Security Audit Report

Kora Solana

Delivered: October 22nd, 2025
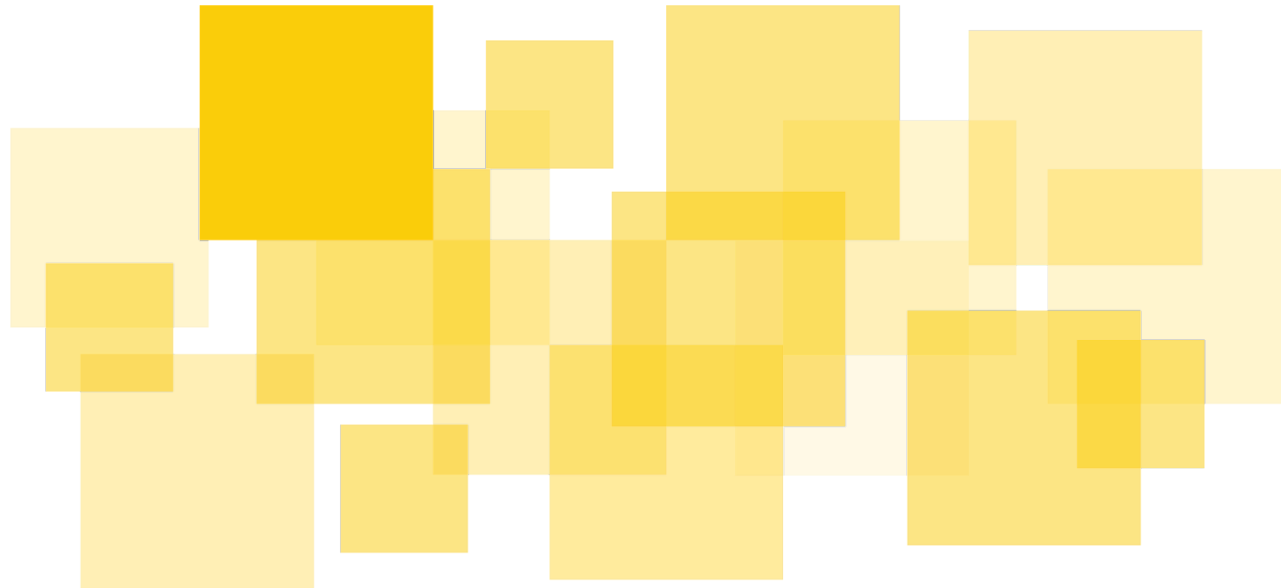
Prepared for Solana Foundation by

runtime verification

Table of Contents

Disclaimer

This report does not constitute legal or investment advice. You understand and agree that this report relates to new and emerging technologies and that there are significant risks inherent in using such technologies that cannot be completely protected against. While this report has been prepared based on data and information that has been provided by you or is otherwise publicly available, there are likely additional unknown risks that otherwise exist. This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system. This report is for informational purposes only and is provided on an 'as-is' basis, and you acknowledge and agree that you are making use of this report and the information contained herein at your own risk. The preparers of this report make no representations or warranties of any kind, either express or implied, regarding the information in or the use of this report and shall not be liable to you or any third parties for any acts or omissions undertaken by you or any third parties based on the information contained herein.

Blockchain technology is still a nascent software arena, and its deployment and public offering carry substantial risk.

Finally, the possibility of human error in the manual review process is very real, and we recommend seeking multiple independent opinions on any claims that impact a large quantity of funds.

Executive Summary

Solana Foundation engaged Runtime Verification, Inc. to conduct a security audit of the Kora protocol and its associated infrastructure code. The objective was to review the business logic and implementation in Rust and identify any issues that could cause the system to malfunction or be vulnerable to exploitation.

Kora is a Solana paymaster service that enables fee abstraction and gasless transactions on the Solana blockchain. The protocol allows end users to pay transaction fees in tokens other than SOL, with operators providing SOL to cover network fees and receiving payment in alternative tokens. The system features a Rust-based JSON-RPC server with multi-signer pool support, configurable pricing models, and fine-grained access control via allowlists and fee payer policies.

The audit was conducted over eight calendar weeks, from September 3rd to October 29th. Runtime Verification performed a design review to assess the protocol's high-level intent and security-critical invariants, followed by a comprehensive manual review of the codebase, threat modeling, and fuzz testing. We used automated static analysis tools to support this process by identifying potential vulnerabilities and code quality issues.

Scope

This audit covers only the code contained in the Kora GitHub repository. Within this repository, specific modules and components were highlighted as being in scope for this engagement. The repository, commit information, and operational assumptions are detailed below:

Kora GitHub Repository
https://github.com/solana-foundation/kora
Commit: c2843ac

The following modules were in scope for the audit:

- crates/lib/src/admin/
- crates/lib/src/fee/
- crates/lib/src/oracle/
- crates/lib/src/rpc_server/
- crates/lib/src/signer/
- crates/lib/src/token/
- crates/lib/src/transaction/
- crates/lib/src/usage_limit/
- crates/lib/src/validator/
- crates/lib/src/cache.rs
- crates/lib/src/config.rs
- crates/lib/src/error.rs
- crates/lib/src/state.rs

The audit is limited to the artifacts listed above. Off-chain components, third-party dependencies, deployment and upgrade scripts, and any client-side logic are excluded from the scope of this engagement.

Our security analysis is based on the following operational assumptions. If any of these assumptions are violated, the protocol's security guarantees may no longer hold, and additional review may be required.

- Solana RPC Integrity
  Solana RPC endpoints are trusted and provide accurate on-chain data without manipulation or censorship.

- Remote Signer Security
  Remote signer services are securely configured and properly authenticated, and their APIs behave as documented.

- Cache Security
  Redis cache is properly secured, not publicly accessible, and maintains data integrity throughout the system lifecycle.

- Configuration Validity
  Configuration files (kora.toml, signers.toml) contain valid, properly formatted data and are managed through secure operational procedures.

- Operator Best Practices
  Operators follow security best practices documented in AUTHENTICATION.md and maintain secure key management for all signer types.

Methodology

Although manual code review cannot guarantee the discovery of all possible security vulnerabilities, as noted in our Disclaimer, we followed a systematic approach to make this audit as thorough and impactful as possible within the allotted timeframe.

The audit engagement lasted eight calendar weeks, from September 3rd to October 29th, 2025, and began with a focused design review. We allocated the first two weeks to analyze the architecture and intended functionality of the Kora system. This included reasoning about the interactions between the protocol components and identifying properties that should be upheld throughout the system's lifecycle. Following the design review, we conducted a thorough manual code review of the in-scope modules, progressing systematically.

The engagement included specialized focus areas such as fee calculation, arithmetic safety with particular attention to integer overflow vulnerabilities, multi-signer key management across different backend providers (Memory, Turnkey, Privy), and price oracle integration with Jupiter API for token pricing.

Findings presented in this report stem from a combination of:

- Manual inspection of the Rust source code.
- Design-level reasoning about system interactions and security invariants.
- Fuzz testing to identify edge cases and potential vulnerabilities.

In addition to identifying bugs and vulnerabilities, we also evaluated arithmetic safety patterns, error handling strategies, reviewed edge-case scenarios, and provided recommendations for code clarity and safety improvements.

Throughout the engagement, we held internal discussions among auditors to cross-review the findings and validate risk assessments.

Overview

Kora is a Solana paymaster that allows users to submit transactions while paying fees in a token of their preference. Kora serves as an intermediary, signing the transaction as the fee payer and handling the network fees in SOL, while accepting payment from the user in SPL tokens. This creates a smoother experience for users of applications that primarily hold non-native tokens, who can avoid having to convert their tokens into SOL in order to pay gas fees.

The Kora node operator configures their deployment via a `kora.toml` file, allowing them to customize various aspects of the system, including:

- The price model determining how the user will be charged per transaction.
- What operations will be available to the users.
- The strategy that will be used for selecting a signer from the signing pool.
- Restrictions on the transactions being processed, including what programs, tokens and accounts it can interact with.
- Extra functionalities such as authentication, caching and usage limits.

A Kora node is deployed as an RPC server, allowing users to send requests via the Kora RPC client in order to estimate transaction fees, generate payment instructions to be included in a transaction, and have the transaction signed and submitted to the Solana network. Before signing a transaction, Kora performs validation to ensure that the transaction complies with the restrictions specified in the configuration file and that it includes sufficient payment to cover the Kora fees in accordance with the price model.

Critical Properties

The following correctness and security properties must be upheld by the Kora implementation. We also list possible attack vectors that need to be protected against.

Network Security

- Authentication layers (API or HMAC), if enabled, must block requests from unauthenticated users.

Private Key Security

- Signer private keys are not leaked by the system.

Fees

- Fees charged by the system match the pricing model specified in the config file.
- If using the `margin` pricing model, fees match the fee calculation formula.
- The system must refuse to sign any transaction that does not include a payment of at least the required amount to the correct payment address, in one of the tokens accepted for payment. Note that if the pricing model is `free`, the required amount is 0, so no payment is required.
- If using the `margin` pricing model, the amount transferred by the user to the payment address is at least the amount paid in fees by the fee payer.

Config and Transaction Validation

- Only transactions that match the parameters specified in the config file are signed.
- All settings specified in the config file must be correctly enforced during system execution.

Caching

- The cache must not return outdated values when a more recent value is required.
- Cache is not vulnerable to manipulation by the user.

Possible Attack Vectors

- Oracle manipulation.
- Denial of service.
- Precision errors in fee calculation.
- Numeric overflows.
- Lookup table address validation bypass.
- Instructions inserted into the transaction siphoning funds from the Kora node.

Module Summaries

Below we present summaries of the central modules in the Kora library, including diagrams of the different submodules, descriptions of the main data structures and functions defined in each module, and their interactions with other modules. We also include shorter descriptions of the additional modules at the end.

Fee Module (`fee`)
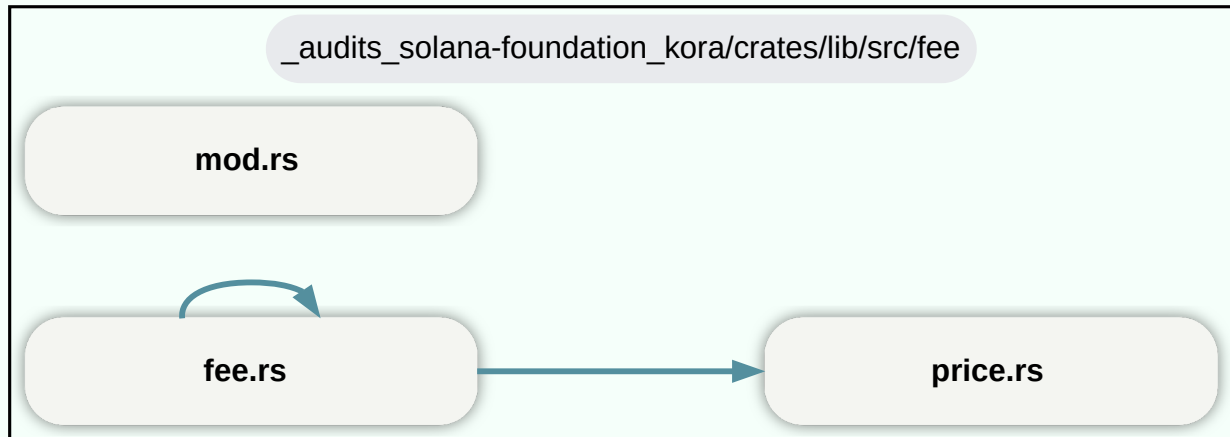
The Fee module implements a six-component fee structure that calculates transaction costs across multiple token types. As the central pricing authority in Kora's architecture, this module integrates with external oracle systems for real-time token valuation while supporting three pricing models: Free, Margin-based, and Fixed.

The architecture handles financial scenarios including account creation fees, transfer fees, and Kora service charges, providing transparency through detailed component breakdowns while maintaining efficiency through caching mechanisms.

Data Structures

- `TotalFeeCalculation` holds the breakdown of all six fee components (base fee, account creation, signature, outflow, payment instruction, transfer fees) along with the total amount in lamports.
- `PriceConfig` wraps the price model configuration and provides methods to apply Free, Margin, or Fixed pricing strategies to calculated fees.
- `PriceModel` is an enum with three variants: Free (zero cost), Margin (percentage markup), and Fixed (oracle-based token conversion).
- `FeeConfigUtil` and `TransactionFeeUtil` are empty structs that function as namespaces for fee calculation and transaction fee estimation functions respectively.

Fee Calculation Formula

```
total_fee_lamports = base_fee + account_creation_fee + kora_signature_fee
                   + fee_payer_outflow + payment_instruction_fee + transfer_fee_amount
```

Main Functions

- `FeeConfigUtil::estimate_kora_fee`
  - Calculates all six fee components then applies the configured price model to return accurate cost breakdown.

