Alpha Sec. - audit

Security Assessment

CertiK Assessed on Dec 19th, 2025

CertiK Assessed on Dec 19th, 2025

Alpha Sec. - audit

The security assessment was prepared by CertiK.

Executive Summary

TYPES	ECOSYSTEM	METHODS
DEX, Layer 2	EVM Compatible	Manual Review, Static Analysis

LANGUAGE	TIMELINE
Go, Solidity	Preliminary comments published on 11/14/2025 Final report published on 12/20/2025

Vulnerability Summary

67	54	0	13	0
Total Findings	Resolved	Partially Resolved	Acknowledged	Declined

0	Centralization		Centralization findings highlight privileged roles & functions and their capabilities, or instances where the project takes custody of users' assets.
0	Critical		Critical risks are those that impact the safe functioning of a platform and must be addressed before launch. Users should not invest in any project with outstanding critical risks.
7	Major	7 Resolved	Major risks may include logical errors that, under specific circumstances, could result in fund losses or loss of project control.
13	Medium	12 Resolved, 1 Acknowledged	Medium risks may not pose a direct risk to users' funds, but they can affect the overall functioning of a platform.
35	Minor	28 Resolved, 7 Acknowledged	Minor risks can be any of the above, but on a smaller scale. They generally do not compromise the overall integrity of the project, but they may be less efficient than other solutions.
12	Informational	7 Resolved, 5 Acknowledged	Informational errors are often recommendations to improve the style of the code or certain operations to fall within industry best practices. They usually do not affect the overall functioning of the code.

TABLE OF CONTENTS | ALPHA SEC. - AUDIT

CODEBASE | ALPHA SEC. - AUDIT

Repository

https://github.com/kaiachain/go-ethereum

https://github.com/kaiachain/kaia-orderbook-dex-core

https://github.com/kaiachain/kaia-orderbook-dex-core-contracts

https://github.com/kaiachain/kaia-orderbook-dex-token-bridge-contracts

Commit

6101af6996bf7b18cc86c89fae7bb0425663fc24

188b1089712be2a547433a584d1813f03e2ca6e8

6bb9e9eeeef6aabeccbef1f608618e1ea2f00737

5aed5069b5b13e120eae06a58a53303dece1ea33

Audit Scope

The file in scope is listed in the appendix.

APPROACH & METHODS | ALPHA SEC. - AUDIT

This audit was conducted for Kaia to evaluate the security and correctness of the smart contracts associated with the Alpha Sec. - audit project. The assessment included a comprehensive review of the in-scope smart contracts. The audit was performed using a combination of Manual Review and Static Analysis.

The review process emphasized the following areas:

- Architecture review and threat modeling to understand systemic risks and identify design-level flaws.
- Identification of vulnerabilities through both common and edge-case attack vectors.
- Manual verification of contract logic to ensure alignment with intended design and business requirements.
- Dynamic testing to validate runtime behavior and assess execution risks.
- Assessment of code quality and maintainability, including adherence to current best practices and industry standards.

The audit resulted in findings categorized across multiple severity levels, from informational to critical. To enhance the project's security and long-term robustness, we recommend addressing the identified issues and considering the following general improvements:

- Improve code readability and maintainability by adopting a clean architectural pattern and modular design.
- Strengthen testing coverage, including unit and integration tests for key functionalities and edge cases.
- Maintain meaningful inline comments and documentations.
- Implement clear and transparent documentation for privileged roles and sensitive protocol operations.
- Regularly review and simulate contract behavior against newly emerging attack vectors.

OVERVIEW | ALPHA SEC. - AUDIT

Introduction

The **Alpha Sec.** (Kaia Orderbook DEX) is a high-performance decentralized exchange (DEX) implemented as a protocol-level extension on a customized Layer 2 (L2) chain. Its primary objective is to provide an on-chain orderbook trading environment with low latency, high throughput, and minimal costs, rivaling the performance of centralized exchanges (CEX).

The project's core innovation lies in its protocol-native design philosophy. Unlike traditional DEXs built entirely on smart contracts, Alpha Sec. integrates critical trade processing functions—such as order matching, balance management, and conditional order handling—directly into the client execution logic of the Arbitrum Nitro L2 node (a fork of Geth). This design allows computationally intensive operations to bypass the performance bottlenecks of the EVM, enabling significant performance gains. The blockchain's role is transformed into a highly optimized, application-specific state machine, while retaining the security guarantees provided by the Arbitrum Rollup framework.

To further optimize user experience and cater to high-frequency trading scenarios, the protocol introduces several foundational account model extensions, including native multi-token accounts, session key delegation, and an enhanced nonce mechanism.

The scope of current engagement mainly focuses on the following 3 components:

1. Orderbook Based Matching Engine
2. Deposit & Withdrawal Processing
3. Session Wallet Usage

Note: Per the Alpha Sec. team's request, the current report has been redacted, including the sections covering the finding description, potential scenarios, proof of concept, recommendations, and remediation details.

Architecture & Key Innovations

Architectural Overview

The Alpha Sec. employs a deeply integrated, layered architecture that moves core trading functions from the EVM application layer down to the L2 protocol's execution layer. This design is intended to minimize overhead and enable direct, high-efficiency communication between components. The system architecture can be divided into five logical layers:

- **Foundation Layer**: Defines the core data structures (e.g., `Order`, `Trade`, `StopOrder`) and behavioral contracts (interfaces) for the entire orderbook system.
- **Core Logic Layer**: Contains the concrete implementations of the orderbook (`OrderBook`), priority queues (`Buy/SellQueue`), and the matching algorithm (`PriceTimePriority`).
- **Business Logic Layer**: Orchestrates the core logic components to form complete business functions. At this layer, the `SymbolEngine` orchestrates matching, conditional order processing, and balance operations for each trading pair.
- **Persistence Layer**: Manages the system's state snapshots and recovery logic, ensuring data consistency after a node restart or crash.

- **External Interface Layer**: Serves as the system's main entry point. The `Dispatcher` at this layer manages all `SymbolEngine` instances, handles asynchronous requests from users, and coordinates with the on-chain state and balance management modules.

Key Innovations

The protocol achieves its core functionality through several key modifications to the underlying Arbitrum Nitro stack:

- **Protocol-Level Orderbook**: Order matching logic is executed natively by the modified L2 node client (a fork of Geth) rather than through EVM smart contracts. This allows order processing to avoid EVM overhead, aiming for ultra-low latency.
- **Command-Based Transactions**: Users' DEX operations (e.g., placing or canceling orders) are encoded as specific commands, encapsulated within standard Ethereum transactions, and sent to a dedicated address (`0x...cc`) for interception and native processing, without introducing new transaction types for DEX operations.
- **Unified Token System**: The account state is extended at the protocol level to natively support balances for multiple tokens. Internal asset transfers and trade settlements directly modify this underlying state, bypassing the EVM and significantly reducing operational costs.
- **Session Delegation**: Introduces temporary, time-limited "session keys" that allow users to grant one-time authorization for continuous, high-frequency trading within a dApp, eliminating the need to sign every transaction and thus optimizing the user experience.
- **Enhanced Nonce System**: Combines the standard State Nonce with a Time Nonce, which is based on millisecond-level timestamps and designed for session keys, to support high concurrency and a degree of out-of-order transaction processing while maintaining security.

Core Components & Workflows

This section details the core components that constitute the Alpha Sec. and describes their roles and interactions in processing the transaction lifecycle.

Account Model & State Extensions

To support the native orderbook functionality, the standard Ethereum account model has been extended at the protocol layer with several key fields:

- **`Balances` (Native Multi-Token Balances)**: Each account contains a `Balances` structure to store the balances of multiple native tokens. This structure implements a dual-balance system:
  - **`Available`** : Funds that can be used to place new orders or make transfers.
  - **`Locked`** : Margin that has been locked by active orders and cannot be used for other operations. This model is managed by the `balance.Manager` module, which ensures atomic operations for balance changes during order placement, settlement (including for partial fills), and cancellation.

- **`Sessions` (Session List)**: This supports the session key delegation feature. Each account can be associated with one or more `Session` objects, each defining a delegated temporary public key (`PublicKey`) and its expiration (`ExpiresAt` block number). Session lifecycle management is handled via `SessionContext` commands, which require an EIP-712 signature from the main account (`L1Owner`).

- `TimeNonce` **(Timestamp Nonce List)**: This is a cache list designed to support high-concurrency transactions for session keys. It stores recently used nonces, which are based on millisecond-level timestamps, to prevent replay attacks while allowing for a degree of out-of-order transaction processing.

Command-Based Transaction System

All DEX-related operations are executed through a command system that leverages standard Ethereum transactions as carriers.

- **Entry Point**: A user sends a standard transaction to a predefined contract address `0x...cc`.
- **Command Format**: The transaction's `Input Data` is formatted as `[Command Byte] + [Serialized Data]`. The `Command Byte` identifies the operation type, and the `Serialized Data` contains the specific parameters encoded in JSON.
- **Core Commands**:
  - `SessionContext`: Used to manage session keys.
  - `ValueTransferContext`: Used to execute internal L2 native token (Kaia) transfers.
  - `TokenTransferContext`: Used to execute internal L2 ERC20 token transfers.
  - `OrderContext`: Used to submit new orders, supporting limit, market, Base/Quote Mode, and optional TPSL settings.
  - `CancelContext` / `CancelAllContext`: Used to cancel a single or all active orders.
  - `ModifyContext`: Used to modify the price or quantity of an existing order.
  - `StopOrderContext`: Used to submit standalone conditional orders.
- **Validation Flow**: Before execution, every command is rigorously validated by the `validate` method defined in `tx_input.go`. This includes checks for parameter sanity, business logic consistency (e.g., TPSL price relationships), market rules (price/quantity precision), and user balances.

Asset Bridging (ArbTokenIssuer)

The deposit and withdrawal of assets are managed by a precompiled contract named `ArbTokenIssuer`, deployed at address `0xdf`.

- **Deposit (`Mint`)**:
  1. When a user deposits an ERC20 token via the L1 gateway, an L1-to-L2 message triggers a call to the L2 gateway contract.
  2. The L2 gateway contract calls the `mint` function of the precompile.
  3. The Go implementation of `ArbTokenIssuer` performs strict permission checks (verifying the caller's code hash and the aliased address of the message sender) to ensure only messages from the official L1 gateway can execute a mint.
  4. For new tokens, the system automatically parses metadata from the `calldata`, registers the token, and emits a `TokenRegistered` event.

5. Finally, `ArbTokenIssuer` calls the `StateDB` interface to add the corresponding native balance to the user's account and emits a `TokenTransfer` event with `address(0)` as the `from` address.

- **Withdrawal (`Burn`)**:
  1. A user signs a transaction to call a withdrawal function on the L2 gateway contract.
  2. The L2 gateway contract calls the `burn` function of the precompile.
  3. `ArbTokenIssuer` verifies that the caller is an authorized gateway and, critically, validates that the transaction's signer (`msg.sender`) is the owner of the account from which assets are being burned.
  4. After checking for sufficient balance, `ArbTokenIssuer` calls the `StateDB` interface to subtract the native balance from the user's account and emits a `TokenTransfer` event with `address(0)` as the `to` address, signaling the L1 gateway to release the assets.

Core Orderbook Workflow

The orderbook system operates around a clear, layered architecture with the `Dispatcher` as the top-level coordinator.

1. **Request Dispatching (`Dispatcher`)**: The `Dispatcher` receives all commands via an asynchronous request channel (`requestChan`). Acting as the Balance Coordinator, it first calls the `balance.Manager` to pre-lock the margin for new orders. It then routes the request to the appropriate `SymbolEngine` instance based on the order's trading pair (`Symbol`).

