PUBLIC

Code Assessment

of the Mento Core v3 Smart Contracts

February 17, 2026

Produced for

mento

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Mento team,

Thank you for trusting us to help Mento with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Mento Core v3, according to Scope, to support you in forming an opinion on their security risks.

Mento offers Mento Core v3, a collateralized debt position (CDP)-backed Foreign Exchange (FX) system. Mento Core v3 implements a Fixed Price Market Maker (FPMM) contract, which offers fixed-rate stablecoin swaps based on oracle prices. The system also includes a separately audited Liquity v2 fork, which allows users to open CDPs for different currencies by depositing USDm tokens and minting stablecoins against them. The FPMM can also use the Liquity fork's StabilityPool and redemptions for rebalancing.

The most critical subjects covered in our audit are functional correctness, oracle implementation and its operational implications, as well as rounding issues. Functional correctness has been improved by addressing Incorrect Redemption Fee Formula in CDPLiquidityStrategy. Rounding directions do not consistently follow best practices (though no specific attack was uncovered), see Rounding Should Favor Protocol LPs.

The general subjects covered are documentation and specification, observability, and correctness of preview functions. Documentation is good thanks to detailed descriptions of the Liquity fork and rebalancing flow, although practical liquidity caveats remain, see Forex Market Low-liquidity Considerations. Helper utilities such as the zap parameter generators have known limitations, see Zap Parameters Helpers Ignore Price/State Changes.

We have also provided Notes on important considerations which can aid in understanding the system.

In summary, we find that the codebase provides a satisfactory level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered, and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	7
• Code Corrected	3
• Specification Changed	1
• Code Partially Corrected	1
• Risk Accepted	2
Low-Severity Findings	9
• Code Corrected	6
• Risk Accepted	3

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Mento Core v3 repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	22 October 2025	683af1f4f396d2fbf60eb2c362dd2ee4e184c201	Initial Version
2	22 January 2026	095d4cec3253f1b693e1e1f14490b7a36644920a	First fixes
3	2 February 2026	ebbaffece1ab0f5b62e5fb8981c55240617af436	Second fixes
4	13 February 2026	e77233608e1923b6aa1441aced757e27ca0aa6a9	Third fixes
5	17 February 2026	0e07807c222fabf93556bd48b263483f994b9332	Final Version

For the solidity smart contracts, the compiler version `0.8.24` was chosen.

The following contracts were in scope for the review:

```
contracts/libraries/LiquidityStrategyTypes.sol
contracts/libraries/TradingLimitsV2.sol
contracts/liquidityStrategies/CDPLiquidityStrategy.sol
contracts/liquidityStrategies/LiquidityStrategy.sol
contracts/liquidityStrategies/ReserveLiquidityStrategy.sol

contracts/oracles/OracleAdapter.sol
contracts/oracles/breakers/MarketHoursBreaker.sol

contracts/swap/router/Router.sol
contracts/swap/virtual/VirtualPool.sol
contracts/swap/virtual/VirtualPoolFactory.sol

contracts/swap/FactoryRegistry.sol
contracts/swap/FPMM.sol
contracts/swap/FPMMFactory.sol
contracts/swap/FPMMProxy.sol
contracts/swap/OneToOneFPMM.sol

contracts/tokens/StableTokenV3.sol
contracts/tokens/StableTokenSpoke.sol
```

In Version 2 the following contract was added to the scope:

```
contracts/swap/ReserveV2.sol
```

2.1.1 Excluded from scope

All other contracts are out of scope.

The economic model and the choice of values for configurable parameters are also out of scope.

2.2 System Overview

This system overview describes Version 4 of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Mento offers Mento v3, an evolution of the Mento stablecoin protocol that enables the creation of fiat-pegged stablecoins backed by on-chain collateral.

While Mento v2 operated as a Reserve-backed stablecoin issuance protocol, v3 transitions to a collateralized debt position (CDP)-backed Foreign Exchange (FX) system. This new design allows the protocol to maintain peg stability through over-collateralized positions and redemption mechanisms.

At the core of Mento v3 is the Fixed Price Market Maker (FPMM) contract, which enables oracle-driven, fixed-rate stablecoin swaps. The system integrates a Liquity v2 fork, enabling users to open CDPs by depositing collateral and minting stablecoins against it, as well as rebalancing the FPMM by leveraging Liquity v2's redemption mechanism and its Stability Pool.

This system overview focuses on the components that were introduced or modified in Mento v3 compared to v2.