- For `PriceModel::Free`, returns all zero fee components without expensive RPC calls or account lookups.
- For non-free models, delegates to `estimate_transaction_fee` for detailed component analysis, then applies price model via `PriceConfig::get_required_lamports`.
- Components include base fee, account creation, Kora signature, fee payer outflow, payment instruction, and transfer fees.
- Returns `TotalFeeCalculation` with all six fee components, based on current network conditions and transaction analysis.
- Returns an error in any of the following cases:
    - Configuration retrieval fails (`get_config()` returns error).
    - Fee component calculation fails (`estimate_transaction_fee()` returns `RpcError`, `AccountNotFound`, or `FeeEstimationFailed`).
    - Price model application fails (`get_required_lamports()` returns `ValidationError` or `UnsupportedFeeToken`).
- `FeeConfigUtil::estimate_transaction_fee`
    - Computes comprehensive fee breakdown for transactions.
    - Components calculated in order:
        - Base transaction fee via RPC (`TransactionFeeUtil::get_estimate_fee_resolved`).
        - Account creation fees for ATA creation (`get_associated_token_account_creation_fees`).
        - Kora signature fee if fee payer not in transaction signers (`is_fee_payer_in_signers`). Adds `LAMPORTS_PER_SIGNATURE` (5000 lamports).
        - Fee payer outflow for System Program operations using saturating arithmetic (`calculate_fee_payer_outflow`).
        - Payment instruction fee if payment required (`has_payment_instruction`). Returns 0 if payment found, otherwise 50 lamports (`ESTIMATED_LAMPORTS_FOR_PAYMENT_INSTRUCTION`).
        - Transfer fees at current epoch for Token2022 transfers (`calculate_transfer_fees`).
    - Aggregates components using regular addition (not saturating).
- `FeeConfigUtil::calculate_fee_payer_outflow`
    - Calculates total SOL the fee payer sends out in System Program operations.
    - Parses System Program instructions using `IxUtils::parse_system_instructions` to identify transfers, account creation funding, and nonce withdrawals.
    - Uses saturating arithmetic for individual operations to prevent underflow.
    - Only tracks System Program operations; other program types (SPL, custom programs) are excluded from outflow calculation.
    - Returns total outflow amount in lamports for inclusion in fee calculation.
- `FeeConfigUtil::has_payment_instruction`
    - Detects SPL transfer to Kora payment destination.
    - Iterates through transaction instructions to find transfers matching payment address and allowed token mints.
    - Returns 0 if payment instruction found, otherwise returns estimated payment instruction fee (50 lamports).
    - Uses `get_payment_instruction_info` helper for cache-first account lookups and program detection.
    - Supports both SPL Token and Token2022 program transfers.
- `TransactionFeeUtil::get_estimate_fee_resolved`
    - Estimates transaction fees for Legacy and V0 transactions.
    - Calls `rpc_client.get_fee_for_message` directly for Legacy transactions.
    - For V0 transactions, creates Legacy message using `Message::new_with_compiled_instructions` with resolved lookup table addresses for RPC fee calculation.
    - Returns base network fee in lamports without Kora-specific additions.
    - Essential for fee calculation as it provides the Solana network's base transaction cost.

- `PriceConfig::get_required_lamports`
  - Applies configured price model to calculated fee amounts.
  - For Free model, returns zero regardless of calculated fees.
  - For Margin model, multiplies fee by `(1.0 + margin)` using f64 arithmetic, then casts to u64.
  - For Fixed model, converts token amount to lamports via `TokenUtil::calculate_token_value_in_lamports` using oracle price data.
  - Returns final fee amount in lamports after price model application.

Error Conditions

- `FeeConfigUtil::estimate_kora_fee`
  - RPC Client Failure: `get_fee_for_message()` returns RPC error during base fee calculation.
  - Cache miss with RPC timeout during account retrieval exceeds configured bounds.
  - Empty instruction set or malformed message during fee parsing (HIGH - validation bypass risk).
  - Price data staleness or values outside configured bounds for Fixed price model (HIGH - fund drain risk).
  - Token2022 mint account fails to deserialize during transfer fee calculation.
  - Unable to determine token program type for account size calculation.
  - Token2022 transfer fee calculation fails due to epoch boundary conditions.
  - Unable to resolve payment destination account for validation.

- `FeeConfigUtil::estimate_transaction_fee`
  - Transaction signature count exceeds maximum allowed signatures.
  - Instruction references program not in allowed programs list.
  - V0 transaction lookup table index out of bounds during resolution.
  - Malformed System Program instruction data fails parsing.
  - Invalid instruction format during outflow calculation.
  - Unable to locate fee payer in transaction signer set.
  - System Program outflow exceeds configured maximum limits.
  - Integer overflow during final fee component aggregation.

- `PriceConfig::get_required_lamports`
  - Percentage margin calculation overflows for large base amounts.
  - Oracle price conversion fails due to decimal precision limits.
  - Invalid price model configuration prevents calculation.
  - Token decimal places inconsistent between mint and oracle data.

Interaction with Other Modules

The fee module maintains dependencies on oracle module for price data, token module for metadata, cache module for account state, and state module for configuration. Integration points with transaction module for instruction parsing and rpc_server module for client responses.

- Calls `oracle` module via `PriceSource` for Jupiter API price discovery.
- Calls `token` module for Token2022 transfer fee calculations and decimal handling.
- Calls `transaction` module for instruction parsing and transaction resolution.
- Calls `cache` module for account lookups with RPC fallback.
- Calls `state` module for global configuration access.
- Called by `rpc_server` methods for fee estimation endpoints.

- Called by `validator` module for fee validation checks.

Security Considerations

- Integer overflow risk in main fee totaling: uses regular `+` addition without overflow protection.
- Uses fixed 50 lamport estimate for priority fees when no payment instruction found.
- ATA creation fees calculated before payment instruction analysis, potential double-counting.
- Uses floating-point arithmetic for margin calculations.
- Fee calculations rely on external price feeds without implementing explicit staleness bounds or fallback mechanisms at the fee layer.

Properties and Invariants

- Total fee represents exact sum of six components:
  `base_fee + account_creation_fee + kora_signature_fee + fee_payer_outflow + payment_instruction_fee + transfer_fee`
  .
- All fee components must be non-negative integers.
- Total fees cannot exceed safe limits.
- If any component calculation fails, entire estimation returns error rather than partial results.
- If `PriceModel::Free`, short-circuits to return all zero components early without RPC calls or account lookups.
- Kora signature fee (5000 lamports) added only if fee payer not among transaction signers per `is_fee_payer_in_signers` check, `0` otherwise.
- Fee payer outflow tracks only System Program operations (transfers, CreateAccount, nonce withdrawals).
- Token2022 transfer fees are epoch-aware and should be calculated at current network epoch via mint configuration.
- The fee module ensures the following:
  - Fee calculations always finish in reasonable time.
  - Valid transactions get complete fee estimates.
  - All failure conditions are reported without silent failures.

RPC Server Module (`rpc_server`)
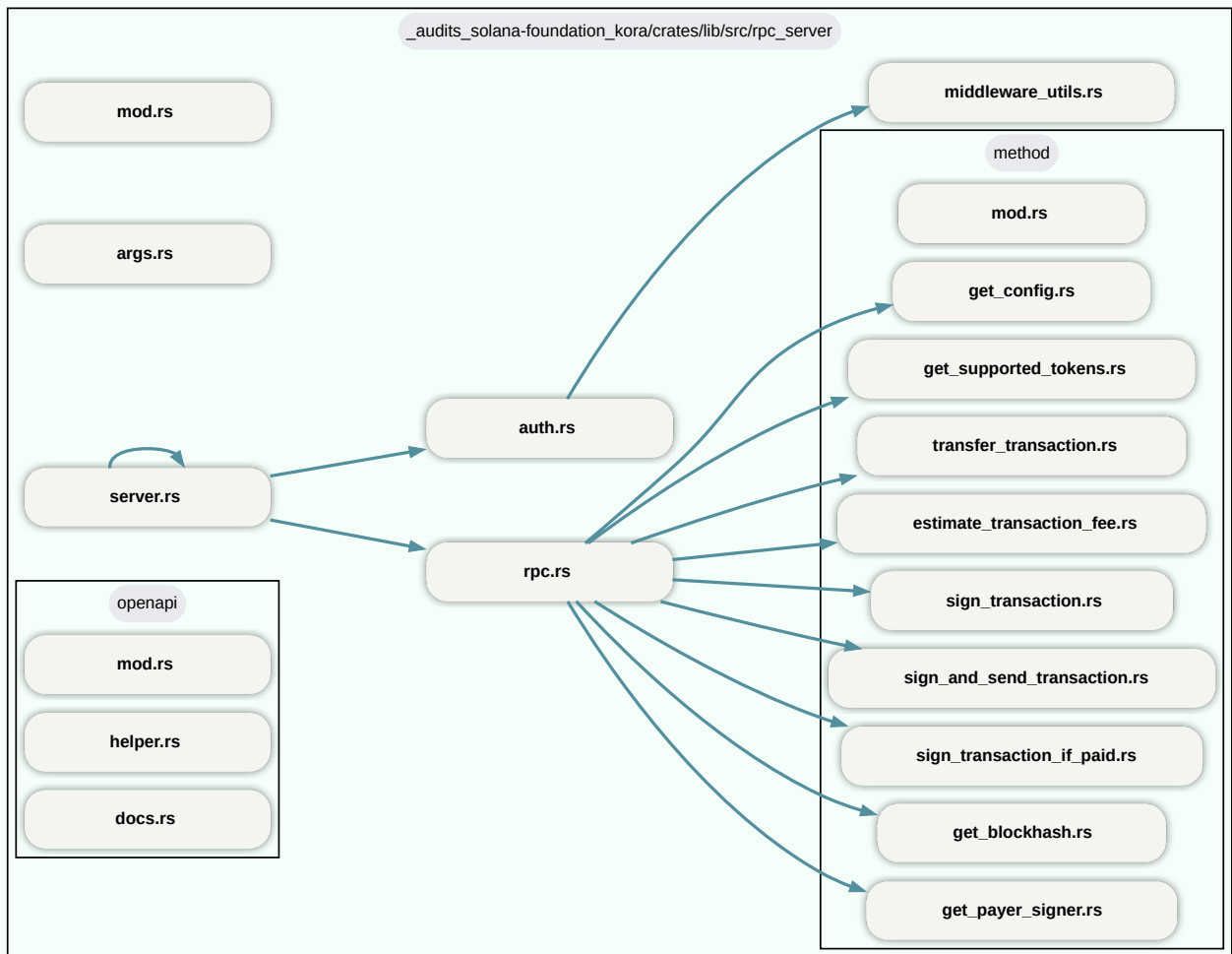
This module contains the functionality for running the Kora RPC server, representing the central coordination point of Kora's operation. It can be split into the following key submodules:

- `server` : Initializes the RPC server, utilizing the Tower library (https://crates.io/crates/tower) for building the multiple layers of the middleware stack.
- `auth` : Implements functionality for API and HMAC authentication layers.
- `method` : Implements the methods that can be called on the RPC server.
- `rpc` : Defines the `KoraRpc` struct that will handle the requests sent to the server and call the appropriate method.

The methods provided by the server are the following:

- `estimate_transaction_fee` : Calculates the estimated fee required for a given transaction.
- `get_blockhash` : Queries the latest blockhash from the Solana RPC client.
- `get_config` : Retrieves information from the Kora global config.
- `get_payer_signer` : Gets the next signer from the signer pool and the payment destination.
- `get_supported_tokens` : Gets the list of allowed tokens as defined in the Kora global config.
- `sign_transaction` : Signs the given transaction, trying to use the requested signer if provided.

- `sign_transaction_if_paid` : As `sign_transaction` , but checks first if the transaction includes a payment of the transaction fee.
- `sign_and_send_transaction` : As `sign_transaction` , additionally submitting the transaction to the Solana network.
- `transfer_transaction` : Builds a transfer transaction according to the given parameters, returning the signed transaction. Intended to be used by a client to build the payment instruction to include in the transaction.

Data Structures

- `KoraRpc` holds a pointer to the Solana RPC client and is responsible for handling the requests to the Kora RPC server, calling the appropriate corresponding function defined in the `method` submodule.
- The `auth` submodule defines the `ApiKeyAuthLayer` and `HmacAuthLayer` data structures, which represent the respective authentication layers integrated into the server if each of those options is configured.
- Each method in the `methods` submodule defines corresponding `...Request` and `...Response` data structures, containing the data sent to the server as part of the request and the data returned by the server as a response.

`server` Submodule

- `run_rpc_server`
  - Starts up RPC server on given port.
  - Initializes usage limiter. (`UsageTracker::init_usage_limiter`)
  - Enables HTTP support.
  - Registers the methods from `method` module that are enabled by the global config.
  - Middleware stack for the server comprised of several layers:
    - `ProxyGetRequest` layer
    - Rate-limit layer
    - Metrics handler layer (optional, if metrics enabled in config)
    - CORS layer
    - Metrics collection layer (optional, if metrics enabled in config)
    - Authentication layer for API key (optional, if API key provided in config or environment variable)
    - Authentication layer for HMAC (optional, if HMAC secret provided in config or environment variable)

`method` Submodule

- `estimate_transaction_fee`
  - Given a transaction, returns the estimated transaction fee in lamports, as well as in a specific token if requested.
  - Calculation is made based on a particular signer. This signer can be provided in the request, otherwise it is selected from the signer pool based on the configured selection strategy. The public key of this signer is returned along with the fee.
  - The payment address is also returned. This will be the address specified in the global config, or the signer address if none has been specified.
  - Does not submit the transaction or change on-chain state, only estimates the fees.
  - If signer is selected using a round-robin strategy, will increment the next signer index.

- `get_blockhash`
  - Returns a recent blockhash from the Solana RPC client.
  - Returned blockhash must not be empty.

- `get_config`
  - Returns list of fee payer public keys from the signer pool, as well as the validation parameters and list of enabled methods from the global config.

- `get_payer_signer`

- Returns the next signer from the signer pool based on the configured selection strategy.
- If using a round-robin strategy, will increment the next signer index.
- `get_supported_tokens`
  - Returns the list of allowed tokens.

- `sign_transaction`
  - Returns the signed transaction, along with the signature and the signer's public key.
  - If usage limiter is enabled, checks transaction usage limit for the sender, and returns an error if it's exceeded.
  - If signer key is specified in the request, uses this as the signer. Otherwise, selects the next signer from the pool.

- `sign_transaction_if_paid`
  - Same as `sign_transaction`, but returns an error if the required fee is not paid by the transaction.

- `sign_and_send_transaction`
  - Same as `sign_transaction`, but also submits the transaction to the Solana RPC client, waiting for confirmation.

- `transfer_transaction`
  - Builds a transfer transaction with the given parameters.
  - Uses a `TransactionValidator` to validate that the source and destination addresses are not in the list of disallowed accounts.
  - Token can be either native SOL or an SPL token. In the latter case, uses the `token` module to build the instructions.
  - Message blockhash is the latest blockhash obtained from the Solana RPC client.
  - Validates the transaction with the `TransactionValidator` as well.
  - Signature is added to the transaction at the fee payer position.