2. **Per-Symbol Processing (`SymbolEngine`)**: The `SymbolEngine` serves as the business logic hub for a single trading pair. It receives an order and invokes the `matching.PriceTimePriority` module (the matching algorithm). The matching algorithm interacts with the `book.OrderBook` (the orderbook data structure) to perform matching and produce `Trade` records. After matching, the `SymbolEngine` checks if any conditional orders (like Stop Orders) were triggered by the latest trade price, or if a filled main order needs its associated TPSL to be activated. Triggered or activated orders are placed into an internal triggered order queue and are processed iteratively in a BFS (Breadth-First Search) manner within the same transaction to ensure atomicity of chained triggers.

3. **Settlement**: The `SymbolEngine` returns a list of generated `Trade`s to the `Dispatcher`. The `Dispatcher` iterates through the `Trade`s and calls the `balance.Manager`'s `SettleTrade` function for each one. `SettleTrade` performs the final clearing: it consumes the `Locked` balances of the buyer and seller, calculates and deducts fees, and then credits the net amounts to the respective parties' `Available` balances. `SettleTrade` performs the final clearing, a process that includes precise fee calculation and distribution.

Order Lifecycle & Processing Flow

Order processing in the Kaia DEX follows a well-defined, structured lifecycle designed to ensure atomicity, consistency, and high performance.

Order States

An order progresses through several core states during its lifecycle:

- `NEW`: The initial state when an order is created.

- `PENDING` : For a regular limit order, this state indicates the order has passed validation and is resting in the order book, awaiting a match. For a conditional order (`StopOrder` , `TPSL`), this state indicates the order is waiting for the market price to reach its trigger condition.
- `FILLED` : The order has been fully executed.
- `PARTIALLY_FILLED` : The order has been partially executed, with the remainder still resting in the order book.
- `CANCELED` : The order was actively canceled by the user.
- `REJECTED` : The order was rejected during order matching.
- `TRIGGERED` : A conditional order's trigger condition has been met, and it is being converted into a regular order to enter the matching flow.
- `TRIGGERED_WAIT` : A conditional order's trigger condition has not been met yet.

Regular Order Flow

The processing flow for a regular limit or market order is as follows:

1. **Validation**: Upon receiving an `OrderRequest` , the `Dispatcher` performs initial validation as defined in `tx_input.go` , checking the command format and business logic.
2. **Locking**: After validation, the `Dispatcher` calls the `balance.Manager` to calculate and lock the user's margin for the order (`Available` -> `Locked`).
3. **Routing & Matching**: The `Dispatcher` routes the order to the corresponding `SymbolEngine` . The `SymbolEngine` invokes the `matching.PriceTimePriority` algorithm, which matches the incoming order (Taker) against existing counter-party orders (Makers) in the `OrderBook` .
4. **Conditional Check**: After matching produces a new market price, the `SymbolEngine` calls the `conditional.Manager` to check if any pending conditional orders (Stop or SL orders) have been triggered.
5. **Settlement**: The `SymbolEngine` returns the execution records (`Trade` s) to the `Dispatcher` . The `Dispatcher` then calls `balance.Manager.SettleTrade` to perform clearing, which includes consuming the `Locked` balances of both parties, calculating fees, and crediting the net amounts to their `Available` balances.
6. **Queue Update**: If the Taker order was a limit order and was not fully filled, its remaining portion is added to the appropriate buy/sell queue in the `OrderBook` , becoming a new Maker order. The remainder of a market order is canceled.
7. **Post-processing**: For orders that are fully filled or canceled, the `balance.Manager` releases any remaining locks, and the `Dispatcher` cleans up the order's information from its internal caches.

`TPSL` **Order (Attached Conditional Orders)**: The processing of a `TPSL` is a multi-stage activation flow that occurs after its parent order is filled:

1. **Activation**: A main order with a `TPSL` field is fully filled (`FILLED`). The `SymbolEngine` detects this and calls the `conditional.Manager` 's `CreateTPSLForFilledOrder` method.
2. **Decomposition**: The `activation_rule` module decomposes the TPSL context into two separate child orders, whose quantities are set to the original quantity of the parent order:

- **Take-Profit (TP) Order**: As a regular, opposite-side limit order.

- **Stop-Loss (SL) Order**: As a conditional order encapsulated in a `StopLossTrigger`.

3. **Routing**: The `conditional.Manager` routes the decomposed orders to different destinations via callbacks: The newly created `TP` limit order is received by the `orderProcessor` callback and is placed directly into the main order book as an active resting order. The newly created `SL` conditional order is registered with the `triggerManager` to await its price trigger.
4. **OCO Relationship Binding**: The `oco_controller` module registers the "One-Cancels-the-Other" (OCO) relationship between the `TP` and `SL` orders.
5. **Execution & Finalization**: Subsequently, when either the `TP` limit order is matched in the main orderbook or the `SL` conditional order is triggered by price in the `triggerManager`, the `oco_controller` is invoked to automatically cancel the other outstanding order, thus concluding the TPSL's lifecycle.

Conditional Order Flow

`StopOrder` (Standalone Conditional Order):

1. A user submits a `StopOrderContext`.
2. The `Dispatcher` validates it and fully locks the margin that would be required for its future execution.
3. The order is encapsulated into a `StopOrderTrigger` object and registered with the `conditional.Manager`'s `triggerManager`, entering the `PENDING (conditional)` state.
4. After each new trade occurs in the market, the `SymbolEngine` calls `conditional.Manager.CheckTriggers` with the latest market price.
5. If the `StopPrice` is met, the `triggerManager` changes the `StopOrderTrigger`'s state to `TRIGGERED`, converts it into a regular order, and places it in the `SymbolEngine`'s processing queue, where it enters the regular order flow.

Persistence & State Recovery

Orderbook v2 durability hinges on periodic full snapshots plus per-block deltas managed by `PersistenceManager`, written via `SnapshotManager` / `DeltaWriter`, and replayed by `RecoveryEngine` so the dispatcher, engines, and balance locks always return to their pre-crash state.

- **PersistenceManager.WriteSnapshot** captures an immutable copy of all dispatcher + engine state at the configured block interval and queues it for synchronous or async storage through `SnapshotManager`, based on the active `OrderbookConfig`.
- After each block, Dispatcher hands the block's `types.DispatcherDelta` to `DeltaWriter`, which serializes it under `orderbook-delta-<block>`; sync mode writes immediately, while async mode batches via a background goroutine before committing to `ethdb`.
- On restart, **RecoveryEngine.RecoverUpToBlock** finds the most recent snapshot at or before the target height, restores it via **dispatcher.RestoreFromSnapshot**, then streams every stored delta in order until the dispatcher reflects the desired block.
- Once replay completes, the dispatcher resumes live processing with the exact same order books, conditional queues, and balance locks that existed before the outage, ensuring no accepted command is lost.

Summary

The architecture of the Alpha Sec. presents a well-considered, protocol-native trading system designed for high performance and low latency. By moving the core functionalities of an orderbook—such as matching, balance management, and conditional orders—from the EVM application layer down to the L2 client's execution layer, the project aims to fundamentally address the performance bottlenecks of traditional on-chain DEXs.

Overall, the Alpha Sec. architecture is a complex and deeply customized L2 solution. It deliberately trades some EVM generality for ultimate performance in the specific domain of trading. Its workflows and component designs reflect an adoption of modern centralized exchange architectural patterns, creatively combined with the decentralized nature of the blockchain.

Reference

- https://docs.alphasec.trade/
- Internal design documentation
- https://docs.arbitrum.io/get-started/overview

FINDINGS | ALPHA SEC. - AUDIT

67	0	0	7	13	35	12
Total Findings	Critical	Centralization	Major	Medium	Minor	Informational

This report has been prepared for Kaia to identify potential vulnerabilities and security issues within the reviewed codebase. During the course of the audit, a total of 67 issues were identified. Leveraging a combination of Manual Review & Static Analysis the following findings were uncovered:

ID	Title	Category	Severity	Status
ASA-125	Missing Ownership Validation In Order Cancellation	Inconsistency, Denial of Service	Major	Resolved
ASA-68	Unrestricted `Session.Metadata` Field Enables Potential DoS Attack	Denial of Service	Major	Resolved
ASA-69	TPSL Lock Logic May Fail Due To Premature Locking Of Unsettled Assets	Logical Issue	Major	Resolved
ASA-70	`GetOrdersSorted()` Corrupts Original Queue	Logical Issue	Major	Resolved
ASA-71	Market Order Locking Allows DoS Via Insufficient Balance	Design Issue	Major	Resolved
ASA-72	Unbounded Wallet Sessions Enable Denial Of Service	Denial of Service	Major	Resolved
ASA-73	Missing Handling Of `FailedOrders` In `ModifyOrder()`	Inconsistency, Logical Issue	Major	Resolved
ASA-74	Balance Manager Records Locks Even When State Locking Fails	Coding Issue	Medium	Resolved
ASA-75	Inconsistent Order State Due To Incorrect Lock Amount Update	Coding Issue	Medium	Resolved
ASA-76	Improper Locking Order (Race Condition) In `Lock()`	Logical Issue	Medium	Resolved
ASA-77	Potential Transaction Bloat Attack Due To Trailing Bytes	Denial of Service	Medium	Resolved

ID	Title	Category	Severity	Status
ASA-78	Ambiguous Quantity Semantics Between Quote And Base Tokens	Coding Style	Medium	Resolved
ASA-79	Lot-Size/Dust Validation Bypass For SELL Market Orders In Quote Mode After Lock Limiting	Logical Issue	Medium	Resolved
ASA-80	Incorrect Unlock Identifier In `createTPSLOrders` May Cause Stuck TPSL Locks And Balance Inconsistency	Inconsistency, Logical Issue	Medium	Resolved
ASA-81	Discussion On Gas-Free Dex Commands Design That Enables Multi-Layer DoS	Design Issue, Denial of Service	Medium	Acknowledged
ASA-82	Both TP Order And SL Orders Could Exist In Orderbook In Some Edge Cases	Design Issue	Medium	Resolved
ASA-83	TriggeredQueue Not Restored From Engine Snapshot	Coding Issue	Medium	Resolved
ASA-84	Incorrect Value Copy During Aggregation Leads To Erroneous Market Depth	Logical Issue	Medium	Resolved
ASA-85	Non-Atomic TPSL Creation Can Lead To Orphaned Orders And Inconsistent State	Logical Issue	Medium	Resolved
ASA-86	Funds Unlocked Before Order Removal In `handleCancelAllRequest()`	Volatile Code, Denial of Service	Medium	Resolved
ASA-100	Potential Exploitation Of SL Market Orders Via Extreme Price Updates	Design Issue	Minor	Acknowledged
ASA-101	Unsynchronized And Unvalidated Metadata Persistence In `Stop()` Causes WAL Inconsistency	Logical Issue	Minor	Resolved
ASA-102	Potential Overflow Leads To Panic With `MustFromBig()`	Volatile Code	Minor	Resolved
ASA-103	Insufficient Constraint On Data Size	Volatile Code	Minor	Resolved
ASA-104	Expired Session Wallet Does Not Fail	Inconsistency	Minor	Resolved

