndex++;
        }
        bitOffset = 0;
    }

    // ... return logic ...
}
```

`lastChunk` is derived from `lastOrderId`.

OrderBook.sol:

`contracts/OrderBook.sol:1089-1097` - Forwards global `lastOrderId` to library function:

```
function getUserOrderIdsPaged(address user, uint256 startOrderId, uint256 limit)
    external
    view
    returns (uint64[] memory orderIds)
{
    return OrderBookAux.getUserOrderIdsPaged(
        user, startOrderId, limit, lastOrderId, USER_ORDER_HISTORY_CHUNK_BITS, userOrderHistoryBitmaps
    );
}
```

`lastOrderId` is a global counter that increments with every order creation.

Impact

View Function Denial of Service:

- On-chain order history queries become impossible for sparse users when `lastOrderId` is high
- UI and indexer integrations that rely on this function fail
- Users must rely on off-chain indexers or event logs instead

Attack Scalability:

- Attacker can gradually spam order creation to inflate `lastOrderId` (permissionless)
- Cost per order: ~150-200k gas (can be done gradually over time)

- Impact: All sparse users affected once `lastOrderId` exceeds ~3.5M
- No mitigation exists once `lastOrderId` is inflated
- Impact worsens continuously as `lastOrderId` grows with normal protocol usage

Gas Consumption Example:

- For `lastOrderId = 5,000,000`: `lastChunk = 5,000,000 / 256 ~= 19,531 chunks`
- User with 2 orders and `limit=50` must scan all 19,531 chunks
- Each empty chunk: ~2,100 gas (cold SLOAD)
- Total: ~19,531 * 2,100 ~= 41M gas (exceeds block limit)

Remediation

Add Maximum Chunk Scan Limit:

Modify `getUserOrderIdsPaged` to cap the number of chunks scanned:

```
function getUserOrderIdsPaged(...) public view returns (uint64[] memory orderIds) {
    // ... existing validation ...

    uint256 maxChunksToScan = 1000; // Cap at 1000 chunks (~256k order IDs)
    uint256 chunksScanned = 0;

    while (chunkIndex <= lastChunk && found < limit && chunksScanned < maxChunksToScan) {
        // ... existing chunk scanning logic ...
        unchecked {
            chunkIndex++;
            chunksScanned++;
        }
    }

    // Return partial results if limit reached
}
```

Track User-Specific Order Bounds:

Maintain per-user maximum order ID to bound scans:

```
mapping(address => uint64) public userMaxOrderId;

function _recordUserOrderHistory(address user, uint64 orderId) internal {
    // ... existing logic ...
    if (orderId > userMaxOrderId[user]) {
        userMaxOrderId[user] = orderId;
    }
}

// In getUserOrderIdsPaged, use userMaxOrderId[user] instead of global lastOrderId
```

Implement Pagination with Cursor:

Return a cursor for continuation instead of scanning from start:

```
struct OrderHistoryPage {
    uint64[] orderIds;
    uint256 nextChunkIndex; // Cursor for next page
}

function getUserOrderIdsPaged(address user, uint256 startChunkIndex, uint256 limit)
    external view returns (OrderHistoryPage memory page)
{
    // Scan from startChunkIndex instead of from startOrderId
    // Return nextChunkIndex for pagination
}
```

POC

Attack Scenario:

1. Attacker creates many small orders over time to inflate global `lastOrderId`
    - Each order costs ~150-200k gas
    - Can be done gradually to avoid detection
    - No special permissions required
2. Over time, `lastOrderId` reaches 5,000,000 (example)
3. Legitimate user with sparse order history (e.g., 2 orders) attempts to query:
    - UI calls `getUserOrderIdsPaged(user, startOrderId=1, limit=100)`
    - Function calculates: `lastChunk = 5,000,000 / 256 = 19,531`
4. Function iterates through all 19,531 chunks:
    - Scans empty chunks 0 through 19,530
    - Each empty chunk: ~2,100 gas (cold SLOAD)
    - Total gas: ~41M (exceeds block gas limit)
5. Transaction reverts with out-of-gas error
6. Order history becomes inaccessible via on-chain view function

Code Evidence:

The vulnerability is demonstrated by `getUserOrderIdsPaged()` looping from `startChunk` to `lastChunk` where `lastChunk = lastOrderId / 256`. The loop at line 485 continues until `chunkIndex <= lastChunk`, meaning it scans all chunks up to the global maximum regardless of the user's actual order distribution. Each iteration performs a storage read (`userOrderBitmaps[user][chunkIndex]`) which costs ~2,100 gas for cold storage, causing the function to exceed block gas limits when `lastOrderId` is high and the user's orders are sparse.

Finding 9: Liquidity Provision Censorship Via Post Only Griefing

Severity: MEDIUM
Status: ACKNOWLEDGED

Description

An attacker can block legitimate market makers from placing post-only orders at specific price levels by placing minimal-sized orders (dust) at those prices. The post-only validation checks if the incoming order price would cross the spread by comparing against `bestBidPrice`/`bestAskPrice`, but it does not consider the size of existing orders at those prices. An attacker can place a dust order that meets the `_minExecutableBase(price)` threshold to manipulate `bestBidPrice`/`bestAskPrice`, causing legitimate post-only orders to be rejected even though the dust order represents negligible liquidity.

The vulnerability creates severe economic asymmetry where an attacker can block unlimited amounts of professional market maker liquidity for minimal cost. With the current testnet configuration (`minQuoteAmount = $10 USDC`), an attacker can block a price level for just $10 in locked capital (recoverable) plus $1-2 in gas fees, while professional market makers attempting to place $100,000+ post-only orders are completely censored.

Source

OrderBook.sol:

`contracts/OrderBook.sol:1526-1556` - `_validateOrderInputInline()` contains the `postOnly` check that reverts if `price >= currentLowestSell` (for buys) or `price <= currentHighestBuy` (for sells):

```
function _validateOrderInputInline(uint256 baseAmount, uint256 price, bool postOnly, bool isBuy)
    internal
    view
    returns (bool requiresFillToSurvive, uint256 quoteValue)
{
    // ... validation checks ...

    if (postOnly) {
        if (requiresFillToSurvive) revert OrderBook__QuoteTooLow();
        if (isBuy) {
            uint256 currentLowestSell = bestAskPrice;
            if (currentLowestSell > 0 && price >= currentLowestSell) revert
    OrderBook__PostOnlyOrderWouldMatch();
        } else {
            uint256 currentHighestBuy = bestBidPrice;
            if (currentHighestBuy > 0 && price <= currentHighestBuy) revert
    OrderBook__PostOnlyOrderWouldMatch();
        }
    }
}
```

`contracts/OrderBook.sol:2091-2108` - `_syncExecutablePriceLevel()` activates/deactivates price levels based on `volume >= _minExecutableBase(price)`:

```
function _syncExecutablePriceLevel(uint256 price, bool isBuy) internal {
    if (price == 0 || price == BUY_SENTINEL_PRICE || price == SELL_SENTINEL_PRICE) return;

    IOrderBook.PriceLevelData storage levelData = isBuy ? buyPriceData[price] : sellPriceData[price];
    uint256 volume = levelData.volume;
    uint256 required = _minExecutableBase(price);
    bool shouldBeActive = volume >= required;

    mapping(uint256 => bool) storage activeMap = isBuy ? buyPriceActive : sellPriceActive;

    if (shouldBeActive) {
        if (!activeMap[price]) {
            _activatePriceLevel(price, isBuy);
        }
    } else if (activeMap[price]) {
        _deactivatePriceLevel(price, isBuy);
    }
}
```

`contracts/OrderBook.sol:2287-2452` - `_activatePriceLevel()` adds price to active linked list, updating `bestBidPrice`/`bestAskPrice` via `_refreshBestPrice()`:

```
function _activatePriceLevel(uint256 price, bool isBuy) internal {
    mapping(uint256 => bool) storage activeMap = isBuy ? buyPriceActive : sellPriceActive;
    if (activeMap[price]) return;

    activeMap[price] = true;

    // ... linked list insertion logic ...

    _refreshBestPrice(isBuy);
}
```

`contracts/OrderBook.sol:2019-2027` - `_refreshBestPrice()` updates cached best prices from active linked list head:

```
function _refreshBestPrice(bool isBuy) internal {
    if (isBuy) {
        uint256 head = nextActiveBuyPrice[BUY_SENTINEL_PRICE];
        bestBidPrice = head == SELL_SENTINEL_PRICE ? 0 : head;
    } else {
        uint256 head = nextActiveSellPrice[SELL_SENTINEL_PRICE];
        bestAskPrice = head == BUY_SENTINEL_PRICE ? 0 : head;
    }
}
```

Impact

Liquidity Provision Denial of Service:

- Market makers using post-only orders (standard practice to avoid adverse selection) cannot place orders at griefed price levels
- Professional market makers may avoid the platform due to this griefing vector

- Affects any protocol integrating this order book for liquidity provision

Spread Manipulation:

- Attacker can artificially widen spreads by blocking tightening attempts, harming market quality
- Market makers forced to choose between paying taker fees (0.3%) or waiting for dust to clear
- Opportunity cost of missing market-making opportunities

Economic Asymmetry:

- Attack cost per price level: $10 USDC locked capital (recoverable) + $1-2 gas fees = $2 net cost
- Victim impact per price level: $100,000+ liquidity blocked, failed transaction gas costs ($2-$20), opportunity cost, forced taker fees ($300 per $100K)
- Asymmetry ratio: 1:50,000 to 1:100,000 (attacker pays $2 to block $100,000+ liquidity)
- One attacker can grief dozens of market makers simultaneously

No Admin Mitigation:

- No emergency controls can remove dust orders
- Only natural market clearing or attacker cancellation resolves the issue
- Attack scales to DoS entire order book for $100-$1,000 in capital

Remediation

Volume-Weighted Best Price Calculation:

Modify post-only validation to ignore dust orders below a meaningful threshold when determining best prices:

```
function _getEffectiveBestPrice(bool isBuy) internal view returns (uint256) {
    uint256 bestPrice = isBuy ? bestAskPrice : bestBidPrice;
    if (bestPrice == 0) return 0;

    uint256 volume = isBuy ? sellPriceData[bestPrice].volume : buyPriceData[bestPrice].volume;
    uint256 minSignificantVolume = (averageOrderSize * MIN_SIGNIFICANT_RATIO) / 100;

    if (volume < minSignificantVolume) {
        // Treat as effectively empty for post-only purposes
        return _getNextBestPrice(bestPrice, isBuy);
    }

    return bestPrice;
}
```

Unconditional minQuoteAmount Enforcement:

Implement a minimum order value check that applies to all resting orders, not just those with `requiresFillToSurvive` flag:

```
function _validateOrderInputInline(...) internal view returns (...) {
    // ... existing checks ...

    // Unconditionally enforce minQuoteAmount for all orders that will rest
    if (minQuoteAmount > 0 && quoteValue < minQuoteAmount && !postOnly) {
        // Even if order gets a fill, remainder must meet minimum
        requiresFillToSurvive = true;
    }
}
```

Time-Weighted Best Price:

Require a price level to maintain sufficient volume for a minimum duration before it affects post-only validation:

```
mapping(uint256 => uint256) public priceLevelActivationTime;
mapping(uint256 => uint256) public priceLevelMinVolume;

function _refreshBestPrice() internal {
    // Only consider price levels active for at least 1 block
    // and with volume above threshold
}
```

POC

Proof of Concept Location:

```
test/poc/Liquidity_Provision_Censorship_via_Post_Only_Griefing/
LiquidityProvisionCensorshipPoC.t.sol
```

Attack Scenario:

Test Configuration:

- `minQuoteAmount`: 10,000,000 (10 USDC with 6 decimals)
- Test price: 100 USDC per BASE token (simplified for demonstration)
- `QUEUE_UNIT_SIZE`: 1,000,000,000 wei (1e9)

Attack Execution:

1. Attacker monitors order book (initially empty)
2. Attacker places minimal sell order at target price (100 USDC/BASE)
    - Order must meet `minQuoteAmount` requirement of 10 USDC

- Calculates minimum base amount needed to meet threshold (~0.1 BASE)
- Places minimal order worth exactly $10 USDC

3. Result:
   - `bestAskPrice` becomes 100 USDC/BASE (target price)
   - Price level is now active with $10 of liquidity
4. Professional market maker attempts to place $100,000 post-only buy order (1000 BASE at 100 USDC/BASE)
5. Transaction reverts with `OrderBook__PostOnlyOrderWouldMatch`
   - Validation check: `price >= bestAskPrice` -> `100 >= 100` -> true -> revert
6. Market maker cannot provide liquidity at that price

Economic Analysis:

- Attacker cost: $10 USDC locked (recoverable) + $1-2 gas = $2 net cost per price level
- Victim impact: $100,000+ liquidity blocked, failed transaction gas ($2-$20), opportunity cost, forced taker fees ($300 per $100K)
- Asymmetry: 1:50,000 ratio ($2 blocks $100,000+ liquidity)
- Scaling: To block 10 price levels costs $100 USDC + $5-20 gas; to block 100 price levels costs $1,000 USDC + $50-200 gas

Code Evidence:

The vulnerability is demonstrated by `_validateOrderInputInline()` checking only price comparison without volume consideration, `_minExecutableBase()` returning extremely small values for high-priced assets, and `_refreshBestPrice()` updating best prices based on any active price level regardless of size. The post-only validation at lines 1549-1553 does not distinguish between dust orders and meaningful liquidity.

How to Run:

```
cd <dbook-contracts-repo>
forge test --match-path test/poc/Liquidity_Provision_Censorship_via_Post_Only_Griefing/
    LiquidityProvisionCensorshipPoC.t.sol -vv
```

Discussion

Developer: Implemented unconditional minQuoteAmount enforcement. A price level becomes active and eligible to update best prices only if above minQuoteAmount, ensuring any order influencing best price represents real, executable liquidity. The post-only invariant is intentionally strict and must never execute immediately against

resting liquidity. Volume-weighted best prices would violate this invariant. While a small but valid order can temporarily block post-only orders at the same price, any participant can clear such liquidity via a non-post-only trade. This is a known economic trade-off, not a security issue.

Auditor: Acknowledged as a design decision. The unconditional minQuoteAmount enforcement helps but doesn't fully eliminate the economic asymmetry. The developer's defense is valid from a design perspective: strict post-only semantics require this behavior. The remaining concern is economic rather than technical.

Finding 10: Minquoteamount Bypass Via Fill First

Severity: MEDIUM
Status: RESOLVED

Description

Users can bypass `minQuoteAmount` validation by ensuring their orders receive an immediate fill, since the validation only applies to orders that rest without any fill. The validation logic uses an AND condition that allows any non-zero fill to bypass the minimum, enabling attackers to place dust orders below the economic threshold and rest them on the book. This enables orderbook pollution, FIFO priority gaming, and potential gas griefing on market orders.

The vulnerability exists because `_createOrder()` only reverts when `requiresFillToSurvive && finalBaseFilled == 0`. If an order receives any fill (even 1 wei), the remainder can rest on the book regardless of whether it meets `minQuoteAmount`. Attackers can self-match by placing a counter-order to ensure their dust order gets a micro-fill, then rest the remainder below the minimum threshold.

Source

OrderBook.sol:

`contracts/OrderBook.sol:1392-1515` - `_createOrder()` calls `_validateOrderInputInline()` to check `minQuoteAmount`:

```
function _createOrder(uint256 baseAmount, uint256 price, bool postOnly, ExecutionType executionType,
    bool isBuy)
    internal
    returns (uint64 orderId, uint256 finalBaseFilled, uint256 finalQuoteFilled, uint256 takerFee)
{
    (bool requiresFillToSurvive, uint256 quoteAmountToLock) =
        _validateOrderInputInline(baseAmount, price, postOnly, isBuy);
    // ... order creation logic ...

    if (requiresFillToSurvive && finalBaseFilled == 0) {
        revert OrderBook__QuoteTooLow();
    }
    // ... rest of function ...
}
```

`contracts/OrderBook.sol:1526-1556` - `_validateOrderInputInline()` sets `requiresFillToSurvive = true` if `quoteValue < minQuoteAmount`:

```
function _validateOrderInputInline(uint256 baseAmount, uint256 price, bool postOnly, bool isBuy)
    internal
    view
```

```
    returns (bool requiresFillToSurvive, uint256 quoteValue)
{
    // ... validation checks ...
    quoteValue = calculateQuoteNeeded(baseAmount, price);
    if (minQuoteAmount > 0 && quoteValue < minQuoteAmount) {
        requiresFillToSurvive = true;
    }
    // ... postOnly checks ...
}
```

Lines 1470-1472: `if (requiresFillToSurvive && finalBaseFilled == 0) { revert OrderBook__QuoteTooLow(); }` - This check only reverts if `finalBaseFilled == 0`, meaning if *any* fill occurs, the `minQuoteAmount` check is bypassed.

OrderBookExtended.sol:

`contracts/OrderBookExtended.sol:97-163` - `decreaseOrderSize()` - `minQuoteAmount` check only applies if `!hasSeenFill`, allowing filled orders to shrink below threshold:

```
function _decreaseOrderSizeInternal(uint64 orderId, uint256 reduceAmount, address trader)
    internal
    returns (uint256 newTotalSize)
{
    // ... validation ...

    bool hasSeenFill = totalFilled > 0;
    uint256 newTotalOrderSize = totalOrderSize - reduceAmount;
    if (minQuoteAmount > 0 && !hasSeenFill) {
        uint256 newQuoteValue = calculateQuoteNeeded(newTotalOrderSize, price);
        if (newQuoteValue < minQuoteAmount) {
            revert OrderBook__QuoteTooLow();
        }
    }

    // ... rest of function ...
}
```

Impact

Dust Order Spam:

- Attackers can place orders below `minQuoteAmount` by ensuring immediate micro-fills
- Orderbook becomes polluted with economically insignificant dust orders
- Degrades market quality and increases gas costs for market orders

FIFO Priority Gaming:

- Dust orders placed via bypass capture early FIFO priority positions
- Legitimate makers' larger orders execute after dust orders at the same price

- Unfair advantage for attackers who game the system

Gas Griefing Potential:

- If many dust orders are placed across different price levels, market orders must iterate through more levels
- While `minQuoteAmount = $10 USDC` provides some defense, the bypass still works
- Can degrade UX and increase gas costs for takers

Economic Asymmetry:

- Attack cost: ~$9.99 per dust order (recoverable) + $1-2 gas
- Impact: Blocks $100K+ liquidity provision, degrades market quality
- Scales with number of dust orders placed

Remediation

Enforce MinQuoteAmount on Resting Remainder:

Modify `_createOrder()` to check remainder size even after partial fills:

```
if (baseAmountRemaining > 0) {
    uint256 remainderQuoteValue = calculateQuoteNeeded(baseAmountRemaining, price);
    if (minQuoteAmount > 0 && remainderQuoteValue < minQuoteAmount) {
        revert OrderBook__QuoteTooLow();
    }
    // ... rest of logic ...
}
```

Remove Fill-First Bypass:

Remove the exception for partially filled orders:

```
// Remove the condition that allows bypass
if (requiresFillToSurvive) {
    revert OrderBook__QuoteTooLow(); // Always enforce, regardless of fill status
}
```

Track Original Order Size:

Maintain original order size and enforce minimum on original, not remainder:

```
struct Order {
    // ... existing fields ...
    uint256 originalAmount; // Track original size
}