Interactions with Other Modules

As Kora's central coordination point, the `server` module calls all other modules either directly or indirectly. Most significantly:

- Calls the `fee` module to estimate transaction fees.
- Calls the `signer` module to retrieve signers from the signer pool in order to sign transactions.
- Calls the `transaction` module to resolve and sign user-submitted transactions, or to build a payment transaction.
- Calls the `validation` module to validate transactions.

Security Considerations

- Authentication layers should be properly configured to prevent unauthorized access.
- Validation must be performed at the appropriate points before each operation is executed.
- Protection against DoS attacks.

Properties and Invariants

- Liveness check should not require authentication.
- If an authentication layer is configured, requests from unauthenticated clients should not be processed.

Signer Module (`signer`)
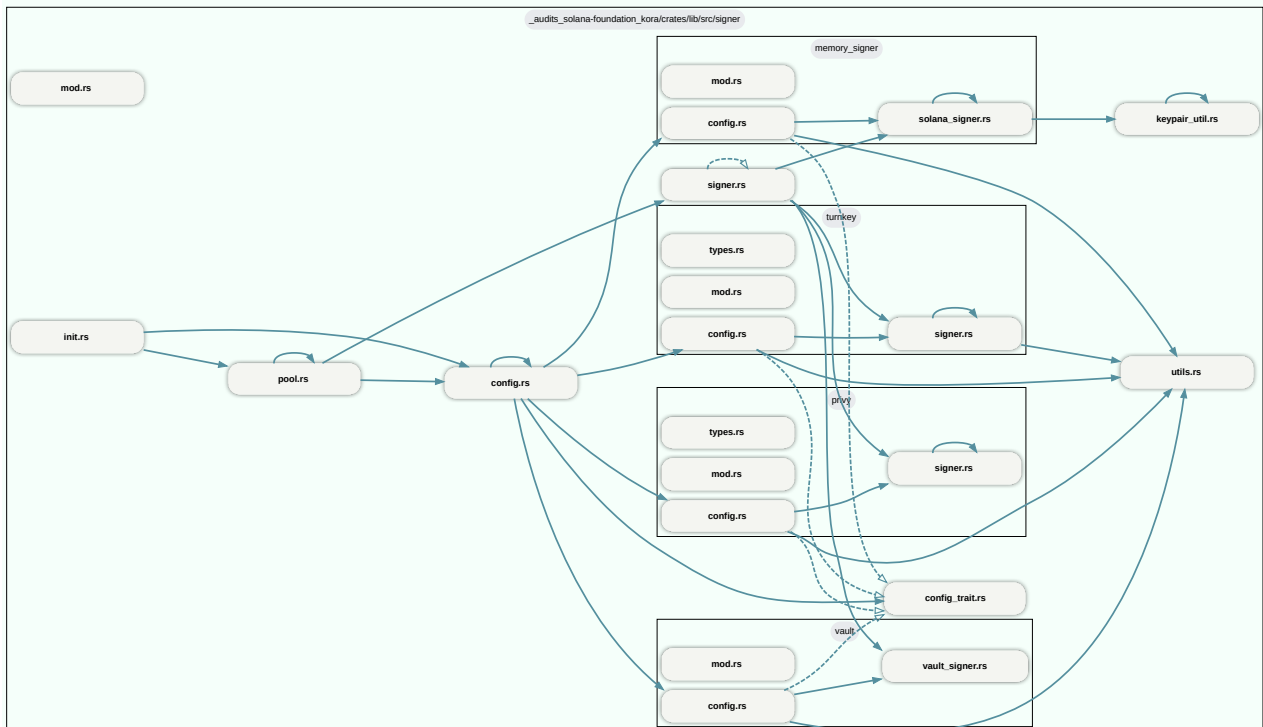
This module implements the functionality in Kora dealing with the signers, including initializing the signer pool based on the signer config file, selecting a signer from the pool, and performing the actual signing of messages depending on the signer type. Each of the signer types supported by Kora is implemented in its own submodule:

- `memory_signer` directly uses a Solana private key provided in an environment variable or passed as a CLI argument.
- `privy` uses a key managed by a Privy account.
- `turnkey` uses a key managed by a Turnkey account.
- `vault` uses a key stored in a HashiCorp Vault server.

Other key submodules include the `config` submodule, which loads and validates the signer config provided in the `signers.toml` file, and the `pool` submodule, which implements and manages the signer pool, being able to retrieve signers from their public keys or selecting a signer based on the configured selection strategy.

Data Structures

- `KoraSigner` is an enum representing the different signer types supported by Kora: `Memory`, `Turnkey`, `Vault` and `Privy`. Each variant contains a different struct representing the signer implementation from the corresponding submodule.
- `SelectionStrategy` is an enum representing the three possible signer selection strategies: round-robin, random or weighted.
- `SignerWithMetadata` is a struct containing a `KoraSigner` along with accompanying metadata, including its human-readable name, its weight (if using the weighted selection strategy) and the timestamp of when it was last selected.
- `SignerPool` is a collection of signers together with a signer selection strategy and additional information used to implement the strategy (such as the current index for the round-robin strategy). It is initialized from the `signers.toml` config file.

Main Functions

- `init_signers`

- Initializes signer pool based on the RPC args.
- If the `--no-load-signer` option was provided, skips initialization.
- Otherwise, loads signer pool config file from the provided path and creates a signer pool based on it, saving it in the global pointer in the `state` module.

- `SignerPoolConfig::validate_signer_config`
  - Validates the signer pool configuration, returning an error in any of the following cases:
    - The list of signers is empty.
    - A signer has an empty name.
    - One of the required environment variables has not been provided for a signer (according to its signer type).
    - Two signers have the same human-readable name.
    - A signer is given a weight of 0 for the weighted strategy.

- `SignerPool::get_next_signer`
  - Returns the next signer based on the configured signer selection strategy.
  - If using the round-robin strategy, this increments the index.

- `SignerPool::get_signer_by_pubkey`
  - Returns the signer with the given public key.

- `KoraSigner::sign_solana`
  - Signs a transaction in the format expected by Solana, delegating to the proper implementation depending on the signer type.

Interactions with Other Modules

- Called by `admin` module to iterate over the signers and create ATAs for all of them.
- Called by `rpc_server` module to select signers for fee estimation and transaction signing.
- Called by `transaction` module to sign transactions.
- Calls `validator` module to perform validation checks on the signer pool configuration.

Security Considerations

- Signer pool holds private keys of the signers and returns them on some of its functions. It's important to ensure that these cannot be leaked to the user.

Token Module (`token`)
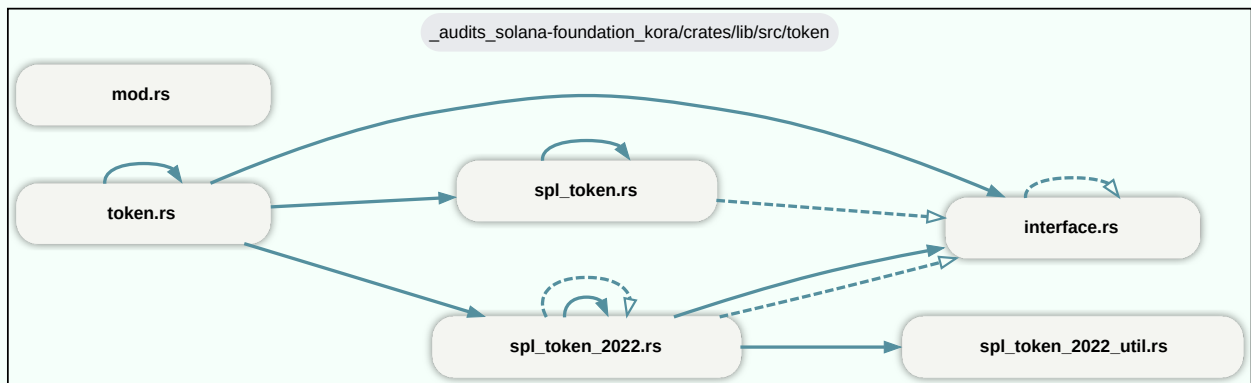
This module defines a common interface for both SPL and 2022 tokens, consisting of the following traits:

- `TokenState` is used as an interface for ATAs, allowing to retrieve information such as the owner and the current balance.
- `TokenMint` is used as an interface for a token's mint, allowing to retrieve information such as the mint authority and the total supply.
- `TokenInterface` is used as an interface for the token program (either the original SPL Token Program or the Token 2022 Program), with functionality to unpack mint and token account data and to build instructions for account creation and token transfers.

Each of these traits is implemented for appropriate data structures in the `spl_token` and `spl_token_2022` submodules. The module also includes a number of utility functions in the `token` submodule for operating with tokens, encompassing functionality such as converting between tokens and lamports, validating Token2022 extensions and checking that a transaction includes a required token payment.

Main Functions

- `TokenUtil::calculate_token_value_in_lamports`

  - Convert an amount in a token's base units to lamports, given the token price as provided by the given oracle.

- `TokenUtil::calculate_lamports_value_in_token`

  - Convert an amount in lamports to a token's base units, given the token price as provided by the given oracle.

- `TokenUtil::process_token_transfer` (note: does not strictly conform to spec currently)

  - Validates that the transaction includes a payment to the expected destination account amounting to the required lamport value.
  - If there are no errors, returns `Ok(true)` if there is a payment to the expected destination account using a supported payment token, and this payment is of value greater than or equal to the required amount of lamports. Returns `Ok(false)` otherwise.

Interactions with Other Modules

- Called by the `admin` module to retrieve the token programs to create ATAs for.
- Called by the `fee` module to calculate token prices and convert between tokens and lamports.
- Called by the `validation` module to validate that a payment of the Kora fees is included in the submitted transaction.
- Calls the `oracle` module to query token prices.
- Calls the `transaction` module to parse token instructions.

Security Considerations

- Token price determines the amount that the user will be required to pay, so it's important that it is accurate in the calculations. Oracle manipulation, for instance, could allow a user to effectively underpay the Kora node.
- Floating point imprecision in the calculations also affects the amount the user is required to pay. Even small errors in the user favor could add up over multiple transactions and be exploited by malicious users.

Properties and Invariants

- Conversion between token and lamports must follow the correct exchange rate.
- Conversion should be reversible: converting from tokens to lamports and back should give back the original value (within some small margin of error to account for floating-point imprecisions), and vice-versa.
- `process_token_transfer` must only approve a transaction if it has a correct payment of the required fees.

Transaction Module (`transaction`)
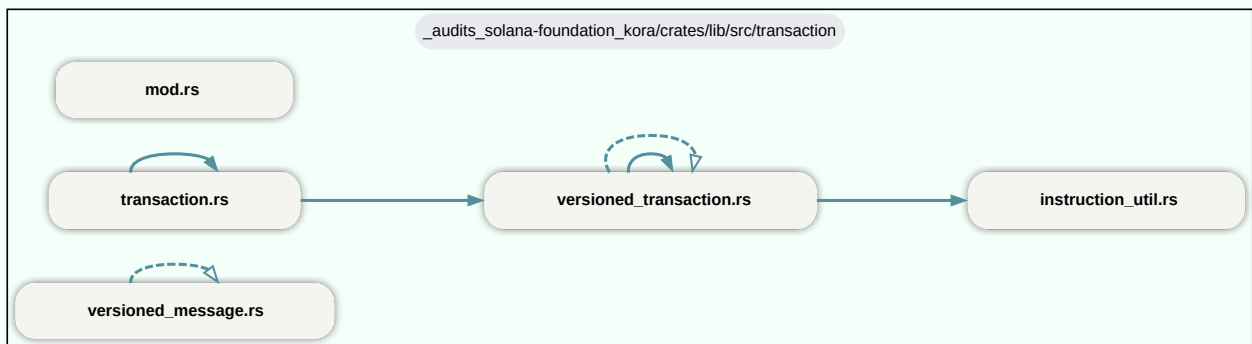
The Transaction module processes both Legacy and V0 Solana transactions. It takes raw transaction data from clients and turns it into validated, signable transactions ready for execution.

Data Structures

- `VersionedTransactionResolved` holds the original transaction along with resolved account keys (including lookup table addresses) and all instructions (outer and inner).
- `ParsedSystemInstructionType` and `ParsedSPLInstructionType` are enums that classify different instruction types for System Program and SPL/Token2022 operations respectively.
- `ParsedSystemInstructionData` and `ParsedSPLInstructionData` contain the structured data extracted from parsed instructions, including account references and operation-specific fields.
- `VersionedTransactionOps` is a trait that defines transaction signing operations for `VersionedTransactionResolved`.
- `TransactionUtil`, `IxUtils`, and `LookupTableUtil` are structs that function as namespaces for transaction operations, instruction utilities, and lookup table resolution respectively.

Main Functions

- `VersionedTransactionResolved::from_transaction(tx, rpc_client, sig_verify: bool)`
  - Creates a fully resolved transaction from a raw VersionedTransaction with RPC client.
  - This function should be the first call in any method that uses transactions, to ensure that all later validation and processing includes resolved lookup table addresses.

- For V0 transactions, resolves lookup table addresses via `LookupTableUtil::resolve_lookup_table_addresses` and extends `all_account_keys` with resolved addresses.
- Fetches inner instructions via `fetch_inner_instructions` using transaction simulation.
- On simulation failure, returns `InvalidTransaction` error.
- Returns `VersionedTransactionResolved` with complete account keys and all instructions (outer + inner) for comprehensive analysis.
- Returns error in any of the following cases:
    - Transaction simulation fails (`fetch_inner_instructions` returns `RpcError` or `InvalidTransaction`)
    - Lookup table resolution fails for V0 transactions (`AccountNotFound` or `RpcError`)
    - Address lookup table deserialization fails (`InvalidTransaction`)

- `VersionedTransactionResolved::from_kora_built_transaction`

    - Creates resolved transaction from Kora-built transactions without external RPC calls.
    - Used for transactions constructed internally where lookup tables are not expected.
    - Sets `all_account_keys` to static keys only and `all_instructions` from message instructions.