ID	Title	Category	Severity	Status
ASA-105	Missing Nonzero Check Of Input `data`	Volatile Code	Minor	Acknowledged
ASA-106	Dispatcher Panics On Shutdown If New Requests Arrive After `Stop()`	Denial of Service, Volatile Code	Minor	Resolved
ASA-107	Missing `LockedAmount` In Deep Copy Method `Copy()` Of `Order`	Volatile Code, Inconsistency	Minor	Resolved
ASA-108	`validate()` Misses Validation Of `OrderMode`	Volatile Code, Inconsistency	Minor	Resolved
ASA-109	Missing Validation Of Existing Order In `validate()` Of ModifyContext	Volatile Code, Inconsistency	Minor	Resolved
ASA-110	Incorrect Order Of Return Values In `GetOrderbookSnapshot()`	Logical Issue	Minor	Resolved
ASA-111	Non-Determinism Due To Map Iteration	Volatile Code, Inconsistency	Minor	Acknowledged
ASA-112	Time-Nonce Validation Could Possibly Be Bypassed In `timeNonceDriftAcceptable()`	Inconsistency, Volatile Code	Minor	Resolved
ASA-113	Missing Validation Of Existing Order In `validate()` Of `CancelAllContext`	Volatile Code	Minor	Resolved
ASA-114	Missing Deep Copy Of Order Information In `CreateModifiedOrder()`	Volatile Code	Minor	Resolved
ASA-115	Async Delta Loss Due To Premature Dirty-Flag Reset	Volatile Code, Inconsistency	Minor	Acknowledged
ASA-126	Non-Atomic Lock Consumption Can Leave Balances Partially Consumed On Failure	Volatile Code	Minor	Resolved
ASA-127	Discussion On Logged Settlement And OCO Failures Without Proper Handling	Inconsistency	Minor	Resolved
ASA-128	Missing Comparison Between `SLLimit` And `SLTrigger`	Logical Issue	Minor	Acknowledged

ID	Title	Category	Severity	Status
ASA-129	Stale Depth From In-Place Order Mutation In `UpdateOrder()`	Volatile Code	Minor	Resolved
ASA-130	TPSL Creation Failure Leaves Stale Pre-Registered TP/SL Routes	Inconsistency	Minor	Resolved
ASA-131	Market Orders Accept Negative Prices	Volatile Code	Minor	Resolved
ASA-87	Order Lock Can Be Removed When `oldOrderID` Equals `NewOrderID`	Volatile Code, Logical Issue	Minor	Resolved
ASA-88	Missing Nil Pointer Check In `Copy()` Of `ValueTransferContext`	Volatile Code, Coding Issue	Minor	Resolved
ASA-89	Unsafe Internal Pointer Exposure Via `GetBuyOrders()`	Logical Issue	Minor	Acknowledged
ASA-90	Order Quantity And Price Validation Uses `IsZero()` Instead Of `Sign()` To Ensure Strict Positivity	Volatile Code	Minor	Resolved
ASA-91	Reversed Conditional In `TokenTransferContext.copy()`	Logical Issue	Minor	Resolved
ASA-92	Missing Copy Of `LockedBalance` In `Copy()` Of `StateAccount`	Inconsistency	Minor	Resolved
ASA-93	Missing `LockedBalance` In `Account`	Inconsistency	Minor	Resolved
ASA-94	Non-Deterministic `MarshalJSON()` Of `Balances`	Inconsistency	Minor	Resolved
ASA-95	Mutable Aliasing In `NewOrder()` Allows Caller Modify `price/quantity/TPSL` After Order Creation	Logical Issue	Minor	Resolved
ASA-96	FILLED Orders Can Be Reactivated	Logical Issue	Minor	Resolved
ASA-97	`AllOrNone` OCO Strategy Incorrectly Implemented — Behaves Same As `OneCancelsOther`	Inconsistency	Minor	Resolved

ID	Title	Category	Severity	Status
ASA-98	Invalid State Transition In `TPSLOrder.Cancel()`	Logical Issue	Minor	Acknowledged
ASA-99	Missing Check In `MakeTimeNonceError()` Function	Logical Issue, Inconsistency	Minor	Resolved
ASA-116	Incorrect `fromAmount` Logging In `TransformLock()` Function	Logical Issue	Informational	Resolved
ASA-117	Incorrect Error Messages In `validate()`	Inconsistency	Informational	Resolved
ASA-118	Discussion On Missing `Metadata` In Signing Message	Inconsistency	Informational	Resolved
ASA-119	Discussion On Non-Functional WAL Manager Initialization	Logical Issue	Informational	Resolved
ASA-120	Discussion On Logging Errors Without Return	Design Issue, Coding Issue	Informational	Acknowledged
ASA-121	Discussion On Order Cleanup After Trade Settlement Failure	Coding Issue	Informational	Acknowledged
ASA-122	Duplicate `OrderType` Check In `validate()` Of `OrderContext` And `StopOrderContext`	Code Optimization	Informational	Resolved
ASA-123	Discussion On Latest Traded Price Updated As Orderbook's Price	Design Issue	Informational	Acknowledged
ASA-124	Discussion On Incomplete Stage Logic	Coding Issue	Informational	Acknowledged
ASA-132	Missing Checks In `Copy()` Of `StopOrder` And `TPSLOrder`	Volatile Code	Informational	Resolved
ASA-133	Missing Nil Check Of Trade In `processTradesAndCleanup()`	Volatile Code	Informational	Resolved
ASA-67	Discussion On Any Token That Is Pre-Registered	Logical Issue, Inconsistency	Informational	Acknowledged

ASA-125 | Missing Ownership Validation In Order Cancellation

Category	Severity	Location	Status
Inconsistency, Denial of Service	Major	core/types/tx_input.go (go-ethereum-6101af6): 572	Resolved

Description

`CancelContext.validateBalance()` only checks for the existence of the order and does not verify that the order belongs to the provided L1Owner. There is no subsequent ownership enforcement in `dispatcher.handleCancelRequest` or `engine.CancelOrder`, so any account can cancel another user's order by supplying its orderId.

Recommendation

Fetch the order by ID and verify order.UserID matches L1Owner before returning success.

Alleviation

[Kaia, 12/11/2025]:
Issue acknowledged. Changes have been reflected in the commit `de68b010ccb1ad77c6f89f64445fc56e65948489`.

ASA-68 Unrestricted `Session.Metadata` Field Enables Potential DoS Attack

Category	Severity	Location	Status
Denial of Service	Major	core/types/session.go (go-ethereum-6101af6): 28	Resolved

Description

The `Metadata` field in the `Session` struct of the session DEX command transaction does not appear to be utilized and restricted during the transaction process.

Recommendation

Recommend adding size validation of `Metadata` during the transaction validation or removing it if it's not intended to be used in the codebase.

Alleviation

[Kaia, 10/30/2025]:
Issue acknowledged. Changes have been reflected in the commit `feca04c6d0945fce0d24e7e8c317a65f323f1375` .

ASA-69 TPSL Lock Logic May Fail Due To Premature Locking Of Unsettled Assets

Category	Severity	Location	Status
Logical Issue	Major	core/orderbook/v2/dispatcher/dispatcher.go (go-ethereum-6101af6): 366; core/orderbook/v2/engine/symbol_engine.go (go-ethereum-6101af6): 814	Resolved

Description

The `TPSL` locking mechanism assumes that the original order's settlement has already been completed, meaning the user has received the traded tokens (base tokens for a `BUY` order or quote tokens for a `SELL` order).

Recommendation

Defer TPSL lock creation until after trade settlement is completed.

Alleviation

[Kaia, 11/26/2025]:
Issue acknowledged. Changes have been reflected in the commits `c52b8e253d3f3d465b419b392d8e08e1fa495738` and `7a5c1fa2d15d273e505933a614b7c97e201c310c`.

ASA-70 | `GetOrdersSorted()` Corrupts Original Queue

Category	Severity	Location	Status
Logical Issue	Major	core/orderbook/v2/queue/buy_queue.go (go-ethereum-6101af6): 89~108; core/orderbook/v2/queue/sell_queue.go (go-ethereum-6101af6): 89~108	Resolved

Description

`GetOrdersSorted()` is supposed to be a read-only getter, but it mutates the original BuyQueue/SellQueue's orders. It creates a shallow copy of the orders slice, so the temporary heap operates on the exact same `*types.Order` objects as the original queue.

Recommendation

Sorting must not rely on or modify the heap.

Alleviation

[Kaia, 11/05/2025]:
Issue acknowledge. Changes have been reflected in commit `f7851a97ebb5b22debbfdad16a4866eebea0165d` .

ASA-71 | Market Order Locking Allows DoS Via Insufficient Balance

Category	Severity	Location	Status
Design Issue	Major	core/orderbook/v2/balance/manager.go (go-ethereum-6101af6): 325~326, 490	Resolved

Description

In `LockForOrder()`, when handling market orders, the function calls `calculateMarketOrderAmount` to determine the amount to lock. During execution, `m.Lock()` fails due to insufficient balance, causing the transaction to revert and not be included in the block.

Recommendation

Consider preventing these type of transactions during the transaction validation process.

Alleviation

[Kaia, 11/26/2025]:
Issue acknowledged. Changes have been reflected in the commit `9897543b4bcec2bfdff064941f0e5b3588076cc0`.

ASA-72 | Unbounded Wallet Sessions Enable Denial Of Service

Category	Severity	Location	Status
Denial of Service	Major	core/types/transaction.go (go-ethereum-6101af6): 448; arbos/tx_processor.go (dex-core-188b108): 549	Resolved

Description

The Kaia orderbook stack persists every wallet session directly in the account record without enforcing any per-address limit.

Recommendation

Enforce a strict per-address session cap.

Alleviation

[Kaia, 11/06/2025]:

Issue acknowledged. Changes have been reflected in the commit `ea27589bcb58d6c5222023a6ec0eb7287aea1543` .

ASA-73 | Missing Handling Of `FailedOrders` In `ModifyOrder()`

Category	Severity	Location	Status
Inconsistency, Logical Issue	Major	core/orderbook/v2/engine/symbol_engine.go (go-ethereum-6101af6): 675, 686~691	Resolved

Description

`ModifyOrder()` ignores the information of failed IDs, its `ModifyResult` only exposes `NewOrder`, `Trades`, `TriggeredOrderIds`, and `CancelledOrderIds`, but drops `FailedOrders`.

Recommendation

Expose the `FailedOrders` list through `ModifyResult` and ensure they are processed by the dispatcher.

Alleviation

[Kaia, 11/23/2025]:

Issue acknowledged. Changes have been reflected in the commit `912bee211bc712425e6d9730f58b758f91c5b332`.

ASA-74 Balance Manager Records Locks Even When State Locking Fails

Category	Severity	Location	Status
Coding Issue	Medium	core/orderbook/v2/balance/manager.go (go-ethereum-6101af6): 89, 147, 283, 769~772; core/state/state_object.go (go-ethereum-6101af6): 796; core/state/statedb.go (go-ethereum-6101af6): 582~587	Resolved

Description

The `UpdateLockForTriggeredMarketOrder()` persists `LockInfo` entries after calling `StateDB.LockTokenBalance()` without checking the returned error. The state layer also drops the error from `stateObject.LockTokenBalance()`.

Recommendation

Propagate the error returned by `stateObject.LockTokenBalance()` (and `ConsumeLockTokenBalance()`) at the StateDB level.

Alleviation

[Kaia, 11/26/2025]: Issue acknowledge. Changes have been reflected in the commit `30a767da95b547cbefe3077364dc72e29871699f`.

ASA-75 | Inconsistent Order State Due To Incorrect Lock Amount Update

Category	Severity	Location	Status
Coding Issue	Medium	core/orderbook/v2/balance/manager.go (go-ethereum-6101af6): 403~415	Resolved

Description

The `UpdateLockForTriggeredMarketOrder()` function is designed to increase the locked balance for a triggered stop-market order by securing any additional available funds. The function correctly calculates a `newAmount` and updates the central lock record via `m.UpdateLock`. However, the function then assigns the old lock amount back to the in-memory `order.LockedAmount` field.

Recommendation

Update the in-memory value to reflect the latest amount after the lock update.

Alleviation

[Kaia, 10/29/2025]:
Issue acknowledged. Changes have been reflected in the commit `c3d0dc00cd6fd173fa940bcf2ed61d1ffdf9cd5c`.