2.2.1 Fixed Price Market Maker (FPMM)

The FPMM allows users to trade a pair of tokens at fixed prices (plus a fee) that correspond to those of the global FX market. The prices are brought on-chain through a Chainlink oracle. It implements Uniswap v2-like mechanics for swapping and providing liquidity:

- **Oracle-based swaps**: Every swap executes at the external FX rate supplied by the OracleAdapter. Traders receive the oracle-priced amount of the opposite asset minus the configured LP and protocol fees. Pool reserves never determine the quoted price, they only serve as depth to settle trades. The `swapCheck()` function ensures that, at the end of a swap, the reserve value according to the oracle price increased by at least the fees for the swap.
- **Reserve-proportional liquidity**: `mint()` and `burn()` follow the usual constant-share rules: on an empty pool the first LP must deposit both tokens. LP tokens represent pro-rata claims on reserves. Any subsequent LPs must mint by adding tokens in the same proportions as the current reserves. `burn()` returns each token in proportion to the reserves. LP minting and burning is independent of the oracle price used for swaps.
- **Threshold-based rebalancing**: As swap prices are independent of reserves, the FPMM reserves can become imbalanced. When the pool's reserve-implied price diverges from the oracle rate by more than the configured threshold, a registered LiquidityStrategy can call `rebalance()`. Strategies take the token which there is too much of out of the pool, source the opposite asset from their backing facility (e.g., Liquity StabilityPool or the Reserve), and deposit it to the pool. The liquidity source is awarded a rebalancing incentive, which is applied by giving a discounted price during rebalancing compared to the oracle price. The FPMM's `_rebalanceCheck()` enforces that, after the strategy returns tokens, the pool's reserve price must have moved closer to the oracle's price and that the token coming back into the pool must cover at least the oracle-priced value of what left.
- **Per-token trading limits**: Governance can assign 5-minute and 1-day net-flow caps to each token, via the TradingLimitsv2 library.

- **Modifiable parameters**: Parameters such as LP fee, protocol fee, incentive, thresholds, oracle adapter, reference feed, invert flag, and protocol fee recipient are owner-settable. Storage follows the ERC-7201 pattern so proxies can be upgraded safely.

During swaps, if the caller passes nonempty data in the call, the pool invokes `IFPMMCallee.hook()` on the recipient after sending the output tokens to support flash-swap callbacks. During rebalances, the pool calls `ILiquidityStrategy.onRebalance()` on the calling strategy so that it can source the opposite asset before the transaction settles.

TradingLimitsv2 is a library for implementing trading limits on asset net-flows. It tracks two sliding windows: L0 (5 minutes) and L1 (1 day). Net flows are stored at 15-decimal precision. Limits are configured in native token units but scaled and stored at 15-decimal precision. Swaps that would push cumulative inflows or outflows beyond those caps revert, until sufficient time has passed and the limit resets. This limits the maximum value that can be lost to arbitrage within a short time. For example, it limits value loss in case the oracle price lags the true market price during periods of high volatility. Trading limits can be set in the FPMM by the `owner` using the `configureTradingLimit()` function.

OneToOneFPMM is an extension of FPMM. It overrides `_getRateFeed()` to always return a one-to-one oracle price after verifying the reference feed is valid via the oracle adapter. This hardcodes the exchange rate while keeping breaker protections. It is intended to be used for stablecoin-to-stablecoin pools where both tokens represent the same currency. Note that such a pool should be expected to lose value until the oracle starts reverting in case one of the stablecoins loses its peg.

FPMMFactory allows governance to deploy FPMM pools as FPMMProxy instances. The governance registers implementations, manages default parameters (bounded to 2% combined fee and ≤10% thresholds), and can deploy pools with custom configurations or reuse default values. Every deployment records the address under both (token0, token1) and (token1, token0) to prevent duplicates, and marks the created pool as valid for Router lookups. Ownership is transferred to the owner address provided during initialization. FPMMFactory also offers a `getOrPrecomputeProxyAddress()` function to get the precomputed or current proxy address for a token pair.

2.2.2 Liquidity Strategies

Liquidity strategies are used to rebalance FPMM pools when they become imbalanced. The strategy is compensated for this service by receiving a better price during rebalancing, as defined by the strategy's incentive parameters.

LiquidityStrategy is a base contract that stores a set of rebalancing pools. Governance can call `addPool()` to register an FPMM plus its debt token, rebalance cooldown, and incentive, as well as remove pools using `removePool()`. When `rebalance(pool)` is called, the strategy calculates how many tokens need to be exchanged to bring the pool's reserves in line with the rebalance threshold. If the ideal amounts cannot be exchanged due to a limitation, it uses the highest possible amount instead. It calls `IFPMM.rebalance()` to receive tokens from the FPMM. Then, the rebalance hook calls back into the strategy to complete the exchange. A liquidity strategy can only rebalance a pool if the rebalance cooldown has passed since the last rebalance and the pool is imbalanced by at least the rebalance threshold. Concrete strategies override `_clampExpansion`, `_clampContraction`, and `_handleCallback` to receive and send funds from their respective liquidity source.

The CDPLiquidityStrategy contract connects FPMMs to Liquity v2's Stability Pool and redemption flows:

- Expansions withdraw debt from the Liquity Stability Pool using the newly added `swapCollateralForStable` path and send collateral tokens from the FPMM in return. A `stabilityPoolPercentage` parameter caps the proportion of pool deposits that can be used, and the Liquity pool's `MIN_BOLD_AFTER_REBALANCE` constant enforces a floor of remaining debt token. If insufficient debt is available, the helper recomputes the collateral owed to match the reduced amount.
- Contractions redeem FPMM debt via `ICollateralRegistry.redeemCollateralRebalancing()` with a fixed redemption fee

derived from the strategy's incentive parameters. A `maxIterations` field lets governance control batched redemptions in Liquity's redeem loop.