- `VersionedTransactionOps::sign_transaction`

    - Signs transactions using configured KoraSigner with validation via `TransactionValidator` .
    - Process flow: validation -> blockhash update -> fee validation -> signature generation -> signature placement.
    - Performs validation via `TransactionValidator::validate_transaction` .
    - Updates recent blockhash using finalized commitment if signatures array is empty.
    - Validates lamport fees before signing via `validate_lamport_fee` to ensure fee bounds compliance.
    - Uses `find_signer_position` to determine correct signature placement in signatures array.
    - Returns signed transaction with valid signature at correct position, preserving existing signatures.
    - Returns an error in any of the following cases:
        - Transaction validation fails (`TransactionValidator::validate_transaction` returns `ValidationError`).
        - Fee validation fails (`validate_lamport_fee` returns `FeeEstimationFailed`).
        - Signer backend fails (`KoraSigner` returns `SigningError`).
        - Recent blockhash update fails (`RpcError`).

- `VersionedTransactionOps::sign_transaction_if_paid`

    - Conditional signing with payment verification for fee-required transactions.
    - Process flow: fee estimation -> payment validation -> conditional signing.
    - Estimates fees using `FeeConfigUtil::estimate_kora_fee` to determine required payment amount.
    - Validates payment via `TransactionValidator::validate_token_payment` to ensure sufficient token transfer to payment destination.
    - Delegates to `sign_transaction` only after payment validation passes.
    - Returns signed transaction if payment adequate, otherwise returns payment validation error.
    - Returns error if fee estimation fails (`FeeEstimationError`), payment validation fails (`PaymentValidationError`), or signing fails (`SignerError`).

- `TransactionUtil::decode_b64_transaction`

    - Decodes base64 transaction data with backward compatibility fallback.
    - Attempts `VersionedTransaction` deserialization first for modern format support.
    - Falls back to `Transaction` deserialization with conversion to `VersionedTransaction` for legacy compatibility.
    - Handles both serialization formats to support diverse client implementations.
    - Supports both standard base64 and URL-safe base64 encoding variants.

- Returns an error in any of the following cases:
  - Invalid base64 format (`InvalidTransaction`).
  - Transaction deserialization fails for both `VersionedTransaction` and `Transaction` formats (`InvalidTransaction`).
  - Legacy to versioned transaction conversion fails (`InvalidTransaction`).

- `TransactionUtil::encode_b64_transaction`

  - Encodes `VersionedTransaction` to base64 format for client transmission.
  - Serializes transaction using bincode for compact representation.
  - Applies standard base64 encoding for compatibility with client libraries.
  - Maintains symmetric operation with `decode_b64_transaction` for round-trip integrity.
  - Used for returning signed transactions to client applications.
  - Returns base64-encoded transaction string for client consumption.

Error Conditions

- `VersionedTransactionResolved::from_transaction`

  - Transaction simulation request exceeds configured timeout bounds.
  - Simulation fails due to account lock conflicts with concurrent transactions.
  - Transaction exceeds compute unit limits during simulation execution.
  - RPC fails to fetch address lookup table accounts during V0 resolution.
  - Address lookup table account data fails to deserialize properly.
  - Lookup table index exceeds available address count.
  - Resolved addresses create duplicate entries in account key array.
  - Unable to compile resolved addresses into Legacy message for fee estimation.
  - Signature verification settings conflict with simulation requirements.

- `VersionedTransactionOps::sign_transaction`

  - `TransactionValidator::validate_transaction()` rejects transaction structure.
  - Calculated lamport fees exceed configured maximum allowed amounts.
  - KoraSigner backend returns signing error.
  - Recent blockhash is too old for transaction validity requirements.
  - Unable to determine correct signature placement in signatures array.
  - Signed transaction exceeds maximum transaction size limits.
  - Required accounts are locked by concurrent transactions.
  - Fee payer account lacks sufficient SOL for transaction execution.

- `VersionedTransactionOps::sign_transaction_if_paid`

  - `validate_token_payment()` rejects insufficient payment amounts.
  - Payment token not in allowed SPL paid tokens list.
  - Payment directed to incorrect destination address.
  - `estimate_kora_fee()` fails during payment validation.
  - Multiple payment transfers fail to aggregate to required amount.

- `TransactionUtil::decode_b64_transaction`

  - Transaction data exceeds maximum deserialization size.
  - Unable to convert Legacy Transaction to VersionedTransaction.
  - Instruction data exceeds maximum allowed instruction size.
  - Transaction contains more account keys than maximum allowed.

- Signature array length mismatches message header specification.

- `IxUtils::parse_system_instructions`

  - Unrecognized System Program instruction variant.
  - System transfer amount exceeds u64 maximum value.
  - `CreateAccount` instruction with invalid account parameters.
  - Nonce-related instruction with incorrect account structure.
  - Assign instruction referencing invalid program ownership.

- `IxUtils::parse_token_instructions`

  - Instruction targets wrong token program.
  - Token transfer instruction with invalid account structure.
  - Token instruction requires authority not present in signers.
  - Instruction targets frozen token account.
  - Token amount decimals inconsistent with mint configuration.

Interaction with Other Modules

The dependencies of transaction module include validator module for transaction validation, fee module for cost calculation, signer module for cryptographic operations, and cache module for account state. External dependencies on Solana RPC for simulation and lookup table data.

- Calls `validator` module via `TransactionValidator` for validation.
- Calls `fee` module via `TransactionFeeUtil` and `FeeConfigUtil` for fee operations.
- Calls `signer` module via `KoraSigner` for transaction signing.
- Calls `cache` module via `CacheUtil` for account and lookup table fetching.
- Calls `state` module for configuration access.
- Called by `rpc_server` methods for all transaction RPC endpoints.
- Uses `instruction_util::IxUtils` for instruction parsing and compilation.

Security Considerations

- Uses `bs58::decode(&ix.data).into_vec().unwrap_or_default()` potentially masking decoding failures.
- Resolved addresses used without additional validation against allow/disallow lists.
- Returns `InvalidTransaction` on simulation errors with no degradation that could hide malicious inner instructions.

Properties and Invariants

- `VersionedTransactionResolved` maintains immutable `all_account_keys` ordering after resolution.
- V0 transactions with lookup tables resolve to deterministic address lists based on lookup table contents and index ordering.
- Transaction encoding/decoding via `encode_b64_transaction` / `decode_b64_transaction` are symmetric operations preserving all transaction data.
- Signature placement preserves existing signatures while adding new ones at calculated positions.
- Transaction message integrity must be preserved through all resolution and signing operations.
- Inner instruction parsing only occurs when transaction simulation returns success (no error field in simulation result).
- The transaction module maintains these invariants:
  - Address lookup table resolution follows deterministic index bounds validation (writable indices before readonly indices).
  - Instruction decompilation reconstructs original `Instruction` structures from `CompiledInstruction` with resolved account references.
  - Signature verification settings are propagated consistently through simulation and validation workflows.

- Parsed instruction caches (`parsed_system_instructions` , `parsed_spl_instructions`) remain valid.
- Cross-program invocation instructions are captured completely when simulation succeeds.

Validator Module (`validator`)
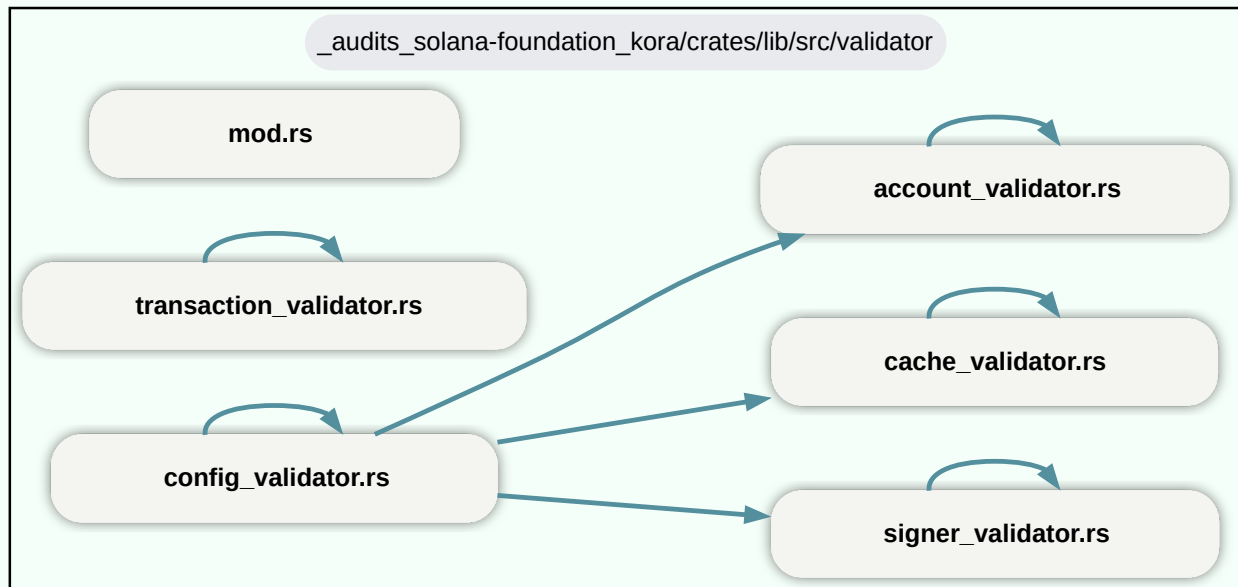
This module has two responsibilities:

- Validating the config files to identify and disallow invalid or inconsistent configurations, as well as to warn users of potentially dangerous or undesirable configurations. This is handled primarily by the `config_validator` submodule, which delegates part of the work to the `account_validator` , `cache_validator` and `signer_validator` submodules.
- Validating transactions for compliance with the restrictions specified in the config file, preventing the submission of transactions that, for example, interact with disallowed programs, tokens or accounts. This is handled by the `transaction_validator` submodule

Data Structures

- The `config_validator` , `cache_validator` and `signer_validator` submodules each have a corresponding `ConfigValidator` , `CacheValidator` and `SignerValidator` struct. However, these are empty structs functioning only as namespaces for the associated validation functions.
- The `account_validator` submodule has an `AccountType` enum with the values `Mint` , `TokenAccount` , `System` and `Program` , which is used to perform validation that an account has data consistent with the appropriate account type. However, the `validate_account` function is on the top-level of the `account_validator` module, with no associated data structure.
- The `transaction_validator` submodule has a corresponding `TransactionValidator` struct, which unlike the structs for the other submodules does contain internal data. This data is loaded from the global config in the `state` module and transactions are checked against it during transaction validation.

`config_validator` Submodule

- The main function in this submodule is `validate_with_result_and_signers` , which validates the global config and creates two lists: one of errors and one of warnings. It prints both lists to the terminal and returns a `Result` : either `Err` with the list of errors or, if that list is empty, `Ok` with the list of warnings.
- `validate_with_result_and_signers` delegates to the other submodules to perform particular steps:
    - `cache_validator` to validate usage limit configuration.
    - `account_validator` to validate accounts in the lists of allowed programs, tokens and SPL paid tokens of the config.
    - `signer_validator` to validate the signers config file.

- `validate_with_result` is a wrapper around `validate_with_result_and_signers` with `signers_config_path` set to `None` . This skips the signers config validation.
- `validate` is a function that performs a simplified validation of the global config, and is only used for testing.
- List of possible errors returned by `validate_with_result_and_signers` :
    - Config is not initialized (in this case, return immediately).
    - Payment address is set but is in an invalid format.
    - List of allowed tokens is empty.
    - A token address in the list of allowed tokens is in an invalid format.
    - A token address in the list of allowed SPL paid tokens is in an invalid format.
    - An address in the list of disallowed accounts is in an invalid format.
    - An extension is in the list of blocked mint extensions or blocked account extensions is in an invalid format.
    - Fees are enabled but neither SPL nor 2022 Token Program is in the list of allowed programs.
    - Fees are enabled but list of allowed SPL paid tokens is empty.
    - There is a token in the list of allowed SPL paid tokens that is not in the list of allowed tokens.
    - Fixed price token address is in an invalid format.
    - Fixed price token address is not in the list of allowed SPL paid tokens.
    - Price margin is negative.
    - Any errors returned by `CacheValidator::validate` , if usage limit is enabled.
    - (only if RPC validation is not skipped) Any element in the list of allowed programs, allowed tokens or allowed SPL paid tokens is in an invalid format for the appropriate account type (as per `account_validator::validate_account`).
    - (only if RPC validation is not skipped) There are missing ATAs for the payment address, or an error occurred when trying to determine if there were missing ATAs.
    - Any errors returned by `SignerValidator::validate_with_result` , if signer pool config is provided.
    - Signer pool config couldn't be loaded from the provided path.

- List of possible warnings returned by `validate_with_result_and_signers` :
    - Rate limit is set to 0.
    - All RPC methods are disabled.
    - Max allowed lamports is set to 0.
    - Max signatures is set to 0.
    - Price source is set to `Mock` .
    - List of allowed programs is empty.
    - System Program is not in the list of allowed programs.
    - Token Program (neither SPL or 2022) is not in the list of allowed programs.
    - List of allowed SPL paid tokens is set to `All` .
    - Fixed price is set to 0 (free).
    - Price margin is greater than 100%.
    - Any warnings returned by `CacheValidator::validate` , if usage limit is enabled.
    - Any warnings returned by `SignerValidator::validate_with_result` , if signer pool config is provided.