ASA-76 | Improper Locking Order (Race Condition) In `Lock()`

Category	Severity	Location	Status
Logical Issue	Medium	core/orderbook/v2/balance/manager.go (go-ethereum-6101af6): 79~80, 89, 147, 283, 769~772; core/state/state_object.go (go-ethereum-6101af6): 796; core/state/statedb.go (go-ethereum-6101af6): 582~587	Resolved

Description

In `Lock()` function, the balance check occurs before acquiring `m.mu`, meaning two concurrent `Lock()` calls for the same user could both see sufficient funds and proceed to lock them simultaneously.

Recommendation

Acquire `m.mu` before calling `GetTokenBalance()` and performing the balance check to make the check-and-lock operation atomic.

Alleviation

[Kaia, 10/29/2025]:
Issue acknowledged. Changes have been reflected in the commit `7116ab82e6e97c1b6da2ea58607a18900f89e747`.

ASA-77 | Potential Transaction Bloat Attack Due To Trailing Bytes

Category	Severity	Location	Status
Denial of Service	Medium	core/types/session.go (go-ethereum-6101af6): 47; core/types/tx_input.go (go-ethereum-6101af6): 102, 191, 564, 627, 666, 977; core/types/value_transfer.go (go-ethereum-6101af6): 30	Resolved

Description

The DEX command transaction type uses `json.Unmarshal()` to decode transaction bytes. However, `json.Unmarshal()` tolerates trailing bytes, without raising an error.

Recommendation

Recommend strictly decoding the transaction bytes via rejecting any trailing bytes.

Alleviation

[Kaia, 11/22/2025]:
Issue acknowledged. Changes have been reflected in the commit `10074c0025921de77b15336574e9dea136c58b5f` .

ASA-78 Ambiguous Quantity Semantics Between Quote And Base Tokens

Category	Severity	Location	Status
Coding Style	Medium	core/orderbook/v2/matching/price_time_priority.go (go-ethereum-6101af6): 283	Resolved

Description

In the trade execution logic, the interpretation of `Quantity` is inconsistent across different parts of the matching process.

Recommendation

Consider unifying the semantics of `Quantity` across all code paths.

Alleviation

[Kaia, 11/27/2025]:

Issue acknowledged. Changes have been reflected in commit `d9134f807b3df2b58315ab8acb4393392c6c4dd6` .

ASA-79 Lot-Size/Dust Validation Bypass For SELL Market Orders In Quote Mode After Lock Limiting

Category	Severity	Location	Status
Logical Issue	Medium	core/orderbook/v2/matching/price_time_priority.go (go-ethereum-6101af6): 275	Resolved

Description

In `matchMarketQuoteMode()`, when the taker is a `SELL` order in quote mode and a base-asset lock (`order.LockedAmount`) is present, the function reduces `execQuantity` to respect the lock after it has already performed lot-size rounding and dust checks. It never re-applies lot-size rounding or dust validation to the new, smaller `execQuantity`.

Recommendation

Consider applying the second round of RoundDownToLotSize/IsQuantityDust on this new `execQuantity` before creating the trade.

Alleviation

[Kaia, 11/26/2025]:
Issue acknowledged. Changes were reflected in the commit `2bf7081e0a0bd0b28690b16ec4abbc1b2ececfe1`.

ASA-80 Incorrect Unlock Identifier In `createTPSLOrders` May Cause Stuck TPSL Locks And Balance Inconsistency

Category	Severity	Location	Status
Inconsistency, Logical Issue	Medium	core/orderbook/v2/engine/symbol_engine.go (go-ethereum -6101af6): 814, 824	Resolved

Description

The function `createTPSLOrders()` is intended to generate `TPSL` orders for a filled order that includes `TPSL`, and it establishes an early `TPSL` lock through the `createTPSLLock()` function, which assigns a lock identifier in the format `"<orderID>_TPSL"`. If `TPSL` order creation fails, the rollback logic calls `Unlock()`, using the original order ID instead of the `TPSL` order ID.

Recommendation

Update the rollback logic in `createTPSLOrders()` to call `e.balanceManager.Unlock()`.

Alleviation

[Kaia, 11/06/2025]:

Issue acknowledged. Changes have been reflected in the commit `e9b3a813ac0ca7c5f371b4e88824c968e9e5c385`.

ASA-81 Discussion On Gas-Free Dex Commands Design That Enables Multi-Layer DoS

Category	Severity	Location	Status
Design Issue, Denial of Service	Medium		Acknowledged

Description

Alpha Sec. dex treats every Dex command transation as gas-free. That design removes the economic backstop that normally throttles abusive traffic, potentially resulting the dos attack with transaction-specific path, oversized transaction and spam transactions.

Recommendation

The gas-free mechanism of the dex command transaction one of the design choice currently implemented in the kaia orderbook dex, the audit team would like to confirm with the team how such DoS attack vectors would be prevented and recommend revisiting the gas-free design.

Alleviation

[Kaia, 11/27/2025]:
The team acknowledged the issue and decided not to implement the recommended change in the current engagement.

ASA-82 Both TP Order And SL Orders Could Exist In Orderbook In Some Edge Cases

Category	Severity	Location	Status
Design Issue	Medium	core/orderbook/v2/dispatcher/dispatcher.go (go-ethereum-6101af6): 650	Resolved

Description

In the `ModifyOrder()` function, `processOrderInternal()` is used to match orders and generate trades. If a passive order is marked as tpsl, a TP order may be created and added to the triggeredQueue via `CreateTPSLForFilledOrder()`.

However, during `ModifyOrder()`, triggered TP orders may remain unprocessed in the queue when the function completes. In rare cases, if a stop-loss condition is triggered before the queued TP order is handled, the SL order may enter the orderbook while the corresponding TP order is still pending.

Recommendation

Consider aligning the post-processing of `ModifyOrder()` with `ProcessOrder()`.

Alleviation

[Kaia, 11/27/2025]:
Issue acknowledged. Changes have been reflected in the commit `912bee211bc712425e6d9730f58b758f91c5b332`.

ASA-83 | TriggeredQueue Not Restored From Engine Snapshot

Category	Severity	Location	Status
Coding Issue	Medium	core/orderbook/v2/engine/symbol_engine.go (go-ethereum-6101af6): 1216~1270	Resolved

Description

The function `RestoreFromSnapshot()` restores the `SymbolEngine` state from a snapshot. While this function restores orders, triggers, and OCO pairs, it does not restore the `triggeredQueue` in the engine.

Recommendation

Ensure that `triggeredQueue` is persisted in the snapshot and restored properly.

Alleviation

[Kaia, 11/27/2025]:
Issue acknowledged. Changes have been reflected in the commit 912bee211bc712425e6d9730f58b758f91c5b332.

ASA-84 Incorrect Value Copy During Aggregation Leads To Erroneous Market Depth

Category	Severity	Location	Status
Logical Issue	Medium	core/orderbook/v2/book/orderbook.go (go-ethereum-6101af6): 340~359	Resolved

Description

The `aggregateOrders()` function incorrectly copies `PriceLevel` structs by value into its result slice. Subsequent updates to the same price level, meant to aggregate order quantities, modify the original struct via a pointer map but fail to update the copy in the slice.

Recommendation

It's recommended to ensure the final slice is constructed from the fully aggregated data, not from intermediate copies.

Alleviation

[Kaia, 11/22/2025]:
Issue acknowledged. Changes have been reflected in the commit `da0d179ce5dc70cefd910867ee86fa9075c196ac` .

ASA-85 Non-Atomic TPSL Creation Can Lead To Orphaned Orders And Inconsistent State

Category	Severity	Location	Status
Logical Issue	Medium	core/orderbook/v2/conditional/manager.go (go-ethereum-6101af6): 69 ~148	Resolved

Description

The `CreateTPSLForFilledOrder()` function establishes a TPSL setup in a multi-step, non-atomic sequence.

Recommendation

Recommend refactoring the TPSL creation process to be atomic.

Alleviation

[Kaia, 11/25/2025]:

Issue acknowledged. Changes have been reflected in the commit `dbbfa5671b05de6d362f3e6e12e2c91dcd76b3be` .

ASA-86 Funds Unlocked Before Order Removal In `handleCancelAllRequest()`

Category	Severity	Location	Status
Volatile Code, Denial of Service	Medium	core/orderbook/v2/dispatcher/dispatcher.go (go-ethereum-6101af6): 560, 574~575	Resolved

Description

The `handleCancelAllRequest()` is intended to handle the request to cancel all the orders from the user (`L1Owner`). If an error occurs during the order cancellation step, some orders may remain in the orderbook even though their previously locked tokens have already been unlocked.

Recommendation

Recommend refactoring the logic so that orders are cancelled first, and tokens are unlocked only for those orders that have been successfully cancelled.

Alleviation

[Kaia, 11/23/2025]:
Issue acknowledged. Changes have been reflected in the commit `6faedee3c1d900871dbda8e1f1860f9e24515ccc` and `370ed4b3d95c612d87087298cb191c4ea46ff660`.

ASA-100 Potential Exploitation Of SL Market Orders Via Extreme Price Updates

Category	Severity	Location	Status
Design Issue	Minor	core/orderbook/v2/engine/symbol_engine.go (go-ethereum-6101af6): 947~959; core/orderbook/v2/matching/price_time_priority.go (go-ethereum-6101af6): 38~39	Acknowledged

Description

After an order is processed, its price may be updated. The updated price can trigger stop-loss (SL) orders, which are added to the `triggerQueue` for the next processing round. Market SL orders are executed immediately against the `OrderBook`.

Recommendation

Consider implementing protections to prevent extreme-price orders from being accepted or processed.

Alleviation

[Kaia, 11/21/2025]:
The team acknowledged the issue and decided not to implement the recommended change in the current engagement

ASA-101 Unsynchronized And Unvalidated Metadata Persistence In `Stop()` Causes WAL Inconsistency

Category	Severity	Location	Status
Logical Issue	Minor	core/orderbook/v2/persistence/wal_manager.go (go-ethereum-6101af6): 95~114	Resolved

Description

The WALManager tracks the last written WAL sequence (`w.currentSequence`) and block number (`w.currentBlock`) in memory and persists them to the database under metadata keys. However, `saveMetadata()` neither acquires `w.mu` (the primary mutex guarding state updates), nor validates that a WAL entry for `w.currentSequence` actually exists on disk.

Recommendation

Validate before persisting to ensure that the WAL entry for `w.currentSequence` exists.

Alleviation

[Kaia, 11/13/2025] :
Issue acknowledged. Changes have been reflected in the commit `5a258ef8f271f48f5d57b03f95e88f5ea3a9281f` .

ASA-102 | Potential Overflow Leads To Panic With `MustFromBig()`

Category	Severity	Location	Status
Volatile Code	Minor	precompiles/ArbTokenIssuer.go (dex-core-188b108): 65, 103, 132	Resolved

Description

The input `amount` passed into the function `Mint()` and `Burn()` misses the overflow check. The `MustFromBig()` panics if the `big.Int` input is overflowed.

Recommendation

Recommend adding the overflow check to prevent unexpected panic.

Alleviation

[Kaia, 11/12/2025]:
Issue acknowledged. Changes have been reflected in the commit `9e44da91d3035c6d19433833ae3fa72f34823eea`.

ASA-103 | Insufficient Constraint On Data Size

Category	Severity	Location	Status
Volatile Code	Minor	precompiles/ArbTokenIssuer.go (dex-core-188b108): 218, 342	Resolved

Description

The following code of data parse only checks that the length of remaining data is no less than the current position plus the data length. If the data length correctly encodes the size of the data, then the check is supposed to be equality.

Recommendation

Recommend changing the check to equality.