// On order creation, check originalAmount meets minQuoteAmount
// On size reduction, ensure new size still meets minimum
```

POC

Proof of Concept Location:

```
test/poc/MinQuoteAmount_Bypass_via_Fill_First/
MinQuoteAmountBypassPoC.t.sol
```

Attack Scenario (Using Testnet Configuration):

Testnet Configuration (Verified):

- Contract Address: `0xf73C123fc8392c84A68f5F048d64e8bC9CB8Cf4c`
- Network: MegaETH Testnet
- `minQuoteAmount`: 10,000,000 (10 USDC with 6 decimals)
- `QUEUE_UNIT_SIZE`: 1,000,000,000 wei (1e9)

Attack Execution:

1. Attacker places liquidity buy order (meets minQuoteAmount):
    - Size: ~0.1 BASE tokens (meets $10 USDC minimum)
    - Quote value: $10 USDC
    - `postOnly=true` to rest on book and provide liquidity
2. Attacker places counter-order (sell) below minQuoteAmount:
    - Size: 100,000,000,000 wei (100 gwei, above `_minExecutableBase` but below minQuoteAmount)
    - Quote value: ~$0.0092 USDC (way below $10 minimum)
    - `postOnly=false` to match against liquidity buy order
    - Gets micro-fill, bypassing minQuoteAmount check
3. Attacker places main order (buy) below minQuoteAmount:
    - Base amount: ~0.099 BASE tokens (calculated to be just below $10 USDC)
    - Quote value: $9.99 USDC (below $10 minimum)
    - `postOnly=false` to match against sell order
    - Order matches 100 gwei against counter-order/sell order
4. Order gets micro-fill:
    - `finalBaseFilled = 100,000,000,000 > 0` (100 gwei)
    - Validation check: `requiresFillToSurvive && finalBaseFilled == 0` -> **FALSE** (no revert)

- minQuoteAmount check **BYPASSED**

5. Remainder rests on book:
    - Remainder: ~0.0989 BASE tokens (original amount minus 100 gwei fill)
    - Remainder quote value: ~$9.98 USDC (below $10 minimum)
    - **Status: BELOW minQuoteAmount but allowed to rest**
6. Attacker repeats 100+ times:
    - Creates dust spam across multiple price levels
    - Each dust order captures FIFO priority
    - Legitimate makers' $100K+ orders execute after dust

Note: The counter-order uses 100 gwei instead of 1 gwei due to `_minExecutableBase(price)` requirements at the test price (100 USDC/BASE). At this price, the minimum executable is ~10 gwei, so 100 gwei ensures the order can match while still being far below minQuoteAmount. The vulnerability mechanism is identical: any non-zero fill bypasses the check.

Economic Analysis:

- Per dust order: ~$9.99 USDC capital (recoverable) + $1-2 gas
- For 100 dust orders: ~$999 USDC capital + $100-200 gas
- Benefit: FIFO priority gaming, maker spread capture
- Impact: Orderbook pollution, unfair priority advantage

Code Evidence:

The vulnerability is demonstrated by `_validateOrderInputInline()` setting `requiresFillToSurvive=true` when `quoteValue < minQuoteAmount`, but `_createOrder()` only reverting when `requiresFillToSurvive && finalBaseFilled == 0`. This AND condition means any non-zero fill bypasses the minimum check, allowing remainders below `minQuoteAmount` to rest on the book. The code comment at line 1470 explicitly documents this behavior: "once partially filled the remaining size may fall below minQuoteAmount and still stay on book".

How to Run:

```
cd <dbook-contracts-repo>
forge test --match-path test/poc/MinQuoteAmount_Bypass_via_Fill_First/MinQuoteAmountBypassPoC.t.sol -vvv
```

Finding 11: Taker Fee Rounding To Zero

Severity: MEDIUM
Status: RESOLVED

Description

Taker fees are computed using integer division that floors toward zero. When trade amounts are sufficiently small, the fee calculation rounds down to zero, allowing traders to pay zero fees. Because the Manager's per-quote-token minimum notional defaults to 0, traders can split large trades into many small trades below the rounding threshold and pay zero aggregate fees, leading to material protocol fee leakage.

The vulnerability exists because fees are calculated per trade using `(amount * takerFeeRate) / FEE_DENOM`, and when `amount * takerFeeRate < FEE_DENOM`, the integer division results in zero fees. Without a protective `minQuoteAmount` set for the quote token, traders can exploit this by splitting orders.

Source

OrderBook.sol:

`contracts/OrderBook.sol:1671-1779` - `_matchAtPriceLevel()` calculates taker fee using `(baseFilledThisLevel * takerFeeRate) / FEE_DENOM` (for buys) or `(quoteValue * takerFeeRate) / FEE_DENOM` (for sells):

```
function _matchAtPriceLevel(
    address takerAddress,
    uint256 priceLevel,
    uint256 maxTakerAmountToFill,
    bool isTakerBuy,
    uint256 takerFeeRate,
    MatchAccumulator memory acc
) internal returns (uint256 baseFilledThisLevel, uint256 remainingVolumeAfter) {
    // ... volume calculation ...

    if (takerFeeRate == 0) {
        if (isTakerBuy) {
            acc.wBase += baseFilledThisLevel;
        } else {
            acc.wQuote += quoteValue;
        }
    } else {
        unchecked {
            if (isTakerBuy) {
                uint256 takerFee = (baseFilledThisLevel * takerFeeRate) / FEE_DENOM;
                uint256 takerReceiveBase = baseFilledThisLevel > takerFee ? baseFilledThisLevel -
    takerFee : 0;
                acc.takerBaseFee += takerFee;
                acc.wBase += takerReceiveBase;
            } else {
                uint256 takerFee = (quoteValue * takerFeeRate) / FEE_DENOM;
```

```
                uint256 takerReceiveQuote = quoteValue > takerFee ? quoteValue - takerFee : 0;
                acc.takerQuoteFee += takerFee;
                acc.wQuote += takerReceiveQuote;
            }
        }
    }