`account_validator` Submodule

- The main function in this submodule is `validate_account`, which makes an RPC call to the Solana node in order to validate that an account exists for a given public key. Optionally, if an expected account type is provided, it calls `AccountType::validate_account_type` to check that the account has data consistent with that account type.
- `validate_account_type` validates a given account as the expected account type, returning an error in any of the following cases:
  - A `Mint` or `TokenAccount` fails to be deserialized as that account type.
  - A `Mint` or `TokenAccount` is not owned by a token program.
  - A `System` is not owned by `SYSTEM_PROGRAM_ID`.
  - A `Program` is not executable.
  - A `Mint` or `TokenAccount` is executable.

`cache_validator` Submodule

- The main function in this submodule is `validate`, which validates a given usage-limiting config, including making a Redis connection test for the usage limit cache.
- The function returns two lists: one with errors and one with warnings.
- If usage limiting is disabled, the function skips validation and just returns empty lists.
- List of possible errors returned by `validate`:
  - A cache URL is not provided and fallback is disabled.
  - The cache URL doesn't start with `redis://` or `rediss://`.
  - The Redis connection test fails when fallback is disabled.

- List of possible warnings returned by `validate`:
  - A cache URL is not provided but fallback is enabled (warns that fallback mode will disable usage limits).
  - Fallback is disabled (warns that service will fail if cache becomes unavailable).
  - The Redis connection test fails when fallback is enabled.

`signer_validator` Submodule

- The main function in this submodule is `validate_with_result`, which validates the signer pool config, returning a list of errors and warnings.
- Delegates error checks to methods of the `SignerPoolConfig` (from the `signer::config` module).
- List of possible errors returned by `validate_with_result`:
  - Signer list is empty.
  - Config for an individual signer is invalid.
  - There are duplicate signer names.
  - The pool uses a weighted selection strategy and one of the signers has a weight of 0.

- List of possible warnings returned by `validate_with_result`:
  - The pool uses a weighted selection strategy and one of the signers has no weight specified.
  - The pool doesn't use a weighted selection strategy but one of the signers has a weight specified.

`transaction_validator` Submodule

- Implements the `TransactionValidator` datatype, which is used to validate transactions according to the parameters specified in the global config.
- The `TransactionValidator::new` function constructs a new validator based on the global config parameters and the provided fee payer public key. It also validates that the configured addresses in the allowed and disallowed lists are valid.

- The main method of the `TransactionValidator` is `validate_transaction`, which validates that a given transaction is consistent with the configured parameters. The method returns an error in any of the following cases:
  - The transaction has no instructions or no account keys.
  - The transaction has no signatures or more signatures than the maximum allowed number.
  - An instruction calls a program that is not in the list of allowed programs.
  - The total amount transferred in the transaction is higher than the maximum total.
  - An instruction calls a program that is in the list of disallowed accounts, or an account in the list of accounts read or written by the program is in the list of disallowed accounts.
  - The fee payer is used in the transaction in a way that is disallowed by the fee payer policy.

- The `TransactionValidator` also includes the following additional public methods:
  - `fetch_and_validate_token_mint`: Returns token mint for a given public key, as long as it's among the allowed tokens.
  - `validate_lamport_fee`: Validates that a given fee is below the maximum allowed value.
  - `is_disallowed_account`: Checks that a given account is in the list of disallowed accounts.
  - `validate_token_payment`: Validates for a given transaction that the token transfer to the expected payment destination matches or exceeds the given required lamport amount.

Interaction with Other Modules

- The `TransactionValidator` methods are called by the `transaction::versioned_transaction` module when signing a transaction, and by the `transfer_transaction` RPC method when building a transfer transaction.
- `ConfigValidator::validate_with_result_and_signers` is called by the CLI on the `kora config ...` commands, and also when starting the RPC server on the `kora rpc start` command.
- The `signer_validator` submodule delegates part of the validation to the `signer::config` module.
- The `transaction_validator` submodule calls the `fee` module to calculate the fee payer outflow, in order to validate that it doesn't exceed the maximum amount.

Security Considerations

- If transaction validation is incomplete, it could allow for an undesirable transaction to be submitted to the network.
- If config validation is incomplete, it could leave Kora misconfigured, potentially opening up attack vectors.

Properties and Invariants

- `TransactionValidator` allows a transaction exactly according to the config.
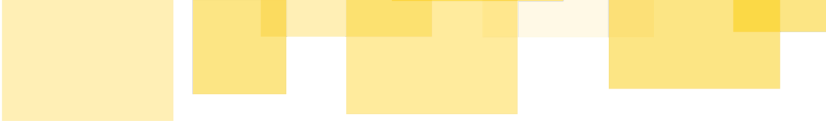
Other Modules

Admin Module (`admin`)

This module is responsible for initializing ATAs for signers so that they are able to receive fee payments in the appropriate tokens. It is called by the Kora CLI on the `kora rpc initialize-atas` RPC command. If a payment address is provided in the config, an ATA is initialized for that address for all SPL payment tokens allowed in the config. If no payment address is specified, it initializes ATAs for all signers in the signer pool.

Cache Module (`cache`)

This module handles caching of account information obtained through Solana RPC calls. When enabled, caching avoids needing to perform expensive RPC calls when the data has already been retrieved previously. The cache is initialized by the CLI and called by the `admin`, `metrics`, `token`, `transaction` and `validation` modules to retrieve information of Solana accounts. The main

function, `get_account`, has a `force_refresh` parameter that when set forces a new value to be retrieved through RPC and cached, rather than returning the previously cached value.

Config Module (`config`)

This module is responsible for loading the Kora config specified in the `kora.toml` file, defining the data structures used to represent this config in memory and accessors for the different configuration parameters. The overall structure will be stored in the `state` module.

Metrics Module (`metrics`)

This module provides monitoring for the Kora RPC server. If configured, this will be added as a middleware layer by the `rpc_server` module during server initialization. Its primary purpose is to track the balance of the signers in order to detect possible attacks leading to draining of funds.

Oracle Module (`oracle`)

This module serves as the interface with a price oracle, used to query the price of tokens in order to calculate fee conversions. Currently only supports Jupiter as a price source.

State Module (`state`)

This module stores the global state of the Kora system, consisting of two parts:

- The global config, containing the configuration parameters specified in the config file.
- The global signer pool, containing the list of signers along with all parameters necessary to implement the signer selection strategy.

The module implements functions to initialize the config and signer pool and access their information, and is called by multiple modules to access Kora's internal state.

Usage Limit Module (`usage_limit`)

This module implements a usage limiter, which if enabled will track usage via a Redis cache and limit the number of transactions that can be submitted by the same sender. The usage limiter is initialized by the `rpc_server` module, and called by the same module before signing a user-submitted transaction to check if the limit has been reached (and increment the user's usage tracking counter if not). Currently, the only form of usage limit supported is one where there is a fixed maximum number of transactions per user. Once a wallet reaches this maximum, it cannot submit any more transactions.

Findings

This section contains all issues identified during the audit that could lead to unintended behavior, security vulnerabilities, or failure to enforce the protocol's intended logic. Each issue is documented with a description, potential impact, and recommended remediation steps.

[A01] Early return in `initialize_atas_with_chunk_size()` when an address has no ATAs to create

`Severity: Medium` `Difficulty: Low` `Recommended Action: Fix Code` `Addressed by client`

In the function `initialize_atas_with_chunk_size()` in `admin/token_util.rs` , inside the `for` loop that iterates over the addresses for which to create ATAs, if the list of ATAs to create for an address is empty, the function will return early without creating ATAs for the remaining addresses:

 runtimeverification/_audits_solana-foundation_kora/crates/lib/src/admin/token_util.rs
Line 94 to 97 in c2843ac

```
94          if atas_to_create.is_empty() {
95              println!("✓ All required ATAs already exist for address: {address}");
96              return Ok(());
97          }
```

Since the function returns successfully, execution will proceed as if ATAs had been created for all addresses, even though this is not the case.

Recommendation

Instead of returning, continue execution with the next iteration of the loop, either by using `continue` or by putting the call to `create_atas_for_signer` in an `else` block.

Status

Commit dd3458c updates `initialize_atas_with_chunk_size` to use `continue` instead of returning when the list of ATAs to create for an address is empty. Additionally, new messages are added to `create_atas_for_signer` when an ATA creation request fails, clarifying which ATAs have been successfully created before the failure.

[A02] `process_token_transfer` return values have conflicting meanings

Severity: Low   Difficulty: Low   Recommended Action: Fix Design   Addressed by client

`process_token_transfer` in `token/token.rs` currently returns `Ok(false)` for a given transaction in two different cases:

- When no payment is found to the expected destination account with a value greater than or equal to the required lamport amount.
- When a payment is found to the expected destination account using an unsupported payment token.

This leads to two issues:

- Since it also returns `Ok(true)` if it finds a valid payment of at least the required lamport amount, if the transaction includes both a valid payment and a payment using an unsupported payment token (both to the expected destination account), the return value will depend entirely on which one happens first.
- If the function returns `Ok(false)`, it's unclear whether it is because the transaction lacks the required payment amount or because there was a transfer using an unsupported payment token. This means, for example, that the error message in `validate_token_payment` might be inaccurate if it's the latter case:

> runtimeverification/_audits_solana-foundation_kora/crates/lib/src/validator/transaction_validator.rs
>
> Line 315 to 317 in c2843ac
>
> ```
> 315        Err(KoraError::InvalidTransaction(format!(
> 316            "Insufficient token payment. Required {required_lamports} lamports"
> 317        )))
> ```

Recommendation

In conversation with the Kora team, it was established that a token transfer to the expected destination account using an unsupported payment token not cause an early return, and should instead just be ignored so that we can continue searching for a valid payment. Additionally, the following modifications should be considered:

- Allow payments to be made in multiple transfers. In other words, if there are multiple token transfers of supported payment tokens to the expected destination account, these should be summed to count towards the required lamport amount.
- Split the check for `validate_token2022_extensions_for_payment` into its own function, and add this to `validate_transaction` in `validator/transaction_validator.rs`.
- Add an additional function to `validate_transaction` that checks if all token transfers are for tokens in the list of allowed tokens.

Status

PR #240 renames `process_token_transfer` to `verify_token_payment` and updates the function to ignore transfers using unsupported tokens rather than returning early. The function has also been updated to sum the total value of all payments in supported tokens, rather than returning the first valid one.

[A03] Payment instruction detection uses first-match logic instead of aggregation

`Severity: Low` `Difficulty: Low` `Recommended Action: Fix Design` `Addressed by client`

The `has_payment_instruction` function in

> 🔗 runtimeverification/_audits_solana-foundation_kora/crates/lib/src/fee/fee.rs
> Line 173 to 193 in c2843ac

```
173        for instruction in resolved_transaction
174            .get_or_parse_spl_instructions()?
175            .get(&ParsedSPLInstructionType::SplTokenTransfer)
176            .unwrap_or(&vec![])
177        {
178            if let ParsedSPLInstructionData::SplTokenTransfer { destination_address, .. } =
179                instruction
180            {
181                if Self::get_payment_instruction_info(
182                    rpc_client,
183                    destination_address,
184                    &payment_destination,
185                    false, // Don't skip missing accounts for has_payment_instruction
186                )
187                .await?
188                .is_some()
189                {
190                    return Ok(0);
191                }
192            }
193        }
```

implements a flawed first-match detection logic that prevents valid multi-transfer payment scenarios from being properly recognized.

The function returns `Ok(0)` immediately upon detecting any transfer to the payment destination, without validating the transfer amount or aggregating multiple transfers to the same destination. This causes valid multi-transfer payments to be rejected when individual transfers are insufficient, but their collective sum would meet the payment requirement.

This prevents users from executing legitimate payments that use multiple transfers from different token accounts, resulting in transaction failures for valid payment scenarios that the protocol should support.

Recommendation

- Modify the payment detection logic to aggregate all transfers to the payment destination.
- Validate the total aggregated amount against the payment requirement.
- Add test cases for multi-transfer payment scenarios.
- Update `process_token_transfer` logic to support aggregation as described in [A02] `process_token_transfer` return values have conflicting meanings.

Status

As of commit 6b1dd87, `has_payment_instruction` has been merged with `calculate_transfer_fees` into a single `analyze_payment_instructions` function. The function only checks that a payment instruction exists, not that the total of all payment instructions is sufficient, but this is checked instead in `verify_token_payment` (see [A02] `process_token_transfer` return values have conflicting meanings).

[A04] Integer Overflow in Fee Aggregation

`Severity: High`   `Difficulty: High`   `Recommended Action: Fix Code`   `Addressed by client`

The final fee summation in `estimate_transaction_fee`

---
runtimeverification/_audits_solana-foundation_kora/crates/lib/src/fee/fee.rs

Line 305 to 310 in c2843ac

```
305        let total_fee_lamports = base_fee
306            + account_creation_fee
307            + kora_signature_fee
308            + fee_payer_outflow
309            + fee_for_payment_instruction
310            + transfer_fee_config_amount;
```
---

uses regular addition while individual fee components use saturating arithmetic, creating a critical integer overflow vulnerability that could enable fund drainage attacks.

The fee components use `saturating_add()` for protection, but the final aggregation uses standard addition that can overflow. When fee components sum beyond `u64::MAX`, the result wraps around to a small value instead of saturating, allowing attackers to process expensive transactions with minimal fees. Although it's unlikely that the total fee could ever actually reach `u64::MAX`, it's a good idea to safeguard against the possibility of exploits that could artificially inflate the fee calculation with the intention of triggering an overflow.

Recommendation

- Replace regular addition in the final fee summation with saturating arithmetic to ensure totals near or above `u64::MAX` safely saturate rather than wrap around, maintaining consistency with the component-level use of `saturating_add()`.
- Add a test covering overflow across all fee components and validate fee monotonicity so that increasing any element cannot reduce the total.
- Additionally, consider replacing the use of `saturating_add` with `checked_add`, allowing the code to detect if an overflow has occurred and return an error instead of silently defaulting to the maximum value. If an overflow were to occur, this would allow logging and debugging the circumstances that caused it.

Status

Commit 01f69c2 updates the final sum of the fee in `estimate_transaction_fee` to use `checked_add`. PR #248 also updates `calculate_fee_payer_outflow` to use `checked_add`.