Alleviation

[Kaia, 11/12/2025]:

Issue acknowledged. Changes have been reflected in the commit `77c7fc563c8d4280fb0ecf4f253cbeb8f193cc6a` .

ASA-104 | Expired Session Wallet Does Not Fail

Category	Severity	Location	Status
Inconsistency	Minor	core/types/transaction.go (go-ethereum-6101af6): 459	Resolved

Description

All dex command transaction invokes the function `ValidateDexCommand()`, which validates the sender matches one of the session wallet. However, it only logs the error when the session wallet is expired, instead of returning error.

Recommendation

Recommend propagating the error so that expired session wallet cannot submit transaction on behalf of l1owner.

Alleviation

[Kaia, 11/12/2025]:
Issue acknowledged. Changes have been reflected in the commit `4bae8074e93013dff7942fe184ec527c7cb961dd`.

ASA-105 | Missing Nonzero Check Of Input `data`

Category	Severity	Location	Status
Volatile Code	Minor	precompiles/ArbTokenIssuer.go (dex-core-188b108): 341	Acknowledged

Description

The function `parseEncodedBytes()` is intended to parse the `name` and `symbol` of the ERC20 metadata, while it does not check the string length of such bytes are nonzero.

Recommendation

Recommend adding the nonzero check to prevent unexpected results.

Alleviation

[Kaia, 11/12/2025]:
The team acknowledged the issue and decided not to implement the recommended change in the current engagement.

ASA-106 Dispatcher Panics On Shutdown If New Requests Arrive After `Stop()`

Category	Severity	Location	Status
Denial of Service, Volatile Code	Minor	core/orderbook/v2/dispatcher/dispatcher.go (go-ethereum-6101af6): 111, 132	Resolved

Description

The `Dispatcher.Stop()` cancels the dispatcher context and closes `requestChan`. Any concurrent call to `Dispatcher.DispatchReq()` that races after the close still executes the select; the send case `d.requestChan <- req` is chosen immediately, but sending on a closed channel panics.

Recommendation

Ensure `DispatchReq()` refuses new requests once `Stop()` begins instead of sending to a closed channel.

Alleviation

[Kaia, 11/23/2025]:
Issue acknowledged. Changes have been reflected in the commit `c4abd026c522d93652744806490fcc2af25b18c9`.

ASA-107 Missing `LockedAmount` In Deep Copy Method `Copy()` Of `Order`

Category	Severity	Location	Status
Volatile Code, Inconsistency	Minor	core/orderbook/v2/types/order.go (go-ethereum-6101af6): 233	Resolved

Description

The `Copy()` method of `Order` is intended to perform a deep copy, but it misses the field `LockedAmount`.

Recommendation

Recommend deep copying `LockedAmount` in the `Copy()` method of `Order`.

Alleviation

[Kaia, 11/23/2025]:

Issue acknowledged. Changes have been reflected in the commit `80116d4dab9868f25de9c22b17ce16b34796220b`.

ASA-108 | `validate()` Misses Validation Of `OrderMode`

Category	Severity	Location	Status
Volatile Code, Inconsistency	Minor	core/types/tx_input.go (go-ethereum-6101af6): 192, 667, 978	Resolved

Description

The `OrderMode` in `OrderContext`, `ModifyContext` and `StopOrderContext` is supposed to be either 0 (base mode (default)) or 1 (quote mode), but there is no validation to ensure that.

Recommendation

Recommend adding the validation of `OrderMode` to ensure a malformed order will be rejected.

Alleviation

[Kaia, 11/25/2025]:
Issue acknowledged. Changes have been reflected in the commit `9bb0aff6f200422ec6deca74f69fb48d9af16020`.

ASA-109 Missing Validation Of Existing Order In `validate()` Of ModifyContext

Category	Severity	Location	Status
Volatile Code, Inconsistency	Minor	core/types/tx_input.go (go-ethereum-6101af6): 667	Resolved

Description

According to the logic in `Dispatcher.handleModifyRequest()`, the modified order should not contain `TPSL` and must be a `LIMIT` order. However, there is no validation on the existing order.

Recommendation

Recommend adding these validation to reject the malformed transaction.

Alleviation

[Kaia, 11/23/2025]:
Issue acknowledged. Changes have been reflected in the commit `ea995eab197e39cd438a25c8dcedf05ca1f5f7c4`.

ASA-110 Incorrect Order Of Return Values In `GetOrderbookSnapshot()`

Category	Severity	Location	Status
Logical Issue	Minor	core/orderbook/v2/engine/symbol_engine.go (go-ethereum-6101af6): 1081	Resolved

Description

In the function `GetOrderbookSnapshot()`, `orderbook.GetDepth()` returns (bids, asks), but the engine assigns it as (asks, bids).

Recommendation

Recommend correcting the variable assignments in the `GetOrderbookSnapshot()` function to ensure `bids` and `asks` are assigned in the proper order.

Alleviation

[Kaia, 11/21/2025]:
Issue acknowledged. Changes have been reflected in the commit `9897543b4bcec2bfdff064941f0e5b3588076cc0`.

ASA-111 | Non-Determinism Due To Map Iteration

Category	Severity	Location	Status
Volatile Code, Inconsistency	Minor	core/orderbook/v2/balance/manager.go (go-ethereum-6101af6): 280, 808; core/orderbook/v2/book/orderbook.go (go-ethereum-6101af6): 562; core/orderbook/v2/tpsl/oco_controller.go (go-ethereum-6101af6): 280; core/orderbook/v2/tpsl/trigger_manager.go (go-ethereum-6101af6): 154, 221, 262	Acknowledged

Description

Go randomises map iteration order, so every call can return the slice in a different order. In the linked places, the map iteration without sorting the keys.

Recommendation

Recommend performing an extra sorting to ensure deterministic results.

Alleviation

[Kaia, 11/23/2025]:
Issue acknowledged. I will fix the issue in the future, which will not be included in this audit engagement.

ASA-112 Time-Nonce Validation Could Possibly Be Bypassed In `timeNonceDriftAcceptable()`

Category	Severity	Location	Status
Inconsistency, Volatile Code	Minor	execution/gethexec/time_nonce.go (dex-core-188b108): 26	Resolved

Description

The `timeNonceDriftAcceptable()` function is intended to validate the time nonce is within drift window. The input `txNonce` multiplies the millisecond nonce in signed int64 before feeding it to `time.Unix`. Any nonce that overflows after multiplication, yet lands on exactly the same `txTime` as the honest timestamp.

Recommendation

Reject out-of-range inputs before doing the multiplication.

Alleviation

[Kaia, 11/24/2025]:
Issue acknowledged. Changes have been reflected in the commit `b533a8060ba3b0f449eea2caa8f0cdcb70445094`.

ASA-113 Missing Validation Of Existing Order In `validate()` Of `CancelAllContext`

Category	Severity	Location	Status
Volatile Code	Minor	core/types/tx_input.go (go-ethereum-6101af6): 635	Resolved

Description

Unlike the `validate()` of `CancelContext`, the `validate()` of `CancelAllContext` does not validate whether the `L1Owner` has any existing order.

Recommendation

Recommend adding the validation to ensure the `L1Owner` has existing order to prevent the execution of these spam transactions.

Alleviation

[Kaia, 11/27/2025]:
Issue acknowledged. Changes have been reflected in the commit `dac83203ac2987a1cf3b694f36888d54acfe7bf2`.

ASA-114 Missing Deep Copy Of Order Information In `CreateModifiedOrder()`

Category	Severity	Location	Status
Volatile Code	Minor	core/orderbook/v2/engine/symbol_engine.go (go-ethereum-54ccff3): 654 ~658, 664, 669	Resolved

Description

`CreateModifiedOrder()` builds the replacement order before the old order is cancelled, but it doesn't clone any of the numeric fields it copies from `existingOrder`.

Recommendation

Recommend performing deep copy of the values to the new order in `CreateModifiedOrder()`.

Alleviation

[Kaia, 11/30/2025]:
Issue acknowledged. Changes have been reflected in the commit `534acb0c7686016b9fad193e07a7ad0998934810`.

ASA-115 | Async Delta Loss Due To Premature Dirty-Flag Reset

Category	Severity	Location	Status
Volatile Code, Inconsistency	Minor	core/orderbook/v2/persistence/delta_writer.go (go-ethereum-54ccff3): 110; core/orderbook/v2/persistence/manager.go (go-ethereum-54ccff3): 225~232	Acknowledged

Description

`PersistenceManager.OnBlockEnd()` enqueues each block's `DispatcherDelta` and immediately calls `dispatcher.ResetAllDirtyTracking()`. Any crash between enqueueing the delta and the writer's disk flush permanently drops that block's mutations from persistence.

Recommendation

Defer `dispatcher.ResetAllDirtyTracking()` until the delta write completes.

Alleviation

[Kaia, 11/30/2025]:
The team acknowledged the issue and decided not to implement the recommended change in the current engagement.

ASA-126 Non-Atomic Lock Consumption Can Leave Balances Partially Consumed On Failure

Category	Severity	Location	Status
Volatile Code	Minor	core/orderbook/v2/balance/settlement.go (go-ethereum-6101af6): 139	Resolved

Description

`verifyAndConsumeLocks()` first checks both sides have sufficient locked balances, then calls ConsumeLock for the buyer, and only afterwards calls ConsumeLock for the seller. If the seller consumption fails (e.g., due to concurrent updates, alias resolution changes, or StateDB inconsistency), the function returns an error after the buyer lock has already been consumed. The caller does not roll back the first consumption, leaving the system in an inconsistent state where part of the trade was charged but assets were not delivered. This can result in user funds being consumed without settlement.

Recommendation

Make the consumption of both locks atomic with rollback.

Alleviation

[Kaia, 12/18/2025]:
Issue acknowledged. Changes have been reflected in the commit `95869593ed4b6f7691d92aea33fda066f7ec8460` and `67ecc7efaa3100b4d2a64ecd3b3c815169acda28`.

ASA-127 Discussion On Logged Settlement And OCO Failures Without Proper Handling

Category	Severity	Location	Status
Inconsistency	Minor	core/orderbook/v2/conditional/manager.go (go-ethereum-9586959): 219 ~221; core/orderbook/v2/dispatcher/dispatcher.go (go-ethereum-9586959): 286~289	Resolved

Description

- In `processTradesAndCleanup()` a failed `settleTrade()` is logged, but we still `CompleteOrder()/untrackOrder()` filled legs; `processSingleOrderResult()` will also finalize the taker on terminal status. So if settleTrade fails (nil FeeRetriever, bad token IDs, StateDB error, etc.), orders are removed/unlocked and OCO/conditional flows continue even though no asset transfers occurred, and verifyAndConsumeLocks may already have consumed locked balances with no rollback.
- Similarly, when an SL trigger fires, `conditional.Manager.CheckTriggers()` removes the OCO pair via `ExecuteOCO()` and then calls the `SymbolEngine.cancelOrderDirect()` for each paired order. If the canceller returns an error (e.g., TP not found), `CheckTriggers()` just logs it; the pair is already removed, so the TP stays live with no OCO mapping and both legs can execute.

Recommendation

The audit team would like to know if the current behavior is by design. If not, recommend handling the error properly in these corner cases.

Alleviation

[Kaia, 12/18/2025]:
Issue acknowledged. Changes have been reflected in the commit `67ecc7efaa3100b4d2a64ecd3b3c815169acda28` and `6684a4643f55a916d84e48ddc633b2ede6df06e6`.

ASA-128 | Missing Comparison Between `SLLimit` And `SLTrigger`

Category	Severity	Location	Status
Logical Issue	Minor	core/types/tx_input.go (go-ethereum-9586959): 317~318	Acknowledged

Description

There is no side-aware check in `validate()` of `OrderContext` that `SLLimit` (if not nil) is on the correct side of `SLTrigger`. As a result, a BUY can submit `SLLimit` above the trigger (or `SELL` below) and pass tx validation, leading to an SL limit that's unlikely to execute when triggered.