    // ... rest of function ...
}
```

`contracts/OrderBook.sol:75-uint256 internal constant FEE_DENOM = 1_000_000;`

Mechanism:

- Zero-fee threshold: `amount < FEE_DENOM / takerFeeRate`
- For `takerFeeRate = 5,000` (0.5% max): threshold = 1,000,000 / 5,000 = 200 raw units
- For 2-decimal quote token: 200 raw units = 2.00 tokens
- Trader can split large sell into many sub-2.00 token trades to avoid fees entirely

Impact

Protocol Fee Leakage:

- Protocol loses taker fee revenue on affected trades
- Attackers pay zero fees instead of configured taker fee rate
- Material revenue loss when exploited at scale

Economic Impact:

- With `takerFeeRate = 0.5%` (max allowed), any trade < 2.00 tokens pays zero fees
- Large trades can be split into many small trades to bypass fees entirely
- Impact scales with number of split trades and trade volume

Configuration Dependency:

- Vulnerability is mitigated if `minQuoteAmount` is set sufficiently high
- Default value of 0 makes the vulnerability exploitable
- Admin must proactively configure minimums for each quote token

Remediation

Enforce Minimum Quote Amount:

Require `minQuoteAmount` to be set during orderbook registration:

```
function registerOrderBook(address baseToken, address quoteToken, address orderBook)
    external
    onlyFactoryRole
{
    // ... existing validation ...

    uint256 minQuote = quoteTokenMinAmounts[quoteToken];
    if (minQuote == 0) {
        revert Manager__MinQuoteNotConfigured();
    }

    // ... rest of registration ...
}
```

Set Default Minimum Based on Fee Rate:

Calculate minimum based on zero-fee threshold:

```
function setQuoteTokenMinAmount(address quoteToken, uint256 minAmount) external onlyAdminRole {
    // Ensure minimum is above zero-fee threshold
    uint256 zeroFeeThreshold = FEE_DENOM / takerFeeRate;
    if (minAmount < zeroFeeThreshold) {
        revert Manager__MinQuoteTooLow();
    }
    quoteTokenMinAmounts[quoteToken] = minAmount;
}
```

Accumulate Fractional Fees:

Track fractional fees across trades and round up when threshold reached:

```
mapping(address => uint256) public fractionalFeeAccumulator;

function _matchAtPriceLevel(...) internal {
    // ... existing logic ...

    uint256 feeNumerator = baseFilledThisLevel * takerFeeRate;
    uint256 wholeFee = feeNumerator / FEE_DENOM;
    uint256 fractionalFee = feeNumerator % FEE_DENOM;

    fractionalFeeAccumulator[msg.sender] += fractionalFee;
    if (fractionalFeeAccumulator[msg.sender] >= FEE_DENOM) {
        wholeFee += 1;
        fractionalFeeAccumulator[msg.sender] -= FEE_DENOM;
    }

    acc.takerBaseFee += wholeFee;
}
```

POC

Attack Scenario:

Testnet Configuration (Verified):

- Contract Address: `0xf73C123fc8392c84A68f5F048d64e8bC9CB8Cf4c`
- Network: MegaETH Testnet
- `takerFeeRate = 300` (0.03% = 300 / 1,000,000)
- `FEE_DENOM = 1,000,000` (constant)
- `quoteTokenMinAmounts[MegaUSD] = 10,000,000` (10 USDC with 6 decimals) - **configured on testnet**
- Quote token decimals: 6

Zero-Fee Threshold Calculation:

- `threshold = FEE_DENOM / takerFeeRate = 1,000,000 / 300 = 3,333.33 raw units`
- For 6-decimal token: 3,333.33 raw units = 3.33333 tokens
- Any trade < 3.33333 tokens pays zero fees

Note: On the testnet deployment, `minQuoteAmount` is set to 10 USDC, which mitigates this vulnerability. However, the code vulnerability exists when `minQuoteAmount` is not configured (defaults to 0) or is set below the zero-fee threshold. The following attack scenario demonstrates the vulnerability when `minQuoteAmount = 0`:

Attack Execution (When minQuoteAmount = 0):

1. Trader wants to sell 1000 USDC worth of base tokens
2. Instead of one large trade, trader splits into many small trades:
    - Each trade: 3.33 USDC (below 3.33333 threshold for testnet takerFeeRate of 300)
    - Fee calculation: `(3.33 * 300) / 1,000,000 = 0` (rounds to zero)
    - Total fees paid: 0
3. Expected fees (if single trade):
    - `(1000 * 300) / 1,000,000 = 0.3 USDC`
    - Protocol loses 0.3 USDC in fees
4. Attack scales:
    - For $100,000 trade: Split into 30,000 trades of 3.33 USDC each

- Expected fees: $30
- Actual fees: $0
- Protocol loses $30 in revenue

With Maximum takerFeeRate (5,000 = 0.5%):

- Threshold: `1,000,000 / 5,000 = 200 raw units = 2.00 tokens` (for 6-decimal token)
- For $100,000 trade: Split into 50,000 trades of 1.99 USDC each
- Expected fees: $500
- Actual fees: $0
- Protocol loses $500 in revenue

Code Evidence:

The vulnerability is demonstrated by `_matchAtPriceLevel()` calculating taker fees using integer division: `takerFee = (baseFilledThisLevel * takerFeeRate) / FEE_DENOM`. When `baseFilledThisLevel * takerFeeRate < FEE_DENOM`, the division rounds down to zero. The Manager's `quoteTokenMinAmounts` mapping defaults to 0, allowing traders to place orders below the zero-fee threshold. Without a configured minimum, traders can split large trades into many small trades to bypass fees entirely.

Finding 12: Extended Flows Inherit Token Risk

Severity: MEDIUM
Status: RESOLVED

Description

OrderBookExtended delegates order creation and collateral handling to the core OrderBook contract without verifying actual token deltas, thus inheriting the same fee-on-transfer and rebasing token risk. Batch entry points in OrderBookExtended can mint withdrawable credits based on arithmetic fills while the contract received fewer tokens than accounted, propagating the accounting mismatch to extended functionality.

The vulnerability exists because OrderBookExtended's batch functions (`cancelAndOrCreate`, `decreaseOrderSize`) call the core OrderBook's `_createOrder` and collateral handling functions, which use nominal transfer amounts without balance reconciliation. When fee-on-transfer or rebasing tokens are used, the same accounting mismatch occurs in extended flows as in core flows.

Source

OrderBook.sol:

`contracts/OrderBook.sol:1392-1515` - `_createOrder()` uses `_handleBuyOrderCollateral()` / `_handleSellOrderCollateral()` without balance checks:

```
function _createOrder(uint256 baseAmount, uint256 price, bool postOnly, ExecutionType executionType,
    bool isBuy)
    internal
    returns (uint64 orderId, uint256 finalBaseFilled, uint256 finalQuoteFilled, uint256 takerFee)
{
    (bool requiresFillToSurvive, uint256 quoteAmountToLock) =
        _validateOrderInputInline(baseAmount, price, postOnly, isBuy);
    if (isBuy) {
        _handleBuyOrderCollateral(quoteAmountToLock);
    } else {
        _handleSellOrderCollateral(msg.sender, baseAmount);
    }
    // ... rest of function ...
}
```

`contracts/OrderBook.sol:2541-2558` - `_handleBuyOrderCollateral()` calls `safeTransferFrom` without verifying actual received amount:

```
function _handleBuyOrderCollateral(uint256 quoteAmountToLock) internal {
    uint256 availableQuote = withdrawableQuote[msg.sender];
    if (availableQuote != 0) {
        if (availableQuote >= quoteAmountToLock) {
            unchecked {
```

```
                withdrawableQuote[msg.sender] = availableQuote - quoteAmountToLock;
            }
            return;
        }

        withdrawableQuote[msg.sender] = 0;
        unchecked {
            quoteAmountToLock -= availableQuote;
        }
    }

    IERC20(address(QUOTE_TOKEN)).safeTransferFrom(msg.sender, address(this), quoteAmountToLock);
}
```

contracts/OrderBook.sol:2564-2582 - `_handleSellOrderCollateral()` calls `safeTransferFrom` without verifying actual received amount:

```
function _handleSellOrderCollateral(address trader, uint256 baseAmount) internal {
    // First consume any withdrawable base balance for the user
    uint256 availableBase = withdrawableBase[trader];
    if (availableBase != 0) {
        if (availableBase >= baseAmount) {
            unchecked {
                withdrawableBase[trader] = availableBase - baseAmount;
            }
            return;
        }

        withdrawableBase[trader] = 0;
        unchecked {
            baseAmount -= availableBase;
        }
    }

    IERC20(address(BASE_TOKEN)).safeTransferFrom(trader, address(this), baseAmount);
}
```

OrderBookExtended.sol:

contracts/OrderBookExtended.sol:27-32 - `cancelAndOrCreate()` delegates to `_createOrder()` in the core `OrderBook` contract:

```
function cancelAndOrCreate(
    uint64[] calldata cancelIds,
    IOrderBookExtended.CreateParams[] calldata createParams
)
    external
    override
    nonReentrant
    returns (uint64[] memory newOrderIds)
{
    // ... cancellation logic ...

    // Delegates to core _createOrder without additional balance verification
    // ... order creation ...
}
```

contracts/OrderBookExtended.sol:166-173 - `decreaseOrderSize()` delegates to `_decreaseOrderSizeInternal()` which eventually calls `calculateQuoteNeeded` and updates balances:

```
function decreaseOrderSize(uint64 orderId, uint256 reduceAmount)
    external
```

```
    override
    nonReentrant
    returns (uint256 newTotalSize)
{
    return _decreaseOrderSizeInternal(orderId, reduceAmount, msg.sender);
}
```

Mechanism:

1. User calls `OrderBookExtended.cancelAndOrCreate()` with fee-on-transfer token
2. Extended contract delegates to `OrderBook._createOrder()`
3. Core contract calls `_handleBuyOrderCollateral(quoteAmountToLock)`
4. Token takes fee, transfers less than `quoteAmountToLock`
5. Core contract records full `quoteAmountToLock` in accounting
6. Extended contract inherits the accounting mismatch
7. Same deficit accumulation as in core OrderBook flows

Impact

Same Risk as Core OrderBook:

- User funds stranded when contract lacks sufficient balance
- Protocol insolvency for counterparties
- DoS of withdrawals for all users on affected pair
- Accounting mismatch propagates to all extended functionality

Extended Functionality Affected:

- Batch order creation via `cancelAndOrCreate()`
- Order size reduction via `decreaseOrderSize()`
- All convenience functions that delegate to core

No Additional Mitigation:

- Extended contract provides no additional protection
- Same vulnerability exists in both core and extended flows
- Users of extended functionality face identical risks

Remediation

Apply Same Fixes as Core OrderBook:

The remediation for this issue is identical to the fee-on-transfer accounting mismatch fix:

Balance Reconciliation in Core Functions:

Modify `SafeTokenTransfer.safeTransferFrom` to measure actual balance delta:

```
function safeTransferFrom(IERC20 token, address from, address to, uint256 amount) internal {
    if (amount == 0) return;

    uint256 balanceBefore = token.balanceOf(to);

    // ... existing transfer logic ...

    uint256 balanceAfter = token.balanceOf(to);
    uint256 actualReceived = balanceAfter - balanceBefore;

    if (actualReceived != amount) {
        revert SafeTokenTransfer__IncompleteTransfer(amount, actualReceived);
    }
}
```

Update All Call Sites:

Modify collateral handling functions to use actual received amount:

```
function _handleBuyOrderCollateral(uint256 quoteAmountToLock) internal {
    // ... consume withdrawableQuote first ...

    uint256 balanceBefore = IERC20(address(QUOTE_TOKEN)).balanceOf(address(this));
    IERC20(address(QUOTE_TOKEN)).safeTransferFrom(msg.sender, address(this), quoteAmountToLock);
    uint256 balanceAfter = IERC20(address(QUOTE_TOKEN)).balanceOf(address(this));
    uint256 actualReceived = balanceAfter - balanceBefore;

    // Use actualReceived for all accounting instead of quoteAmountToLock
}
```

Token Screening:

Add transfer fee detection to `Manager.registerOrderBook` to prevent registration of fee-on-transfer tokens.

POC

Attack Scenario:

1. Fee-on-transfer token (e.g., 1% fee) is registered as QUOTE_TOKEN via Manager
2. User calls `OrderBookExtended.cancelAndOrCreate()`:

- Cancels existing orders
- Creates new buy orders via `_createOrder()`

3. Extended contract delegates to core:

   - `_createOrder()` calculates `quoteAmountToLock = calculateQuoteNeeded(baseAmount, price)`
   - Calls `_handleBuyOrderCollateral(quoteAmountToLock)`

4. Token transfer occurs:

   - Contract calls `safeTransferFrom(trader, contract, quoteAmountToLock)`
   - Token takes 1% fee, transfers only 99% of `quoteAmountToLock`
   - Contract credits internal accounting as if 100% was received

5. Same accounting mismatch as core OrderBook:

   - Makers credited based on nominal amounts
   - Contract lacks sufficient balance for withdrawals
   - Funds become stranded

6. Extended functionality inherits the risk:

   - All batch operations affected
   - All convenience functions affected
   - No additional protection provided

Code Evidence:

The vulnerability is demonstrated by `OrderBookExtended.cancelAndOrCreate()` delegating to `_createOrder()` at line 664, which in turn calls `_handleBuyOrderCollateral()` without balance verification. The extended contract provides no additional token validation or balance reconciliation, inheriting the same fee-on-transfer risk as the core OrderBook. Since extended functions are convenience wrappers around core functionality, they propagate the same accounting mismatch when non-standard tokens are used.

Finding 13: Min Out Check Uses Gross Fill

Severity: LOW
Status: RESOLVED

Description

The slippage guard for market buys enforces the minimum on the gross base filled (pre-fee), while the taker actually receives base minus the taker fee. As a result, a taker can receive fewer base tokens than `minBaseOut` even though the transaction succeeds. This violates the natural expectation that `minBaseOut` represents the minimum net amount the user will receive after fees.

The function's NatSpec comment states "Minimum base amount to receive," implying net-of-fee semantics, but the implementation checks the gross fill amount before fees are deducted. This creates a mismatch between user expectations and actual behavior.

Source

OrderBook.sol - `_marketBuyInternal` function:

`contracts/OrderBook.sol:1224-1316` - Internal logic for market buy orders:

```
function _marketBuyInternal(uint256 maxQuoteIn, uint256 minBaseOut)
    internal
    returns (uint256 baseFilled, uint256 quoteUsed, uint256 takerFee)
{
    // ... matching loop ...

    baseFilled = acc.baseFilled;
    quoteUsed = acc.quoteFilled;

    if (baseFilled == 0) revert OrderBook__MarketDepthInsufficient(0, 0);
    if (quoteUsed > maxQuoteIn) revert OrderBook__SlippageExceeded();
    if (baseFilled < minBaseOut) revert OrderBook__SlippageExceeded();

    _collectQuoteCollateral(msg.sender, quoteUsed);
    takerFee = _applyMatchResults(msg.sender, true, acc);

    // ... rest of function ...
}
```

Line 1309: `if (baseFilled < minBaseOut) revert OrderBook__SlippageExceeded();` - This check uses `baseFilled` (gross amount).

`contracts/OrderBook.sol:1744-1748` - Taker fee is deducted from `baseFilledThisLevel` to calculate `takerReceiveBase` (net amount):

```
if (isTakerBuy) {
    uint256 takerFee = (baseFilledThisLevel * takerFeeRate) / FEE_DENOM;
    uint256 takerReceiveBase = baseFilledThisLevel > takerFee ? baseFilledThisLevel - takerFee : 0;
    acc.takerBaseFee += takerFee;
    acc.wBase += takerReceiveBase;
}
```

OrderBook.sol - `_applyMatchResults` function:

`contracts/OrderBook.sol:1625-1657` - Applies accumulated taker and maker deltas:

```
function _applyMatchResults(address takerAddress, bool isTakerBuy, MatchAccumulator memory acc)
    internal
    returns (uint256 takerFeePaid)
{
    if (acc.baseFilled == 0) return 0;

    takerFeePaid = isTakerBuy ? acc.takerBaseFee : acc.takerQuoteFee;

    uint256 wBase = acc.wBase;
    if (wBase > 0) {
        withdrawableBase[takerAddress] += wBase;
    }

    // ... rest of function ...
}
```

Lines 1633-1636: `withdrawableBase[takerAddress] += wBase;` - `wBase` is the net amount after fees.

Mechanism:

1. User calls `marketBuy(maxQuoteIn, minBaseOut)` expecting to receive at least `minBaseOut` base tokens
2. Matching fills `baseFilled` amount (gross, before fees)
3. Slippage check: `if (baseFilled < minBaseOut)` - passes if gross fill meets minimum
4. Taker fee applied: `wBase = baseFilled - takerFee` (net amount credited to user)
5. User receives less than `minBaseOut` if `baseFilled - takerFee < minBaseOut`

Impact

User Experience Violation:

- Users set `minBaseOut` expecting to receive at least that amount net-of-fees
- Function succeeds but user receives less than expected
- Violates documented behavior and user expectations

Financial Impact:

- Users may receive significantly less than their slippage protection threshold
- With 0.5% taker fee, user setting `minBaseOut = 100` can receive 99.5 base tokens
- Impact scales with fee rate and order size

Invariant Conflict:

- Invariant 1 states: taker's withdrawable base equals `baseFilled - takerFee`
- Invariant 3 states: slippage guard protects minimum received
- Current implementation checks gross amount, violating the intent of Invariant 3

Remediation

Check Net Amount After Fee Application:

Modify `_marketBuyInternal()` to check the net amount after fees:

```
function _marketBuyInternal(uint256 maxQuoteIn, uint256 minBaseOut)
    internal
    returns (uint256 baseFilled, uint256 quoteUsed, uint256 takerFee)
{
    // ... existing matching logic ...

    baseFilled = acc.baseFilled;
    quoteUsed = acc.quoteFilled;

    if (baseFilled == 0) revert OrderBook__MarketDepthInsufficient(0, 0);
    if (quoteUsed > maxQuoteIn) revert OrderBook__SlippageExceeded();

    // Apply fees first, then check net amount
    takerFee = _applyMatchResults(msg.sender, true, acc);
    uint256 netBaseReceived = acc.wBase; // This is baseFilled - takerFee

    if (netBaseReceived < minBaseOut) revert OrderBook__SlippageExceeded();

    _collectQuoteCollateral(msg.sender, quoteUsed);

    return (baseFilled, quoteUsed, takerFee);
}
```

Alternative: Adjust Min-out for Expected Fee:

Calculate expected fee and adjust the check:

```
uint256 expectedTakerFee = (baseFilled * takerFeeRate) / FEE_DENOM;
uint256 netBaseAfterFee = baseFilled - expectedTakerFee;
if (netBaseAfterFee < minBaseOut) revert OrderBook__SlippageExceeded();
```

POC

Attack Scenario:

1. User wants to buy base tokens with slippage protection
2. User calls `marketBuy(maxQuoteIn=10000, minBaseOut=100)`
    - Expects to receive at least 100 base tokens net-of-fees
    - Function comment implies “Minimum base amount to receive”
3. Order matching fills 100 base tokens (gross)
    - `baseFilled = 100`
    - Slippage check: `100 >= 100` -> passes
4. Taker fee applied (0.5% = 0.5 base tokens)
    - `takerFee = (100 * 5000) / 1_000_000 = 0.5`
    - `wBase = 100 - 0.5 = 99.5` base tokens credited to user
5. User receives 99.5 base tokens
    - **Violation:** User expected at least 100 base tokens but received 99.5
    - Transaction succeeded despite receiving less than `minBaseOut`

Code Evidence:

The vulnerability is demonstrated by `_marketBuyInternal()` checking `baseFilled < minBaseOut` at line 471 before applying the taker fee at line 475. The check uses the gross fill amount, but the user actually receives `baseFilled - takerFee` as shown in `_matchAtPriceLevel()` where `acc.wBase += baseFilledThisLevel - takerFee`. This creates a semantic mismatch where the slippage guard passes but the user receives less than the minimum they specified.

Disclaimer

This security report (“Report”) is provided by FailSafe (“Tester”) for the exclusive use of the client (“Client”). The scope of this assessment is limited to the security testing services performed against the systems, applications, or environments supplied by the Client. This Report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer, and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to you (“Customer” or the “Company”) in connection with the Agreement. This Report, provided in connection with the Services set forth in the Agreement, shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This Report may not be transmitted, disclosed, referred to, or relied upon by any person for any purpose, nor may copies be delivered to any other person other than the Company, without FailSafe’s prior written consent in each instance.

This Report is not, nor should it be considered, an “endorsement” or “disapproval” of any particular project, system, or team. This Report is not, nor should it be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts FailSafe to perform security testing. This Report does not provide any warranty or guarantee regarding the absolute security or bug-free nature of the technology analyzed, nor does it provide any indication of the technology’s proprietors, business, business model, or legal compliance.

This Report should not be used in any way to make decisions around investment or involvement with any particular project. This Report in no way provides investment advice, nor should it be leveraged as investment advice of any sort. This Report represents an extensive testing process intended to help our customers identify potential security weaknesses while reducing the risks associated with complex systems and emerging technologies.

Technology systems, applications, and cryptographic assets present a high level of ongoing risk. FailSafe’s position is that each company and individual are responsible for their own due diligence and continuous security practices. FailSafe’s goal is to help reduce attack vectors and the high level of variance associated with utilizing new and evolving technologies, and in no way claims any guarantee of security or functionality of the systems we agree to test.

The security testing services provided by FailSafe are subject to dependencies and are under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. The testing process may include false positives, false negatives, and other unpredictable results. The services may access and depend upon multiple layers of third-party technologies.

ALL SERVICES, THE LABELS, THE TESTING REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF ARE PROVIDED “AS IS” AND “AS AVAILABLE” AND WITH ALL FAULTS AND DEFECTS WITHOUT WARRANTY OF ANY KIND. TO THE MAXIMUM EXTENT PERMITTED UNDER APPLICABLE LAW, FAILSAFE HEREBY DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE WITH RE-

SPECT TO THE SERVICES, TESTING REPORT, OR OTHER MATERIALS. WITHOUT LIMITING THE FOREGOING, FAILSAFE SPECIFICALLY DISCLAIMS ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT, AND ALL WARRANTIES ARISING FROM COURSE OF DEALING, USAGE, OR TRADE PRACTICE. WITHOUT LIMITING THE FOREGOING, FAILSAFE MAKES NO WARRANTY OF ANY KIND THAT THE SERVICES, THE LABELS, THE TESTING REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF, WILL MEET CUSTOMER'S OR ANY OTHER PERSON'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULT, BE COMPATIBLE OR WORK WITH ANY SOFTWARE, SYSTEM, OR OTHER SERVICES, OR BE SECURE, ACCURATE, COMPLETE, FREE OF HARMFUL CODE, OR ERROR-FREE.

WITHOUT LIMITATION TO THE FOREGOING, FAILSAFE PROVIDES NO WARRANTY OR DISCLAIMER UNDERTAKING, AND MAKES NO REPRESENTATION OF ANY KIND THAT THE SERVICE WILL MEET CUSTOMER'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULTS, BE COMPATIBLE OR WORK WITH ANY OTHER SOFTWARE, APPLICATIONS, SYSTEMS OR SERVICES, OPERATE WITHOUT INTERRUPTION, MEET ANY PERFORMANCE OR RELIABILITY STANDARDS OR BE ERROR FREE OR THAT ANY ERRORS OR DEFECTS CAN OR WILL BE CORRECTED.

WITHOUT LIMITING THE FOREGOING, NEITHER FAILSAFE NOR ANY OF FAILSAFE'S AGENTS MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND, EXPRESS OR IMPLIED AS TO THE ACCURACY, RELIABILITY, OR CURRENCY OF ANY INFORMATION OR CONTENT PROVIDED THROUGH THE SERVICE. FAILSAFE WILL ASSUME NO LIABILITY OR RESPONSIBILITY FOR (I) ANY ERRORS, MISTAKES, OR INACCURACIES OF CONTENT AND MATERIALS OR FOR ANY LOSS OR DAMAGE OF ANY KIND INCURRED AS A RESULT OF THE USE OF ANY CONTENT, OR (II) ANY PERSONAL INJURY OR PROPERTY DAMAGE, OF ANY NATURE WHATSOEVER, RESULTING FROM CUSTOMER'S ACCESS TO OR USE OF THE SERVICES, TESTING REPORT, OR OTHER MATERIALS.

ALL THIRD-PARTY MATERIALS ARE PROVIDED "AS IS" AND ANY REPRESENTATION OR WARRANTY OF OR CONCERNING ANY THIRD-PARTY MATERIALS IS STRICTLY BETWEEN CUSTOMER AND THE THIRD-PARTY OWNER OR DISTRIBUTOR OF THE THIRD-PARTY MATERIALS.

THE SERVICES, TESTING REPORT, AND ANY OTHER MATERIALS HEREUNDER ARE SOLELY PROVIDED TO CUSTOMER AND MAY NOT BE RELIED ON BY ANY OTHER PERSON OR FOR ANY PURPOSE NOT SPECIFICALLY IDENTIFIED IN THIS AGREEMENT, NOR MAY COPIES BE DELIVERED TO ANY OTHER PERSON WITHOUT FAILSAFE'S PRIOR WRITTEN CONSENT IN EACH INSTANCE.

NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH SERVICES, TESTING REPORT, AND ANY ACCOMPANYING MATERIALS AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST FAILSAFE WITH RESPECT TO SUCH SERVICES, TESTING REPORT, AND ANY ACCOMPANYING MATERIALS.

THE REPRESENTATIONS AND WARRANTIES OF FAILSAFE CONTAINED IN THIS AGREEMENT ARE SOLELY FOR THE BENEFIT OF CUSTOMER. ACCORDINGLY, NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH REPRESENTATIONS AND WARRANTIES AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST FAILSAFE WITH RESPECT TO SUCH REPRESENTATIONS OR WARRANTIES OR ANY MATTER SUBJECT TO OR RESULTING IN INDEMNIFICATION UNDER THIS AGREEMENT OR OTHERWISE.

FOR AVOIDANCE OF DOUBT, THE SERVICES, INCLUDING ANY ASSOCIATED TESTING REPORTS OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.