[A05] `RustSec` dependency vulnerabilities

Severity: Low   Difficulty: Medium   Addressed by client

Three RustSec vulnerabilities were identified: curve25519-dalek timing side-channel (RUSTSEC-2024-0344), ed25519-dalek double public key signing oracle (RUSTSEC-2022-0093), and Borsh unsoundness (RUSTSEC-2023-0033). All are transitive dependencies via the Solana SDK, requiring coordinated upgrades.

Recommendation

Plan dependency upgrades to curve25519-dalek >= 4.1.3, ed25519-dalek >= 2.0, and borsh >=1.0, coordinated with Solana SDK updates.

Status

PR #242 upgrades the dependencies to the latest version and removes outdated dependencies.

[A06] Double counting of ATA rent fees during fee estimation

`Severity: Medium` `Difficulty: Low` `Recommended Action: Fix Code` `Addressed by client`

The `estimate_transaction_fee` function in `fee.rs` calls `get_associated_token_account_creation_fees` to calculate fees associated with paying the rent of any ATAs created by the transaction:

> ○ runtimeverification/_audits_solana-foundation_kora/crates/lib/src/fee/fee.rs
>
> Line 276 to 280 in c2843ac
>
> ```
> 276        // Get account creation fees (for ATA creation)
> 277        let account_creation_fee =
> 278            FeeConfigUtil::get_associated_token_account_creation_fees(rpc_client, transaction)
> 279                .await
> 280                .map_err(|e| KoraError::RpcError(e.to_string())))?;
> ```

However, it also later calls `calculate_fee_payer_outflow` to account for general calls to the System Program that need to be paid by the fee payer, which includes account creation:

> ○ runtimeverification/_audits_solana-foundation_kora/crates/lib/src/fee/fee.rs
>
> Line 290 to 292 in c2843ac
>
> ```
> 290        // Calculate fee payer outflow if fee payer is provided, to better estimate the potential fee
> 291        let fee_payer_outflow =
> 292            FeeConfigUtil::calculate_fee_payer_outflow(fee_payer, transaction).await?;
> ```

Since ATA creation involves an account creation call to the System Program, ATA creation rent fees will be double counted in the fee calculation.

Recommendation

Modify the code so that ATA rent fees are accounted for in only one place.

Status

Commit 01f69c2 removes the `get_associated_token_account_creation_fees` function, so that account creation fees are added only during the fee payer outflow calculation.

[A07] Unsafe `unwrap()` calls in production code

`Severity: Medium` `Difficulty: Low` `Recommended Action: Fix Code` `Addressed by client`

Several places in the production code call `unwrap()` on an `Option` type. This will cause a panic if the value is `None`. Examples of functions that use `unwrap()` include:

- `get_required_lamports`
- `get_or_parse_system_instructions`
- `get_or_parse_spl_instructions`
- `validate_token2022_extensions_for_payment`
- `encode_versioned_transaction`
- `validate_with_result_and_signers`

Recommendation

Replace the `unwrap()` calls with proper error handling.

Status

Commit 01f69c2 removes calls to `unwrap()` and replaces them with appropriate error handling. The only remaining uses of `unwrap()` are in tests, in the `openapi` module, and one instance in `transaction/instruction_utils.rs` that is safe because the program logic guarantees the value will not be `None` (and there is a comment in the code documenting this). PR #248 also removes uses of `expect()`, which can likewise panic.

[A08] Fee payer policy is maximally permissive by default

Severity: Medium Difficulty: Medium Recommended Action: Fix Code Addressed by client

The default implementation of `FeePayerPolicy` sets all options to `true`, making it maximally permissive:

runtimeverification/_audits_solana-foundation_kora/crates/lib/src/config.rs
Line 148 to 160 in c2843ac

```
148  impl Default for FeePayerPolicy {
149      fn default() -> Self {
150          Self {
151              allow_sol_transfers: true,
152              allow_spl_transfers: true,
153              allow_token2022_transfers: true,
154              allow_assign: true,
155              allow_burn: true,
156              allow_close_account: true,
157              allow_approve: true,
158          }
159      }
160  }
```

This can be a security risk if a Kora operator accidentally forgets to add the fee payer section to the config file, as it would expose them to users using the fee payer account in potentially undesirable ways.

Recommendation

Change the default so that all options are set to `false` if there is no fee payer policy specified in the config file.

Status

PR #240 sets the default of all fee policy configurations to `false`.

[A09] Fixed pricing model does not require user to pay back fee payer outflow

`Severity: High` `Difficulty: Medium` `Recommended Action: Fix Design` `Addressed by client`

When using the `fixed` pricing model, the fee payer outflow is not accounted for in the fee calculation. This means that a user can submit a transaction that forces Kora to spend up to the maximum amount of allowed lamports without having to pay it back. This is especially dangerous if the fee payer policy allows for SOL transfers, since it would allow users to drain funds from Kora to their own account.

Recommendation

Make sure this is clearly documented regarding the `fixed` pricing model, and add warnings (or possibly even errors) during config validation if the `fixed` pricing model is used together with dangerous options, such as `allow_sol_transfers` or with no authentication.

Additionally, consider generalizing the price model to allow certain fee components, like fee payer outflow, to be included even when using the `fixed` pricing model.

Status

- PR #240 adds warnings to the config validator if the `fixed` pricing model is used with no authentication enabled or SOL/SPL/Token2022 transfers allowed in the fee payer policy.
- PRs #243 adds warning to operators in the documentation that `fixed` and `free` pricing models do not include fee payer outflow, and therefore are under risk of the fee payer account being drained if the fee payer policy allows transfers.
- PR #244 adds a `strict` flag to the `fixed` pricing model. If this flag is set, the new function `validate_strict_pricing_with_fee` rejects transactions for which the estimated amount in fees that Kora would need to pay exceeds the price.
- PR #246 adds additional documentation on the risks of enabling transfers and other privileged operations.

[A10] SPL token transfers are not accounted for in fee payer outflow

Severity: High   Difficulty: Medium   Recommended Action: Fix Design   Addressed by client

`calculate_fee_payer_outflow` in `fee/fee.rs` adds up all SOL transfers in order to charge the user for those as part of the fee (as long as the pricing model is `margin`, see [A09] Fixed pricing model does not require user to pay back fee payer outflow). However, it doesn't account for SPL token transfers. Therefore, if the fee payer holds any SPL tokens and the fee payer policy allows token transfers, a user can submit a transaction that transfers tokens out of the fee payer's ATA without having to pay them back.

Note that if the payment address is equal to the fee payer, this includes the tokens transferred by the user to pay Kora's fees. In other words, a user would be able to pay the fees and then immediately transfer the tokens back in the same transaction.

Recommendation

Include token transfers in the fee payer outflow calculation, charging the user for the amount transferred.

Status

Commit 5a61ea1 adds the function `calculate_spl_transfers_value_in_lamports` to `token/token.rs` and calls it in `calculate_fee_payer_outflow` to calculate the value in SOL of the SPL outflow.

[A11] Access control missing for unsupported SPL instructions

Severity: Medium   Difficulty: Medium   Recommended Action: Fix Code   Addressed by client

The `validate_fee_payer_usage` function enforces policy only on SPL `Transfer`, `Approve`, `Burn`, and `CloseAccount` operations. Other SPL Token instructions such as `MintTo`, `FreezeAccount`, `Revoke`, `SetAuthority`, and `ThawAccount` are not checked and will bypass fee payer policy restrictions if included in a transaction with the SPL Token program in `allowed_programs`. Although it's considered unlikely that the fee payer will have the authority to perform these kinds of instructions, it could lead to significant security problems if they did.

Recommendation

- Extend `validate_fee_payer_usage` to cover all SPL Token instructions that impact fee payer usage or token state.
- Add validation for MintTo, FreezeAccount, Revoke, SetAuthority, ThawAccount, and checked versions.
- Use `ParsedSPLInstructionType` or extend it to parse and enforce these additional instruction types.

Status

Commit 8a46b97 adds the other SPL token instructions to `parse_token_instructions`, both for standard SPL tokens and Token2022 tokens. Macros `validate_spl` and `validate_spl_multisig` have been added to check if the instructions conform to the fee payer policy, and are called in `validate_fee_payer_usage`.

[A12] Non-constant-time comparison of HMAC signatures

Severity: High   Difficulty: Low   Recommended Action: Fix Code   Addressed by client

The HMAC authentication

runtimeverification/_audits_solana-foundation_kora/crates/lib/src/rpc_server/auth.rs

Line 187 to 189 in c2843ac

```
187                if signature != expected_signature {
188                    return Ok(unauthorized_response);
189                }
```

compares the provided signature to the expected signature using standard string equality. This comparison isn't guaranteed to execute in constant time, which can introduce a timing side-channel. Requests with more matching prefix bytes take slightly longer. Over many samples, an attacker can average timings to guess the next byte.

Recommendation

Replace the non-constant-time string comparison with a constant-time verification method.

Status

Commit 4319320 replaces the string comparison with the `hmac` crate's `verify_slice` function, which performs constant-time comparison.

[A13] API key comparison vulnerable to timing attacks

Severity: High   Difficulty: Low   Recommended Action: Fix Code   Addressed by client

The API key authentication middleware in

runtimeverification/_audits_solana-foundation_kora/crates/lib/src/rpc_server/auth.rs
Line 68 to 70 in c2843ac

```
68              if let Some(provided_key) = req.headers().get(X_API_KEY) {
69                  if provided_key.to_str().unwrap_or("") == api_key {
70                      return inner.call(req).await;
```

compares the provided API key to the expected key using Rust's standard equality operator (==), which is not guaranteed to execute in constant time. String comparison typically short-circuits on the first differing byte, creating a timing side-channel that could allow attackers to infer the correct API key byte-by-byte.

Recommendation

Use constant-time comparison for API key validation.

Status

PR #240 updates the authentication to use the `ct_eq` constant-time comparison method from the `subtle` crate.

[A14] Panic on invalid UTF-8 in HMAC authentication

`Severity: Medium` `Difficulty: Low` `Recommended Action: Fix Code` `Addressed by client`

The HMAC authentication

> ○ runtimeverification/_audits_solana-foundation_kora/crates/lib/src/rpc_server/auth.rs
>
> Line 174 in c2843ac
>
> ```
> 174              let message = format!("{}{}", timestamp, std::str::from_utf8(&body_bytes).unwrap());
> ```

uses `std::str::from_utf8(&body_bytes).unwrap()` when constructing the HMAC message. If an attacker sends a request body containing invalid UTF-8 sequences and includes headers `X_TIMESTAMP` (parsable as a recent UNIX timestamp) and `X_HMAC_SIGNATURE`, the `unwrap()` will panic.

Recommendation

- Replace the `unwrap()` on UTF-8 conversion with safe error handling.
- Use `std::str::from_utf8(&body_bytes).ok()` to detect and reject invalid UTF-8.

Status

PR #240 updates the HMAC signature verification to gracefully handle errors in the decoding, returning the `UNAUTHORIZED` status code instead of panicking.

[A15] DoS via Unbounded Request Body Buffering

`Severity: Medium`   `Difficulty: Low`   `Recommended Action: Fix Code`   `Addressed by client`

The middleware in

runtimeverification/_audits_solana-foundation_kora/crates/lib/src/rpc_server/middleware_utils.rs

Line 12 to 20 in c2843ac

```
12      let (parts, body) = request.into_parts();
13      let body_bytes = body
14          .try_fold(Vec::new(), |mut acc, chunk| async move {
15              acc.extend_from_slice(&chunk);
16              Ok(acc)
17          })
18          .await
19          .unwrap_or_default();
20      (parts, body_bytes)
```

has no size limit on request body buffering, affecting all RPC requests, including API key authentication, HMAC authentication, and metrics endpoints. This creates a potential denial of service vulnerability where attackers can exhaust server memory by sending extremely large request bodies.

Recommendation

Add request body size limits to the middleware.

Status

Commit 8d4f172 introduces a max request body size, with a default of 2 MB, which can be configured to a custom value in the config file.

[A16] Redis Password Exposure in Logs

`Severity: Medium` `Difficulty: Low` `Recommended Action: Fix Code` `Addressed by client`

The Redis connection initialization logs the full Redis URL

> runtimeverification/_audits_solana-foundation_kora/crates/lib/src/cache.rs
>
> Line 54 in c2843ac
>
> 54   `log::info!("Cache initialized successfully with Redis at {redis_url}");`

including credentials, potentially exposing passwords in log files. This occurs when the Redis URL contains authentication information in the format `redis://user:password@host:port` .

Recommendation

Sanitize Redis URL before logging.

Status

Commit ef8fc5a introduces a `sanitize_error!` macro that redacts certain patterns in error messages, in particular those corresponding to URLs with credentials and hex strings that can potentially be private keys. The macro is used for errors in high-risk modules that deal with private keys and authentication. The sanitization is performed as long as the `unsafe-debug` option is not turned on in the `Cargo.toml` file. In addition, some error messages (including the one above) were modified to remove potentially-sensitive information entirely, and the `Debug` trait was removed from data structures that might contain sensitive information.

[A17] Debug Trait Exposes Sensitive Data in Signer Structs

`Severity: Medium` `Difficulty: Low` `Recommended Action: Fix Code` `Addressed by client`

Multiple signer structs derive the `Debug` trait, which exposes sensitive fields when logging or printing debug information. This risks leaking API secrets, private keys, and key pairs in logs or error messages.