Recommendation

For sensible execution, `SLLimit` should be on the "exit" side of the trigger (BUY: ≤ trigger; SELL: ≥ trigger).

Alleviation

[Kaia, 12/17/2025]:

Issue acknowledged. I won't make any changes for the current version.

ASA-129 | Stale Depth From In-Place Order Mutation In `UpdateOrder()`

Category	Severity	Location	Status
Volatile Code	Minor	core/orderbook/v2/book/orderbook.go (go-ethereum-9586959): 149	Resolved

Description

`UpdateOrder()` assumes it gets an untouched “old” order, but the matcher mutates the passive order in place before calling it. Inside `UpdateOrder()`, oldOrder and order are the same pointer, so the code removes and re-adds the price level using already-reduced quantities. The net effect on aggregated depth is zero: Level2 totals stay at the pre-fill amount even though the queue entry shrank, producing stale depth data.

Recommendation

Fetch and store a copy of the order’s previous state within UpdateOrder before applying any updates, ensuring aggregated price levels are correctly updated and remain consistent with the underlying order queue.

Alleviation

[Kaia, 12/18/2025]:
Issue acknowledged. Changes have been reflected in the commit `1bc200ae59862e658805ceb9e06281ae3e5f8304`.

ASA-130 TPSL Creation Failure Leaves Stale Pre-Registered TP/SL Routes

Category	Severity	Location	Status
Inconsistency	Minor	core/orderbook/v2/dispatcher/dispatcher.go (go-ethereum-9586959): 633	Resolved

Description

When `CreateTPSLAfterSettlement()` fails inside `processConditionalPostSettlement()`, the code only logs the error and appends FailedOrders but does not clean up the TP/SL order IDs that were pre-registered earlier via `preRegisterTPSL()`.

Recommendation

Recommend explicitly untracking the TP/SL IDs in the error branch of `processConditionalPostSettlement()`.

Alleviation

[Kaia, 12/17/2025]:

Issue acknowledged. Changes have been reflected in the commit `31b34416d87ff07a5cabc218c7e56d751c0da090`.

ASA-131 | Market Orders Accept Negative Prices

Category	Severity	Location	Status
Volatile Code	Minor	core/types/tx_input.go (go-ethereum-9586959): 400	Resolved

Description

Negative market prices are not rejected during order validation and are instead converted to their two's complement representation via `MustFromBig()`.

Recommendation

Recommend adding a check to reject the negative price.

Alleviation

[Kaia, 12/16/2025]:
Issue acknowledged. Changes have been reflected in the commit `51ed57f9bb950ffb703951f094bc4fe2e73cee18`.

ASA-87 Order Lock Can Be Removed When `oldOrderID` Equals `NewOrderID`

Category	Severity	Location	Status
Volatile Code, Logical Issue	Minor	core/orderbook/v2/balance/manager.go (go-ethereum-6101af6): 716 ~717; core/orderbook/v2/dispatcher/dispatcher.go (go-ethereum-6101af6): 650	Resolved

Description

In the `ModifyOrderLock()` function, the lock record is updated and then the old key is deleted. If `oldOrderID` is the same as `newOrder.OrderID`, the `delete()` call removes the newly updated lock entry from `m.locks`.

Recommendation

Consider updating `ModifyOrderLock` to handle the same-ID scenario safely without deleting the lock.

Alleviation

[Kaia, 11/04/2025]:

Issue acknowledged. Changes have been reflected in the commit `e52e326a2b8070c658321d64e61787b56ceb2154`.

ASA-88 Missing Nil Pointer Check In `Copy()` Of `ValueTransferContext`

Category	Severity	Location	Status
Volatile Code, Coding Issue	Minor	core/types/value_transfer.go (go-ethereum-6101af6): 56	Resolved

Description

The `Copy()` function of `ValueTransferContext` misses the nil pointer check of its field, `Value`, which could possibly lead to dereference panic when invoking `s.Value.Bytes()`.

Recommendation

Recommend adding a nil pointer check of `s.Value` before calling its method `Bytes()`.

Alleviation

[Kaia, 11/04/2025]:

Issue acknowledged. Changes have been reflected in the commit `a208e3da498cfecb1e60932a0fd45a424d4d51c8`.

ASA-89 | Unsafe Internal Pointer Exposure Via `GetBuyOrders()`

Category	Severity	Location	Status
Logical Issue	Minor	core/orderbook/v2/book/orderbook.go (go-ethereum-6101af6): 204, 212, 220, 228; core/orderbook/v2/queue/buy_queue.go (go-ethereum-6101af6): 44, 82~87, 90~108; core/orderbook/v2/queue/sell_queue.go (go-ethereum-6101af6): 44, 81~87, 90~108	Acknowledged

Description

The functions `orderBook.GetBuyOrders()` and `buy_queue.GetOrdersSorted()` expose internal order pointers `*types.Order` directly to external callers. Although both functions attempt to return a “copy” of the internal slice, the copy operation only performs a shallow copy.

Recommendation

Return deep copies or detached value copies of internal order data rather than raw pointers.

Alleviation

[Kaia, 11/23/2025]:
Issue acknowledged. Changes have been reflected in the commit `f7851a97ebb5b22debbfdad16a4866eebea0165d` .

ASA-90 Order Quantity And Price Validation Uses `IsZero()` Instead Of `Sign()` To Ensure Strict Positivity

Category	Severity	Location	Status
Volatile Code	Minor	core/orderbook/v2/engine/symbol_engine.go (go-ethereum-6101af6): 984~986, 989	Resolved

Description

Within the `validateOrder()` function, `order.Quantity` and `order.price` are of type `*uint256.Int`. Using `IsZero()` only detects if the value is exactly zero, and does not correctly validate that the quantity is strictly positive.

Recommendation

Replace the quantity validation check to enforce strict positivity.

Alleviation

[Kaia, 11/11/2025]:
Issue acknowledged. Changes have been reflected in the commit `b14398f0e3c66668fa1319b0a437c83a1126e530`.

ASA-91 | Reversed Conditional In `TokenTransferContext.copy()`

Category	Severity	Location	Status
Logical Issue	Minor	core/types/tx_input.go (go-ethereum-6101af6): 96~98	Resolved

Description

The `TokenTransferContext.copy()` function contains a reversed conditional check.

Recommendation

Recommend correcting the conditional logic in the `copy()` function.

Alleviation

[Kaia, 11/04/2025]:

Issue acknowledged. Changes have been reflected in the commit `a208e3da498cfecb1e60932a0fd45a424d4d51c8` .

ASA-92 | Missing Copy Of `LockedBalance` In `Copy()` Of `StateAccount`

Category	Severity	Location	Status
Inconsistency	Minor	core/types/state_account.go (go-ethereum-6101af6): 54	Resolved

Description

The `Copy()` function of `StateAccount` misses the copy of its field `LockedBalance`.

Recommendation

Recommend adding the copy of `LockedBalance`.

Alleviation

[Kaia, 11/11/2025]:

Issue acknowledged. Changes have been reflected in the commit `28837125a4cafe35af335b216580a70e1b5eeb1a`.

ASA-93 | Missing `LockedBalance` In `Account`

Category	Severity	Location	Status
Inconsistency	Minor	core/types/account.go (go-ethereum-6101af6): 36	Resolved

Description

The following `Account` struct misses the field `LockedBalance` that was declared in the `StateAccount` to represent the locked balance of Kaia token.

Recommendation

Recommend adding the `LockedBalance` field to `Account` struct.

Alleviation

[Kaia, 11/11/2025]:
Issue acknowledged. Changes have been reflected in the commit `28837125a4cafe35af335b216580a70e1b5eeb1a` .

ASA-94 | Non-Deterministic `MarshalJSON()` Of `Balances`

Category	Severity	Location	Status
Inconsistency	Minor	core/types/token_balance.go (go-ethereum-6101af6): 169	Resolved

Description

The `MarshalJSON()` of `Balances` utilizes the map iteration to append the elements in a slice, which could be non-deterministic in Go.

Recommendation

Recommend sorting the keys in the map to ensure deterministic marshal.

Alleviation

[Kaia, 10/30/2025]:
Issue acknowledged. Changes have been reflected in the commit `4faba41b9dde05fcac01da87bb59b43459d75f01` .

ASA-95 Mutable Aliasing In `NewOrder()` Allows Caller Modify `price/quantity/TPSL` After Order Creation

Category	Severity	Location	Status
Logical Issue	Minor	core/orderbook/v2/types/order.go (go-ethereum-6101af6): 203~222; core/orderbook/v2/types/trade.go (go-ethereum-6101af6): 63~64	Resolved

Description

`NewOrder()` stores caller-provided pointers (`price` , `quantity` , and `TPSLContext`) directly into the returned `Order` without cloning.

Recommendation

Perform deep cloning of all mutable inputs inside `NewOrder()` .

Alleviation

[Kaia, 11/05/2025]:
Issue acknowledged. Changes have been reflected in commit `849a542130da9b3fe2dfced236e075823a9bbee2` .

ASA-96 | FILLED Orders Can Be Reactivated

Category	Severity	Location	Status
Logical Issue	Minor	core/orderbook/v2/types/order.go (go-ethereum-6101af6): 291~302	Resolved

Description

`UpdateStatus()` treats only `REJECTED`, `CANCELLED`, and `EXPIRED` as terminal and proceeds for all other statuses. `FILLED` is a terminal status but is not included in the early-return guard of `UpdateStatus()`.

Recommendation

Update the `UpdateStatus()` method to treat all terminal states, including `FILLED`.

Alleviation

[Kaia, 11/05/2025]:

Issue acknowledged. Changes have been reflected in commit `5d32e82f63b18c2397c4f267c780fee9b823fef3`.

ASA-97 `AllOrNone` OCO Strategy Incorrectly Implemented — Behaves Same As `OneCancelsOther`

Category	Severity	Location	Status
Inconsistency	Minor	core/orderbook/v2/tpsl/interfaces.go (go-ethereum-6101af6): 110; core/orderbook/v2/tpsl/oco_controller.go (go-ethereum-6101af6): 81, 91	Resolved

Description

In the `CancelOCO()` function, the implementation for the `AllOrNone` strategy is identical to that of `OneCancelsOther`. Both strategies currently skip cancelling the triggering order (if id != orderID), meaning that when one order in an `AllOrNone` pair is manually cancelled, only the other orders are cancelled.

Recommendation

Update the `AllOrNone` case in `CancelOCO()` to cancel all orders, including the triggering one, by removing the condition if id != orderID.

Alleviation

[Kaia, 11/04/2025]:
Issue acknowledged. Changes have been reflected in the commit `c8e5a9f81e37efed827aa7bff06e6e8be95983f4`.

ASA-98 | Invalid State Transition In `TPSLOrder.Cancel()`

Category	Severity	Location	Status
Logical Issue	Minor	core/orderbook/v2/types/conditional.go (go-ethereum-6101af6): 283~291	Acknowledged

Description

`TPSLOrder.Cancel()` unconditionally sets the `TPSL` status to `CANCELLED` and the SL child order's status to `CANCELLED` without validating the current state.

Recommendation

Reject invalid state transition from `TRIGGERED` back to `CANCELLED`.

Alleviation

[Kaia, 11/25/2025]: The team acknowledged the issue and decided not to implement the recommended change in the current engagement.

ASA-99 | Missing Check In `MakeTimeNonceError()` Function

Category	Severity	Location	Status
Logical Issue, Inconsistency	Minor	execution/gethexec/time_nonce.go (dex-core-188b108): 35	Resolved

Description

The `MakeTimeNonceError()` function misses the check that Timestamp Nonce must always be greater than the current state nonce, which is inconsistent with the design documentation.

Recommendation

Recommend explicitly adding the check that Timestamp Nonce is greater than the state nonce.

Alleviation

[Kaia, 11/25/2025]:
Issue acknowledged. Changes have been reflected in the commit `0b956f6f8db99cfcfe7595bad340c917bd58b962` .

ASA-116 | Incorrect `fromAmount` Logging In `TransformLock()` Function

Category	Severity	Location	Status
Logical Issue	Informational	core/orderbook/v2/balance/manager.go (go-ethereum-6101af6): 775~784	Resolved

Description

In the `TransformLock()` function, the system logs the transformation details of a token lock (used in `TPSL` scenarios). However, the function updates `lock.Amount` before writing the log entry.

Recommendation

Preserve the original amount before overwriting it and use the preserved value in logs.

Alleviation

[Kaia, 10/30/2025]:
Issue acknowledged. Changes have been reflected in commit `f9ecc1fbd4e6366686b73b2c785bd145cc903ec4` .

ASA-117 | Incorrect Error Messages In `validate()`

Category	Severity	Location	Status
Inconsistency	Informational	core/types/tx_input.go (go-ethereum-6101af6): 120~125; core/types/value_transfer.go (go-ethereum-6101af6): 38~43	Resolved

Description

The following error messages are not accurately describing the previous condition:

- "amount must be positive" should be "amount must be non-negative"
- "price exceeds uint256 max value" should be "amount exceeds uint256 max value"

Recommendation

Recommend correcting the error messages.

Alleviation

[Kaia, 10/30/2025]:
Issue acknowledged. Changes have been reflected in the commit `ea11f988e6c0b721a62be5648096f97b59452447` .

ASA-118 | Discussion On Missing `Metadata` In Signing Message

Category	Severity	Location	Status
Inconsistency	Informational	core/types/session.go (go-ethereum-6101af6): 124, 154~155	Resolved

Description

The `Session` struct contains the `Metadata` field, while it's been ignored during the signing process when converting the `Session` into signing message via `ToTypedData()`.

Recommendation

The audit team would like to understand the design intention of the `Metadata`.

Alleviation

[Kaia, 10/29/2025]:

`Metadata` field is not used for validation actually. Changes have been reflected in the commit `feca04c6d0945fce0d24e7e8c317a65f323f1375`.

ASA-119 | Discussion On Non-Functional WAL Manager Initialization

Category	Severity	Location	Status
Logical Issue	Informational	core/orderbook/v2/persistence/manager.go (go-ethereum-6101af6): 82~83; core/orderbook/v2/persistence/wal_manager.go (go-ethereum-6101af6): 67~69	Resolved

Description

The persistence subsystem claims to provide Write-Ahead Logging (WAL) durability, but the WAL manager is never actually started. In `manager.Start()`, the call to `p.walManager.Start()` is commented out, meaning the WAL subsystem is never initialized.

Recommendation

The audit team would like to confirm with the team if this is an incomplete implementation.

Alleviation

[Kaia, 11/12/2025]:
The relevant logics were removed from the codebase in the commit `5a258ef8f271f48f5d57b03f95e88f5ea3a9281f`.

ASA-120 | Discussion On Logging Errors Without Return

Category	Severity	Location	Status
Design Issue, Coding Issue	Informational	precompiles/ArbTokenIssuer.go (dex-core-188b108): 70~72, 92~94, 108~110, 142~145	Acknowledged

Description

In the `Mint()` and `Burn()` function, there is no `return` after the errors occur. The contract logs failures from the `TokenTransfer` / `TokenRegistered` emitters but then continues execution.

Recommendation

The audit team would like to confirm with the team if this is an intended design.

Alleviation

[Kaia, 11/10/2025]:
The team acknowledged the issue and decided not to implement the recommended change in the current engagement.

ASA-121 | Discussion On Order Cleanup After Trade Settlement Failure

Category	Severity	Location	Status
Coding Issue	Informational	core/orderbook/v2/dispatcher/dispatcher.go (go-ethereum-6101af6): 263~269	Acknowledged

Description

When `d.settleTrade(trade)` fails, the `processTradesAndCleanup()` function proceeds to clean up orders associated with that trade. This can result in passive orders being removed from cache or marked complete despite the trade not being settled successfully. In some cases, these orders would otherwise have remained valid for future matches, leading to missing trades and inconsistent orderbook states.

Recommendation

According to the comment `// Continue processing other trades even if one fails`, this seems to be an intended design. The audit team would like to confirm with the team has considered the above scenario.

Alleviation

[Kaia, 11/24/2025]:
The team acknowledged the issue and decided not to implement the recommended change in the current engagement.

ASA-122 Duplicate `OrderType` Check In `validate()` Of `OrderContext` And `StopOrderContext`

Category	Severity	Location	Status
Code Optimization	Informational	core/types/tx_input.go (go-ethereum-6101af6): 220, 235	Resolved

Description

The `validate()` function validates the `OrderContext` and `StopOrderContext`, which performs the duplication check of `OrderType`.

Recommendation

Recommend removing the second check for code readability and optimization.

Alleviation

[Kaia, 11/22/2025]:

Issue acknowledged. Changes have been reflected in the commit `60f60d5de678745a6434e4bdc7f1c056fab03cd7`.

ASA-123 Discussion On Latest Traded Price Updated As Orderbook's Price

Category	Severity	Location	Status
Design Issue	Informational	core/orderbook/v2/engine/symbol_engine.go (go-ethereum-6101af6): 388	Acknowledged

Description

During the order-matching process, the orderbook's current price is updated based on the most recent executed trade. Because this price may come from either a buy or a sell order, it can fluctuate significantly, especially when the orderbook has low liquidity.

Recommendation

The audit team would like to understand if this is an intended design or the average of sell and buy price and a TWAP price should be utilized.

Alleviation

[Kaia, 11/27/2025]:
The team acknowledged the issue and decided not to implement the recommended change in the current engagement.

ASA-124 | Discussion On Incomplete Stage Logic

Category	Severity	Location	Status
Coding Issue	Informational	core/orderbook/v2/pipeline/locking_stage.go (go-ethereum-6101af6): 13~34; core/orderbook/v2/pipeline/matching_stage.go (go-ethereum-6101af6): 11~38; core/orderbook/v2/pipeline/queue_update_stage.go (go-ethereum-6101af6): 11~51; core/orderbook/v2/pipeline/settlement_stage.go (go-ethereum-6101af6): 11~39	Acknowledged

Description

The current implementation of the MatchingStage/LockingStage/QueueUpdateStage/SettlementStage are not production-ready and introduces several architectural and correctness concerns. While the stage structure is defined, the critical logic for state processing is commented as TODO.

Recommendation

The audit team kindly requests further context to better understand the current implementation.

Alleviation

[Kaia, 11/21/2025]:
The team acknowledged the issue and decided not to implement the recommended change in the current engagement.

ASA-132 | Missing Checks In `Copy()` Of `StopOrder` And `TPSLOrder`

Category	Severity	Location	Status
Volatile Code	Informational	core/orderbook/v2/types/conditional.go (go-ethereum-9586959): 140, 294	Resolved

Description

- The `StopOrder.Copy()` unconditionally calls `s.StopPrice.Clone()` without nil check.
- The `TPSLOrder.Copy()` blindly calls `t.SLOrder.Copy()` and `t.SLTriggerPrice.Clone()` without nil checks, and it drops `UserID`.

Recommendation

Recommend adding nil checks and `UserID` to `TPSLOrder`.

Alleviation

[Kaia, 12/19/2025]:
Issue acknowledged. Changes have been reflected in the `81e1a4296eb1a13ee2e8f695b0d62ff55d6a8a4f`.

ASA-133 | Missing Nil Check Of Trade In `processTradesAndCleanup()`

Category	Severity	Location	Status
Volatile Code	Informational	core/orderbook/v2/dispatcher/dispatcher.go (go-ethereum-9586959): 285	Resolved

Description

Nil trade would lead to nil dereference via `trade.IsBuyerMaker` though it should not occur during normal operation.

Recommendation

Recommend adding the nil pointer check.

Alleviation

[Kaia, 12/19/2025]:

Issue acknowledged. Changes have been reflected in the `81e1a4296eb1a13ee2e8f695b0d62ff55d6a8a4f` .

ASA-67 | Discussion On Any Token That Is Pre-Registered

Category	Severity	Location	Status
Logical Issue, Inconsistency	Informational	precompiles/ArbTokenIssuer.go (dex-core-188b108): 96~101	Acknowledged

Description

`ArbTokenIssuer.Mint()` function logs “L2Contract mapping added to existing token” but never verifies that a mapping exists; it immediately adds balances for that `tokenId`.

Recommendation

The audit team would like to confirm with the team whether this is the intended design.

Alleviation

[Kaia, 12/02/2025]:
Issue acknowledged. I will fix the issue in the future, which will not be included in this audit engagement.

APPENDIX | ALPHA SEC. - AUDIT

Audit Scope

kaiachain/go-ethereum
core/orderbook/v2/balance/manager.go
core/orderbook/v2/book/orderbook.go
core/orderbook/v2/dispatcher/dispatcher.go
core/orderbook/v2/engine/symbol_engine.go
core/orderbook/v2/matching/price_time_priority.go
core/orderbook/v2/pipeline/locking_stage.go
core/orderbook/v2/pipeline/matching_stage.go
core/orderbook/v2/pipeline/queue_update_stage.go
core/orderbook/v2/pipeline/settlement_stage.go
core/orderbook/v2/queue/buy_queue.go
core/orderbook/v2/queue/sell_queue.go
core/orderbook/v2/tpsl/oco_controller.go
core/orderbook/v2/tpsl/trigger_manager.go
core/orderbook/v2/types/conditional.go
core/orderbook/v2/persistence/delta_writer.go
core/orderbook/v2/persistence/manager.go
core/types/tx_input.go
core/orderbook/v2/balance/settlement.go
core/orderbook/v2/conditional/manager.go

kaiachain/go-ethereum
core/orderbook/v2/persistence/manager.go
core/orderbook/v2/persistence/wal_manager.go
core/orderbook/v2/tpsl/interfaces.go
core/orderbook/v2/types/order.go
core/orderbook/v2/types/trade.go
core/types/account.go
core/types/session.go
core/types/token_balance.go
core/types/tx_input.go
core/orderbook/v2/engine/symbol_engine.go
core/orderbook/v2/book/orderbook.go
core/orderbook/v2/conditional/manager.go
core/orderbook/v2/dispatcher/dispatcher.go
core/orderbook/v2/types/conditional.go
core/orderbook/v2/interfaces/conditional.go
core/orderbook/v2/interfaces/core.go
core/orderbook/v2/interfaces/dispatcher.go
core/orderbook/v2/interfaces/market.go
core/orderbook/v2/interfaces/request.go
core/orderbook/v2/interfaces/response.go
core/orderbook/v2/metrics/metrics.go
core/orderbook/v2/system/system.go

kaiachain/go-ethereum
core/orderbook/v2/balance/scaled_math.go
core/orderbook/v2/persistence/recovery.go
core/orderbook/v2/persistence/serialization.go
core/orderbook/v2/persistence/snapshot_manager.go
core/orderbook/v2/pipeline/builder.go
core/orderbook/v2/pipeline/conditional_stage.go
core/orderbook/v2/pipeline/context.go
core/orderbook/v2/pipeline/event_generation_stage.go
core/orderbook/v2/pipeline/integration_example.go
core/orderbook/v2/pipeline/management_pipeline.go
core/orderbook/v2/pipeline/pipeline.go
core/orderbook/v2/pipeline/trading_pipeline.go
core/orderbook/v2/pipeline/validation_stage.go
core/orderbook/v2/tpsl/activation_rule.go
core/orderbook/v2/tpsl/triggers.go
core/orderbook/v2/types/balance.go
core/orderbook/v2/types/common.go
core/orderbook/v2/types/config.go
core/orderbook/v2/types/depth.go
core/orderbook/v2/types/errors.go
core/orderbook/v2/types/fee_retriever.go
core/orderbook/v2/types/market_rules.go