> ⬤ runtimeverification/_audits_solana-foundation_kora/crates/lib/src/signer/privy/types.rs
>
> Line 5 to 8 in c2843ac
>
> ```
> 5  #[derive(Clone, Debug)]
> 6  pub struct PrivySigner {
> 7      pub app_id: String,
> 8      pub app_secret: String,
> ```

> ⬤ runtimeverification/_audits_solana-foundation_kora/crates/lib/src/signer/turnkey/types.rs
>
> Line 4 to 9 in c2843ac
>
> ```
> 4  #[derive(Clone, Debug)]
> 5  pub struct TurnkeySigner {
> 6      pub organization_id: String,
> 7      pub private_key_id: String,
> 8      pub api_public_key: String,
> 9      pub api_private_key: String,
> ```

> ⬤ runtimeverification/_audits_solana-foundation_kora/crates/lib/src/signer/memory_signer/solana_signer.rs
>
> Line 12 to 14 in c2843ac
>
> ```
> 12  #[derive(Debug)]
> 13  pub struct SolanaMemorySigner {
> 14      keypair: Keypair,
> ```

Recommendation

Remove `Debug` trait from sensitive signer structs or implement a custom `Debug` that redacts sensitive fields.

Status

As of PR #233, the `signer` module now uses the new `solana-signers` library, where the signer data structures no longer derive the `Debug` trait.

[A18] System Instructions Bypass in Fee Payer Policy

`Severity: Medium` `Difficulty: Medium` `Recommended Action: Fix Design` `Addressed by client`

The `validate_fee_payer_usage` function in `transaction_validator.rs`

> ○ runtimeverification/_audits_solana-foundation_kora/crates/lib/src/validator/transaction_validator.rs
> Line 188 to 202 in c2843ac
>
> ```
> 188        for instruction in
> 189            system_instructions.get(&ParsedSystemInstructionType::SystemTransfer).unwrap_or(&vec![])
> 190        {
> 191            if let ParsedSystemInstructionData::SystemTransfer { sender, .. } = instruction {
> 192                check_if_allowed(sender, self.fee_payer_policy.allow_sol_transfers)?;
> 193            }
> 194        }
> 195
> 196        for instruction in
> 197            system_instructions.get(&ParsedSystemInstructionType::SystemAssign).unwrap_or(&vec![])
> 198        {
> 199            if let ParsedSystemInstructionData::SystemAssign { authority } = instruction {
> 200                check_if_allowed(authority, self.fee_payer_policy.allow_assign)?;
> 201            }
> 202        }
> ```

parses `SystemCreateAccount` and `SystemWithdrawNonceAccount` instructions but does not validate them against the fee payer policy. This allows attackers to include these instructions that use the fee payer as a funding source or authority, potentially draining SOL balances.

The validation function implements policy checks for only two system instruction types: `SystemTransfer` and `SystemAssign`. The `SystemCreateAccount` and `SystemWithdrawNonceAccount` instructions are parsed in `instruction_util.rs` and passed to the validation function, but no corresponding validation loops exist to enforce fee payer policy restrictions on these instruction types.

Although the fee payer outflow calculation accounts for these instructions and charges them back to the user, they still might not be desirable, and an operator may want to block them entirely through the fee payer policy. This is especially the case as the fee payer outflow is not applied outside of the `margin` pricing model (see [A09] Fixed pricing model does not require user to pay back fee payer outflow).

Recommendation

Add validation for both system instructions in the fee payer policy.

Status

Commit 8a46b97 adds options to the fee payer policy for all system instructions and wraps their validation in the `validate_system!` macro.

[A19] Jupiter price oracle lacks validation

Severity: High   Difficulty: Medium   Recommended Action: Fix Code   Addressed by client

The Jupiter API price oracle integration in

runtimeverification/_audits_solana-foundation_kora/crates/lib/src/oracle/jupiter.rs
Line 34 to 45 in c2843ac

```
34   struct JupiterPriceData {
35       #[serde(rename = "usdPrice")]
36       usd_price: f64,
37       #[serde(rename = "blockId")]
38       #[allow(dead_code)]
39       block_id: u64,
40       #[allow(dead_code)]
41       decimals: u8,
42       #[serde(rename = "priceChange24h")]
43       #[allow(dead_code)]
44       price_change_24h: Option<f64>,
45   }
```

fetches token prices from Jupiter's API but performs no validation on the returned data.

runtimeverification/_audits_solana-foundation_kora/crates/lib/src/oracle/jupiter.rs
Line 154 to 156 in c2843ac

```
154          let price = price_data.usd_price / sol_price.usd_price;
155
156          Ok(TokenPrice { price, confidence: 0.95, source: PriceSource::Jupiter })
```

The implementation accepts any price without checking for staleness (age of price data), extreme values (bounds checking), or confidence thresholds. This creates potential attack vectors through price manipulation and exposes operators to operational risks from stale or unreliable price feeds.

Recommendation

Add comprehensive price validation to the Jupiter oracle.

Status

PR #240 adds the `validate_price_data` function, which performs a sanity check to ensure that the oracle prices are within reasonable bounds. Jupiter's API has no easy way to check for staleness of the data, so node operators are required to accept the associated risks of using Jupiter.

[A20] Unchecked array indexing in instruction account access

Severity: Medium   Difficulty: Low   Recommended Action: Fix Code   Addressed by client

Multiple locations in the codebase perform direct array indexing into `instruction.accounts` without first validating that the array has sufficient elements. This can cause panics if a malformed or unexpected instruction is processed, leading to denial of service or service crashes.

 runtimeverification/_audits_solana-foundation_kora/crates/lib/src/fee/fee.rs
Line 73 to 81 in c2843ac

```
73          for instruction in &transaction.all_instructions {
74              // Skip if not an ATA program instruction
75              if instruction.program_id != spl_associated_token_account::id() {
76                  continue;
77              }
78
79              let ata = instruction.accounts[1].pubkey;
80              let owner = instruction.accounts[2].pubkey;
81              let mint = instruction.accounts[3].pubkey;
```

instruction_util has similar patterns throughout the file.

Recommendation

Add bounds checking before array indexing.

Status

Commit 5da4f28 adds the `validate_number_accounts!` macro, which can be used to check whether the length of the account array for an instruction matches the required number of accounts. This macro is used in `transaction/instruction_util.rs` before every array access into `instruction.accounts`.

[A21] Permanent Delegate extension can be used to undo payments to Kora

`Severity: High` `Difficulty: High` `Recommended Action: Document Prominently` `Addressed by client`

The Permanent Delegate Token 2022 extension allows a token to define a delegate with global authority to transfer, burn or freeze tokens from any accounts associated with the token. Therefore, interacting with such a token would carry significant risk to a Kora node:
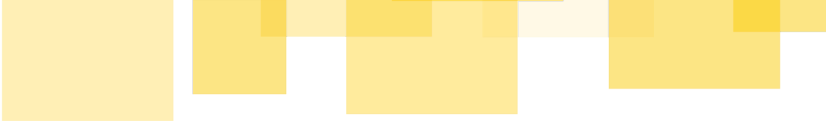
- If a node were to allow fee payments in such a token, the delegate would be able to undo the payment by sending the tokens back.
- Even if the token is not used as a payment token, if the fee payer outflow accounts for token transfers (as per [A10] SPL token transfers are not accounted for in fee payer outflow) it could be transferred to Kora to reduce the outflow, only for the delegate to transfer the tokens back later.

Recommendation

It is important for Kora operators to be aware of the risks of allowing tokens with the Permanent Delegate extension to be used. They should not be used for payment tokens, and even just allowing them as an allowed token can be dangerous. These dangers should be explicitly documented in the documentation for operators. Additionally, it would be advisable to emit a warning during config validation if the Permanent Delegate is configured as an allowed extension.

Status

- PR #240 adds a warning to `validate_with_result_and_signers` in `validator/config_validator.rs` that warns about the risks of the Permanent Delegate extension if it's not blocked.
- PR #243 adds a "Security Consideration" subsection under "Token-2022 Extension Blocking" in `CONFIGURATION.md` , warning the operator about the extension.
- PR #244 adds a `check_token_mint_extensions` function to `validator/config_validator.rs` , which checks if any allowed tokens include risky extensions (including Permanent Delegate) and reports those as warnings to the user.

[A22] `sign_transaction` and `sign_and_send_transaction` allow transactions to be submitted for free independently of the price model

`Severity: High`   `Difficulty: Medium`   `Recommended Action: Fix Design`   `Addressed by client`

The `sign_transaction` method implemented in `rpc_server/method/sign_transaction.rs` allows users to request transactions to be signed by Kora without paying any fees, in contrast to `sign_transaction_if_paid`. This is redundant since if it's desired that users can submit transactions for free, the operator can set the price model to "free" in the config. Additionally, it opens the possibility for operators that do intend to charge fees to accidentally leave this method enabled, thus allowing users to bypass the fees and submit transactions for free.

Additionally, the `sign_and_send_transaction` method implemented in `rpc_server/method/sign_transaction_and_submit.rs` doesn't check that fees are being paid either, so it also can't be used together with a non-free price model.

Recommendation

- Remove the `sign_transaction` method, since its only legitimate use case can be covered by setting the price model to "free."
- Turn `sign_and_send_transaction` into `sign_and_send_transaction_if_paid`, adding validation that the payment is included in the submitted transaction.

Status

PR #241 removes the `sign_transaction_if_paid` method and instead modifies the `sign_transaction` function in `transaction/versioned_transaction.rs` (which is used by both the `sign_transaction` and `sign_and_send_transaction` RPC methods) to require payment.

[A23] Only first transfer fee is added to the Kora fee

`Severity: High` `Difficulty: Low` `Recommended Action: Fix Code` `Addressed by client`

The function `calculate_transfer_fees` in `fee/fee.rs` returns as soon as a transfer is found for a token with a transfer fee configured:

> ⓞ runtimeverification/_audits_solana-foundation_kora/crates/lib/src/fee/fee.rs
>
> Line 246 to 257 in c2843ac

```
246                    // For Token2022, check for transfer fees
247                    if let Some(token2022_mint) =
248                        mint_state.as_any().downcast_ref::<Token2022Mint>()
249                    {
250                        let current_epoch = rpc_client.get_epoch_info().await?.epoch;
251
252                        if let Some(fee_amount) =
253                            token2022_mint.calculate_transfer_fee(*amount, current_epoch)
254                        {
255                            return Ok(fee_amount);
256                        }
257                    }
```

However, a transaction can have multiple such transfer instructions. In that case, only the first transfer fee would be added to the total fee charged to the user. This would allow a user to submit a transaction with multiple transfers and only pay the fees for the first one, forcing Kora to pay for all of the other transfers.

Recommendation

Fix the code so that, instead of returning early, the function adds up the fees for all transfer instructions and returns the total at the end.

Status

Commit 6b1dd87 updates the function to accumulate all Token2022 transfer fees and return the total fee, instead of returning only the first one.

Informative Findings

This section includes observations that are not directly exploitable but highlight areas for improvement in code clarity, maintainability, or best practices. While not critical, addressing these can strengthen the system's overall robustness.

[B01] Error message in `validate_account_type` only considers one of the possible error cases

Severity: Informative Recommended Action: Fix Code Addressed by client

The function `validate_account_type` in `validator/account_validator.rs` returns an error if an account's `executable` flag has a different value than it should based on the expected account type:

 runtimeverification/_audits_solana-foundation_kora/crates/lib/src/validator/account_validator.rs

Line 92 to 98 in c2843ac

```
92          if let Some(should_be_executable) = should_be_executable {
93              if account.executable != should_be_executable {
94                  return Err(KoraError::InternalServerError(format!(
95                      "Account {account_pubkey} is not executable, cannot be a Program"
96                  )));
97              }
98          }
```
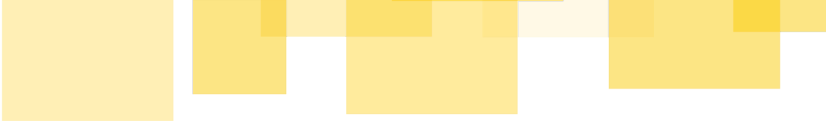
However, the error message only refers to the case when a `Program`, which should be executable, is not. But this error can also be returned when a `Mint` or `TokenAccount` is executable, when it shouldn't. This makes this error message misleading in these cases.

Recommendation

Rewrite the error message to be more generic, or consider both cases separately and return a different error message for each.

Status

PR #240 updates the error message to be more accurate.

[B02] `CacheValidator::validate` has `Result` return type but never returns an `Err` value

`Severity: Informative` `Recommended Action: Fix Code` `Addressed by client`

The `validate` function in `validator/cache_validator.rs` has return type `Result<(Vec<String>, Vec<String>), String>` , but it only ever returns an `Ok` value. Any errors or warnings are just added to the two lists wrapped in the `Ok` value.

Recommendation

Change the return type to just `(Vec<String>, Vec<String>)` instead of wrapping the value into a `Result` .

Status

PR #240 modifies `validate` to no longer wrap the return value in a `Result` .

[B03] If-then-else for `Option` can be replaced by `match` expression

`Severity: Informative` `Recommended Action: Fix Code` `Addressed by client`

The `validate` function in `validator/cache_validator.rs` uses an if-then-else pattern to deconstruct `usage_config.cache_url`, which has type `Option<String>`:

```rust
if usage_config.cache_url.is_none() {
    ...
} else if let Some(cache_url) = &usage_config.cache_url {
    ...
}
```

Instead, it can be replaced by a `match` expression, making it clearer and giving less room for errors:

```rust
match usage_config.cache_url {
    None =>
        ...
    Some(ref cache_url) =>
        ...
}
```

Status

PR #240 rewrites the if-then-else into a `match` expression.

[B04] Potentially misleading comment about signer selection strategy

`Severity: Informative`   `Recommended Action: Document Prominently`   `Addressed by client`

When the `get_request_signer_with_signer_key` in `state.rs` calls `SignerPool::get_next_signer`, the comment says that the default behavior is to select the next signer from round-robin:

○ runtimeverification/_audits_solana-foundation_kora/crates/lib/src/state.rs

Line 32 to 35 in c2843ac

```
32    // Default behavior: use next signer from round-robin
33    let signer_meta = pool.get_next_signer().map_err(|e| {
34        KoraError::InternalServerError(format!("Failed to get signer from pool: {e}"))
35    })?;
```

However, before it defaults to round-robin, it will use the strategy specified in the signer pool config if there is one. Only if no strategy is specified it will default to round-robin.