kaiachain/go-ethereum
core/orderbook/v2/types/price_helpers.go
core/orderbook/v2/types/request.go
core/orderbook/v2/types/snapshot.go
core/orderbook/v2/types/statedb.go
core/orderbook/v2/types/symbol.go
core/orderbook/v2/interfaces/conditional.go
core/orderbook/v2/interfaces/core.go
core/orderbook/v2/interfaces/dispatcher.go
core/orderbook/v2/interfaces/market.go
core/orderbook/v2/interfaces/request.go
core/orderbook/v2/interfaces/response.go
core/orderbook/v2/metrics/metrics.go
core/orderbook/v2/system/system.go
core/orderbook/v2/balance/manager.go
core/orderbook/v2/balance/scaled_math.go
core/orderbook/v2/balance/settlement.go
core/orderbook/v2/book/orderbook.go
core/orderbook/v2/conditional/manager.go
core/orderbook/v2/dispatcher/dispatcher.go
core/orderbook/v2/matching/price_time_priority.go
core/orderbook/v2/persistence/recovery.go
core/orderbook/v2/persistence/snapshot_manager.go

kaiachain/go-ethereum
core/orderbook/v2/pipeline/builder.go
core/orderbook/v2/pipeline/conditional_stage.go
core/orderbook/v2/pipeline/context.go
core/orderbook/v2/pipeline/event_generation_stage.go
core/orderbook/v2/pipeline/integration_example.go
core/orderbook/v2/pipeline/locking_stage.go
core/orderbook/v2/pipeline/management_pipeline.go
core/orderbook/v2/pipeline/matching_stage.go
core/orderbook/v2/pipeline/pipeline.go
core/orderbook/v2/pipeline/queue_update_stage.go
core/orderbook/v2/pipeline/settlement_stage.go
core/orderbook/v2/pipeline/trading_pipeline.go
core/orderbook/v2/pipeline/validation_stage.go
core/orderbook/v2/queue/buy_queue.go
core/orderbook/v2/queue/sell_queue.go
core/orderbook/v2/tpsl/activation_rule.go
core/orderbook/v2/tpsl/interfaces.go
core/orderbook/v2/tpsl/oco_controller.go
core/orderbook/v2/tpsl/trigger_manager.go
core/orderbook/v2/tpsl/triggers.go
core/orderbook/v2/types/balance.go
core/orderbook/v2/types/common.go

kaiachain/go-ethereum
core/orderbook/v2/types/conditional.go
core/orderbook/v2/types/config.go
core/orderbook/v2/types/depth.go
core/orderbook/v2/types/errors.go
core/orderbook/v2/types/fee_retriever.go
core/orderbook/v2/types/market_rules.go
core/orderbook/v2/types/order.go
core/orderbook/v2/types/price_helpers.go
core/orderbook/v2/types/request.go
core/orderbook/v2/types/snapshot.go
core/orderbook/v2/types/statedb.go
core/orderbook/v2/types/symbol.go
core/orderbook/v2/types/trade.go
core/types/account.go
core/types/token_balance.go
core/types/session.go
core/types/tx_input.go
core/orderbook/v2/interfaces/conditional.go
core/orderbook/v2/interfaces/core.go
core/orderbook/v2/interfaces/dispatcher.go
core/orderbook/v2/interfaces/market.go
core/orderbook/v2/interfaces/request.go

kaiachain/go-ethereum
core/orderbook/v2/interfaces/response.go
core/orderbook/v2/metrics/metrics.go
core/orderbook/v2/system/system.go
core/orderbook/v2/balance/manager.go
core/orderbook/v2/balance/scaled_math.go
core/orderbook/v2/balance/settlement.go
core/orderbook/v2/book/orderbook.go
core/orderbook/v2/conditional/manager.go
core/orderbook/v2/dispatcher/dispatcher.go
core/orderbook/v2/engine/symbol_engine.go
core/orderbook/v2/matching/price_time_priority.go
core/orderbook/v2/persistence/delta_writer.go
core/orderbook/v2/persistence/manager.go
core/orderbook/v2/persistence/recovery.go
core/orderbook/v2/persistence/snapshot_manager.go
core/orderbook/v2/pipeline/builder.go
core/orderbook/v2/pipeline/conditional_stage.go
core/orderbook/v2/pipeline/context.go
core/orderbook/v2/pipeline/event_generation_stage.go
core/orderbook/v2/pipeline/locking_stage.go
core/orderbook/v2/pipeline/management_pipeline.go
core/orderbook/v2/pipeline/matching_stage.go

kaiachain/go-ethereum
core/orderbook/v2/pipeline/pipeline.go
core/orderbook/v2/pipeline/queue_update_stage.go
core/orderbook/v2/pipeline/settlement_stage.go
core/orderbook/v2/pipeline/trading_pipeline.go
core/orderbook/v2/pipeline/validation_stage.go
core/orderbook/v2/queue/buy_queue.go
core/orderbook/v2/queue/sell_queue.go
core/orderbook/v2/tpsl/activation_rule.go
core/orderbook/v2/tpsl/interfaces.go
core/orderbook/v2/tpsl/oco_controller.go
core/orderbook/v2/tpsl/trigger_manager.go
core/orderbook/v2/tpsl/triggers.go
core/orderbook/v2/types/balance.go
core/orderbook/v2/types/common.go
core/orderbook/v2/types/conditional.go
core/orderbook/v2/types/config.go
core/orderbook/v2/types/depth.go
core/orderbook/v2/types/errors.go
core/orderbook/v2/types/fee_retriever.go
core/orderbook/v2/types/market_rules.go
core/orderbook/v2/types/order.go
core/orderbook/v2/types/price_helpers.go

kaiachain/go-ethereum
core/orderbook/v2/types/request.go
core/orderbook/v2/types/snapshot.go
core/orderbook/v2/types/statedb.go
core/orderbook/v2/types/symbol.go
core/orderbook/v2/types/trade.go
core/types/account.go
core/types/token_balance.go
core/types/session.go
core/types/tx_input.go
core/orderbook/v2/interfaces/conditional.go
core/orderbook/v2/interfaces/core.go
core/orderbook/v2/interfaces/dispatcher.go
core/orderbook/v2/interfaces/market.go
core/orderbook/v2/interfaces/request.go
core/orderbook/v2/interfaces/response.go
core/orderbook/v2/metrics/metrics.go
core/orderbook/v2/system/system.go
core/orderbook/v2/balance/manager.go
core/orderbook/v2/balance/scaled_math.go
core/orderbook/v2/balance/settlement.go
core/orderbook/v2/engine/symbol_engine.go
core/orderbook/v2/matching/price_time_priority.go

kaiachain/go-ethereum
core/orderbook/v2/persistence/delta_writer.go
core/orderbook/v2/persistence/manager.go
core/orderbook/v2/persistence/recovery.go
core/orderbook/v2/persistence/snapshot_manager.go
core/orderbook/v2/pipeline/builder.go
core/orderbook/v2/pipeline/conditional_stage.go
core/orderbook/v2/pipeline/context.go
core/orderbook/v2/pipeline/event_generation_stage.go
core/orderbook/v2/pipeline/management_pipeline.go
core/orderbook/v2/pipeline/matching_stage.go
core/orderbook/v2/pipeline/pipeline.go
core/orderbook/v2/pipeline/queue_update_stage.go
core/orderbook/v2/pipeline/settlement_stage.go
core/orderbook/v2/pipeline/trading_pipeline.go
core/orderbook/v2/pipeline/validation_stage.go
core/orderbook/v2/pipeline/locking_stage.go
core/orderbook/v2/queue/buy_queue.go
core/orderbook/v2/queue/sell_queue.go
core/orderbook/v2/tpsl/activation_rule.go
core/orderbook/v2/tpsl/interfaces.go
core/orderbook/v2/tpsl/oco_controller.go
core/orderbook/v2/tpsl/trigger_manager.go

kaiachain/go-ethereum

- core/orderbook/v2/tpsl/triggers.go
- core/orderbook/v2/types/balance.go
- core/orderbook/v2/types/common.go
- core/orderbook/v2/types/config.go
- core/orderbook/v2/types/depth.go
- core/orderbook/v2/types/errors.go
- core/orderbook/v2/types/fee_retriever.go
- core/orderbook/v2/types/market_rules.go
- core/orderbook/v2/types/order.go
- core/orderbook/v2/types/price_helpers.go
- core/orderbook/v2/types/request.go
- core/orderbook/v2/types/snapshot.go
- core/orderbook/v2/types/statedb.go
- core/orderbook/v2/types/symbol.go
- core/orderbook/v2/types/trade.go
- core/types/account.go
- core/types/token_balance.go
- core/types/session.go

kaiachain/kaia-orderbook-dex-core

- precompiles/ArbTokenIssuer.go
- execution/gethexec/time_nonce.go

kaiachain/kaia-orderbook-dex-token-bridge-contracts

contracts/tokenbridge/libraries/L2GatewayToken.sol

Finding Categories

Categories	Description
Coding Style	Coding Style findings may not affect code behavior, but indicate areas where coding practices can be improved to make the code more understandable and maintainable.
Coding Issue	Coding Issue findings are about general code quality including, but not limited to, coding mistakes, compile errors, and performance issues.
Denial of Service	Denial of Service findings indicate that an attacker may prevent the program from operating correctly or responding to legitimate requests.
Inconsistency	Inconsistency findings refer to different parts of code that are not consistent or code that does not behave according to its specification.
Volatile Code	Volatile Code findings refer to segments of code that behave unexpectedly on certain edge cases and may result in vulnerabilities.
Logical Issue	Logical Issue findings indicate general implementation issues related to the program logic.
Design Issue	Design Issue findings indicate general issues at the design level beyond program logic that are not covered by other finding categories.

DISCLAIMER | CERTIK

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 purposes, nor may copies be delivered to any other person other than the Company, without CertiK's prior written consent in each instance.

This report is not, nor should be considered, an "endorsement" or "disapproval" of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any "product" or "asset" created by any team or project that contracts CertiK to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies 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 be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. CertiK's position is that each company and individual are responsible for their own due diligence and continuous security. CertiK's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

The assessment services provided by CertiK is subject to dependencies and 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. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.

ALL SERVICES, THE LABELS, THE ASSESSMENT 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, CERTIK HEREBY DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS. WITHOUT LIMITING THE FOREGOING, CERTIK 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, CERTIK MAKES NO WARRANTY OF ANY KIND THAT THE SERVICES, THE LABELS, THE ASSESSMENT 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, CERTIK PROVIDES NO WARRANTY OR

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 CERTIK NOR ANY OF CERTIK'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. CERTIK 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, ASSESSMENT 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, ASSESSMENT 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 CERTIK'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, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS.

THE REPRESENTATIONS AND WARRANTIES OF CERTIK 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 CERTIK 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 ASSESSMENT REPORTS OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.

Elevating Your Web3 Journey

Founded in 2017 by leading academics in the field of Computer Science from both Yale and Columbia University, CertiK is the largest blockchain security company that serves to verify the security and correctness of smart contracts and blockchain-based protocols. Through the utilization of our world-class technical expertise, alongside our proprietary, innovative tech, we're able to support the success of our clients with best-in-class security, all whilst realizing our overarching vision; provable trust for all throughout all facets of blockchain.