Recommendation

Rewrite the comment to clarify that it will use the configured strategy, or default to round-robin if none is specified.

Status

PR #240 updates the comment appropriately.

[B05] Spurious fields in `SignTransactionResponse` and `SignTransactionIfPaidResponse`

`Severity: Informative`   `Recommended Action: Fix Code`   `Addressed by client`

In `rpc_server/method/sign_transaction.rs`, the `sign_transaction` function returns a signature in the `SignTransactionResponse` data structure:

> runtimeverification/_audits_solana-foundation_kora/crates/lib/src/rpc_server/method/sign_transaction.rs
>
> Line 55 to 59 in c2843ac
>
> ```
> 55    Ok(SignTransactionResponse {
> 56        signature: transaction.signatures[0].to_string(),
> 57        signed_transaction: encoded,
> 58        signer_pubkey: signer.solana_pubkey().to_string(),
> 59    })
> ```

However, this signature is being taken from `transaction` rather than `signed_transaction`, and assumes the fee payer signature is necessarily in index 0, while the `sign_transaction` function finds the position of the fee payer in the account list and places the signature in the corresponding index:

> runtimeverification/_audits_solana-foundation_kora/crates/lib/src/transaction/versioned_transaction.rs
>
> Line 257 to 259 in c2843ac
>
> ```
> 257        // Find the fee payer position - don't assume it's at position 0
> 258        let fee_payer_position = self.find_signer_position(&signer.solana_pubkey())?;
> 259        transaction.signatures[fee_payer_position] = signature;
> ```

Additionally, in `rpc_server/method/sign_transaction_if_paid.rs`, the `sign_transaction_if_paid` function returns a `SignTransactionIfPaidResponse`, but the `transaction` field is assigned in a way that just repeats the contents of the `signed_transaction` field:

> runtimeverification/_audits_solana-foundation_kora/crates/lib/src/rpc_server/method/sign_transaction_if_paid.rs
>
> Line 55 to 59 in c2843ac
>
> ```
> 55    Ok(SignTransactionIfPaidResponse {
> 56        transaction: TransactionUtil::encode_versioned_transaction(&transaction),
> 57        signed_transaction,
> 58        signer_pubkey: signer.solana_pubkey().to_string(),
> 59    })
> ```

Both of these fields (`signature` in `SignTransactionResponse` and `transaction` in `SignTransactionIfPaidResponse`) are spurious, and the returned values are not being used. Therefore, they can be removed from their respective structs.

Status

PR #240 removes the `transaction` field from `SignTransactionIfPaidResponse` (also note that the `sign_transaction_if_paid` method was later removed in PR #241). The `signature` field has also been removed from `SignTransactionResponse`.

[B06] Price model documentation in `FEES.md`

`Severity: Informative` `Recommended Action: Document Prominently` `Addressed by client`

It's not clear from the `FEES.md` document that the Fee Calculation Formula only applies when using the `margin` price model (see Price Configuration in `CONFIGURATION.md`), with the `fixed` model instead charging a fixed price and the `free` model charging nothing. Additionally, the last row in the Fee Components table seems to suggest that there is an option to set a fixed amount as the Price Adjustment component, but this is not true: the `margin` price model adds a percentage of the calculated fee, while the `fixed` price model replaces the entire fee calculation with a fixed value.

Recommendation

- Clarify in `FEES.md` that the Fee Calculation Formula only applies when using the `margin` price model, documenting that the other price models are independent of the fee components listed.
- Consider adding a price model that allows setting a fixed Price Adjustment component, rather than using a percentage as in the `margin` price model. This would make the system more flexible, giving more options to Kora node operators. Otherwise, remove the mention of this option from the document.

Status

PRs #243 and #246 update the documentation to clarify that the fee calculation formula is specific to the `margin` price model.

[B07] Unnecessary calculation of transaction fees when using the `fixed` price model

`Severity: Informative` `Recommended Action: Fix Code` `Addressed by client`

When using the `fixed` price model, the `estimate_kora_fee` function in `fees.rs` calculates transaction fees (using `estimate_transaction_fee`) even though those fees will be ignored later and replaced with the fixed price. This introduces unnecessary expensive calculations and RPC calls.

Recommendation

Refactor `estimate_kora_fee` to avoid performing this fee calculation when the price model is set to `fixed` (note that the function already returns early for the `free` price model). It might be worth considering moving the call to `estimate_transaction_fee` inside `get_required_lamports`, since that function already checks the price model.

Status

Commit ec05044 refactors `estimate_kora_fee` to pattern-match on the price model first and call `estimate_transaction_fee` only on the `margin` case. `get_required_lamports` is also split into `get_required_lamports_with_fixed` and `get_required_lamports_with_margin`, thus avoiding having to pattern-match again unnecessarily.

[B08] Price source argument being passed as `Option` unnecessarily

Severity: Informative   Recommended Action: Fix Code   Addressed by client

`estimate_kora_fee` (in `fee.rs`) and `get_required_lamports` (in `price.rs`) take the `price_source` parameter as an `Option<PriceSource>` , but in practice it's always passed a `Some` (there are tests where it's passed a `None` , but nowhere in production code). This complicates the logic of the functions and could potentially lead to issues if they were ever passed a `None` in future updates of the code:

1. In `get_required_lamports` , if `price_source` is `None` when the price model is `fixed` , it will return `*amount` , which is supposed to be the fixed price in token base units, but will be interpreted by the caller as the price in lamports.
2. In `estimate_kora_fee` , if `price_source` is `None` , it will not apply the price adjustment.

Recommendation

Change the function signatures so that the `price_source` parameter is passed as a `PriceSource` (without being wrapped in an `Option`), removing the possibility of receiving a `None` .

Status

Commit 01f69c2 changes the signature of `estimate_kora_fee` and `get_required_lamports` so that `price_source` is no longer passed as an `Option` . The same was also done to the `rpc_client` parameter in `get_required_lamports` .

[B09] No validation that authentication is configured

Severity: Informative   Recommended Action: Fix Code   Addressed by client

As specified in `AUTHENTICATION.md` , authentication is strongly recommended for production deployments of Kora. However, there is no check or warning for the operator if authentication is not enabled.

Recommendation

Add a check in `ConfigValidator::validate_with_result_and_signers` that at least one authentication method is configured, and return a warning if that's not the case.

Status

PR #240 adds a warning to config validation if authentication is not enabled.

[B10] Usage limiter implements a permanent limit that cannot be reset

Severity: Informative   Recommended Action: Document Prominently   Addressed by client

The `usage_limit` module uses a Redis cache to keep a usage counter for each user and prevent users to submit any more transactions once this counter reaches the maximum limit. There is no way to decrease or reset this usage counter (besides the `clear` function, which is meant to only be used in tests), so once a certain wallet reaches the maximum limit, it cannot submit any more transactions.

Recommendation

Document that the only form of usage limiting currently supported by Kora is a permanent limit, and that this limit cannot be reset once reached. Therefore, once the limit is reached for a certain wallet, the user will no longer be able to submit any more transactions using that same wallet.

Status

PR #243 clarifies in `CONFIGURATION.md` the details of the current usage limiting implementation.

[B11] Fee payer policy is not checked during config validation

Severity: Informative   Recommended Action: Fix Code   Addressed by client

If the fee payer policy is misconfigured, it might allow a user to use the fee payer address in undesirable or unsafe ways. Because of this, it is recommended to set all options to `false` unless they are absolutely needed. Despite this, these settings are not checked by the `config_validator` module.

Recommendation

Produce warnings during config validation for any fee payer policy options that are set to `true`, alerting the Kora operator of the consequences of leaving these options enabled.

Status

PR #244 adds new warnings to the config validation, highlighting the risks for each option on the fee payer policy if enabled. Additionally, PR #243 adds additional clarifications in the documentation about these risks.

[B12] Conversion between tokens and lamports uses floating point arithmetic

`Severity: Informative` `Recommended Action: Fix Code` `Addressed by client`

In `token/token.rs`, the functions `calculate_token_value_in_lamports` and `calculate_lamports_value_in_token` use floating-point arithmetic to convert between tokens and lamports. For example:

runtimeverification/_audits_solana-foundation_kora/crates/lib/src/token/token.rs

Line 105 to 122 in c2843ac

```
105     pub async fn calculate_token_value_in_lamports(
106         amount: u64,
107         mint: &Pubkey,
108         price_source: PriceSource,
109         rpc_client: &RpcClient,
110     ) -> Result<u64, KoraError> {
111         let (token_price, decimals) =
112             Self::get_token_price_and_decimals(mint, price_source, rpc_client).await?;
113
114         // Convert token amount to its real value based on decimals and multiply by SOL price
115         let token_amount = amount as f64 / 10f64.powi(decimals as i32);
116         let sol_amount = token_amount * token_price.price;
117
118         // Convert SOL to lamports and round down
119         let lamports = (sol_amount * LAMPORTS_PER_SOL as f64).floor() as u64;
120
121         Ok(lamports)
122     }
```

It is generally not recommended to use floating-point arithmetic for price calculations, since floats are prone to precision issues and can introduce rounding errors in ways that are hard to anticipate.

Recommendation

Rewrite the conversion to use fixed-point rather than floating-point arithmetic, using integers to represent amounts and prices in lamports or token base units. Follow good practices for fixed-point calculations, such as performing multiplications before divisions whenever possible to reduce rounding errors, and upcasting to larger integer types when necessary to avoid overflow in intermediate results.

Status

PR #240 modifies the functions to use the `rust_decimal` crate for arithmetic. PRs #248 and #252 update the calculations to perform multiplication before division and use checked math.

[B13] Signer private key can be accessed outside of signer pool module

Severity: Informative   Recommended Action: Fix Code   Addressed by client

Several functions associated with the `SignerPool` struct in `pool.rs` return a signer or all signers from the pool. Depending on the type of signer, the returned values include the signer's private key. While this is intentional, the information flow should be carefully analyzed to ensure that there is no code path through which a private key can be leaked to the user. Steps can also be taken to reduce the possible attack surface by restricting which functions and modules have access to the private keys.

Recommendation

Reduce the number of functions that return a signer's private key, and the number of modules that have access to them, as much as possible. Be careful that code changes don't accidentally expose private key information to the user.

Status

PR #240 removes the `get_all_signers` function from `signer/pool.rs`. Functions that previously called `get_all_signers` have been modified to instead use `get_signer_pool()?.get_signers_info()`. PR #248 also increases encapsulation of the signer pool by restricting visibility of certain components only to the crate level.

Fuzzing Methodology

Not addressed by client

`libfuzzer` was chosen as the fuzzing library to develop the fuzzing targets, for its ubiquity and ease of setup/use.

In order to maintain determinism and performance for the fuzzing targets, an implementation of a Solana RPC client that is backed by LiteSVM was developed that the Kora library uses for the validator that it talks to. LiteSVM is an in-process Solana VM, which allows the fuzzing targets to be free of any dependency on external processes that Kora would normally need to communicate with.

A set of utilities were made to fuzz randomly over system/spl/spl-2022 instructions for transactions. These instructions are created using information about the accounts and ATAs in the initial state of the LiteSVM instance to keep the transactions within the space of valid transactions in the context of Solana validation, since transactions that can't otherwise be validated on-chain wont pose a threat.

The RPC endpoint `SignTransactionIfPaid` was chosen as the entry point into the fuzzing targets, as it covers most of the pricing and validation scope in the audit.

Fuzzing Targets Overview

Not addressed by client

random_bytes

A target that generates completely random bytes for the `transaction` field of the RPC request and submits it to the Kora server. This is meant to find anything that causes the server to panic or otherwise behave unpredictably.

invalid_instruction

A target that submits a transaction containing an instruction that should not be allowed based on how the Kora server is configured. This tries to find any transactions that somehow circumvent the restrictions that the Kora server's configuration has in place in regards to what instructions are allowed.

balance_check

A target that submits a transaction and - should it be accepted - then checks the balances of the Kora Account/ATAs after execution of that transaction. This is to ensure that the net difference of the Kora signer's balance is no less than the fee required for the transaction.

These fuzzing targets can be found on RV's public fork of Kora: https://github.com/runtimeverification/kora on the branches `fuzzing` (Based on the pinned commit at the beginning of the audit) and `fuzzing-audit-freeze` (Based on the Kora team's fixes for the audit's findings).

Fuzzing Findings

Nothing to preview

[F01] Out of bounds array access

Not addressed by client

The `random_bytes` target found a transaction payload that triggers an out of bounds array access that was described in [A20] Unchecked array indexing in instruction account access.

The transaction in question, within an RPC request to Kora:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "signTransactionIfPaid",
  "params": {
      "transaction": "AOmQAQBxAYMBASEB/YMBAf8BgwE7/4ODYYMBU/8BAQEPyYMBAQEpAQEBIVP/"
  }
}
```

Briefly, this transaction contains a compiled instruction that has a `program_id_index` with an unusually large value of `41`, much larger than the array of accounts in the transaction that it is used to index into. This instruction makes its way to the `uncompile_instructions` function where that index causes a panic on line 107:

runtimeverification/_audits_solana-foundation_kora/crates/lib/src/transaction/instruction_util.rs

Line 100 to 107 in c2843ac

```
100    pub fn uncompile_instructions(
101        instructions: &[CompiledInstruction],
102        account_keys: &[Pubkey],
103    ) -> Vec<Instruction> {
104        instructions
105            .iter()
106            .map(|ix| {
107                let program_id = account_keys[ix.program_id_index as usize];
```

[F02] Lower Fee Payer balance after transaction

Not addressed by client

The `balance_check` target found a transaction that, after aggregating all lamports/token balances, found that the fee payer had lower funds after execution. This transaction was accepted by the `sign_transaction_if_paid` endpoint by Kora.

There wasn't enough time to investigate this transaction, but [A10] SPL token transfers are not accounted for in fee payer outflow is suspected, as the fee payer's balance for an SPL token was drastically reduced.