- The callback path approves the Liquity Stability Pool or Collateral Registry for transfers, executes the swap/redemption, and transfers the owed assets back to the FPMM after verifying balances.
- The rebalance parameters can be set via the `CDPConfig` value when the pool is added or afterwards by the governance using `setCDPConfig()`.

The ReserveLiquidityStrategy contract maintains compatibility with the Mento Reserve:

- Expansions mint the debt stablecoin in exchange for receiving the corresponding collateral from the FPMM. Minting uses `mint` to create tokens directly in the pool's balance. There are no limits to the amount of tokens that can be minted, as long as sufficient collateral is provided.
- Contractions pull collateral from the reserves and burn the stablecoin. If the Reserve does not hold enough collateral to satisfy the ideal transfer, the amount is capped to the balance of the reserve.
- The strategy exposes `setReserve()` so governance can change the underlying reserve if needed.

The LiquidityStrategyTypes contract centralizes the math used by every strategy. It builds a Context by querying the target FPMM for token, metadata, reserves, oracle prices, and incentive settings, then exposes helpers to convert between debt/collateral units, normalize decimals, and generate Action objects that express expansion (push debt in, pull collateral out) or contraction (pull debt out, push collateral in) steps. These helpers ensure consistent handling of decimals, rate scaling, basis-point incentives, and hook callback data across strategies. It ensures that strategies take and return the token amounts the FPMM expects.

2.2.3 Stable Tokens

The existing Mento stablecoins will have their implementation upgraded from StableTokenV2 to StableTokenV3. StableTokenV3 remains an ERC-20 token with EIP-2612 permits, but now implements a role-based mint/burn/operator permission system that the governance can adjust after deployment:

- `minters` are allowed to call `mint()`.
- `burners` can burn from arbitrary accounts.
- `operators` can transfer tokens between addresses using the `sendToPool()` and `returnFromPool()` functions, which are required for compatibility with the StabilityPool. Only the StabilityPool contract is expected to be added as an operator.

Initialization optionally mints seed balances to specific addresses, only when deploying a new token and not when upgrading from a v2 token. The contract is upgradable. Ownership is transferred to governance, and new minters/burners/operators can be added or removed via setter functions emitting events.

StableTokenSpoke mirrors the mint/burn logic for remote chains. It omits operators and the StabilityPool-related functions, but governance can still configure minters/burners and seed initial balances on deployment if needed.

2.2.4 Virtual Pools

The VirtualPool contract integrates the Mento v2 Broker to the new router, exposing existing PoolExchanges via the BiPoolManager. It provides a Uniswap v2-like interface while using underlying BiPoolManager exchange logic. Each virtual pool hardcodes a Broker and an exchange, and reports reserves/metadata by reading the bucket balances from the underlying PoolExchange. During initialization, the VirtualPool approves unlimited transfers of both its tokens to the Broker, so swaps do not require additional permissions. Swaps trigger the Broker's `swapOut()` function, and then forward the received tokens to the designated recipient. Because the buckets carry their own spread, `protocolFee()` simply reflects the configured spread, and flash swaps are not supported (data must be empty in `swap()` calls).

The VirtualPoolFactory contract deploys these wrappers per exchange pair. It determines canonical (token0, token1) ordering, records the deployed address in bidirectional mappings so the VirtualPool can be uniquely identified regardless of the tokens ordering, and exposes the IRPoolFactory interface, allowing `poolFor()` to return virtual pools when the router is configured with this factory. VirtualPoolFactory also offers the `getOrPrecomputeProxyAddress()` function, which returns the existing pool address if deployed, or precomputes the deterministic CREATE3 address for the token pair otherwise.

2.2.5 Oracle & Breaker

OracleAdapter is an ownable and upgradable contract that acts as a central adapter for oracle price feeds and market condition validation. It integrates with SortedOracles for price data and BreakerBox for circuit breakers, and validates FX market hours and trading modes before allowing operations. When calling `getFXRateIfValid()`, the adapter ensures:

1. The breaker box reports bidirectional trading mode for this feed.
2. The rate timestamp is newer than `reportExpiry` seconds.
3. The FX market is currently open (checked via MarketHoursBreaker).

If these 3 conditions are not fulfilled, the call reverts. `getRate()` checks the same conditions but does not revert if one of them is false, and instead returns the information in a `RateInfo` struct. Governance can modify any dependency via `setSortedOracles()`, `setBreakerBox()`, `setMarketHoursBreaker()`, or `setL2SequencerUptimeFeed()`, allowing the system to migrate data providers without redeploying pools.

MarketHoursBreaker is a breaker that reverts when its `shouldTrigger()` function is called during FX weekend hours (when the market is closed, from Friday 21:00 UTC to Sunday 23:00 UTC and on January 1st / December 25th, as well as from 22:00 UTC on Dec 24 and Dec 31). It effectively pauses new price consumption across the protocol until the market reopens and trading resumes. This means that any functionality depending on oracle prices, such as swaps and rebalances in FPMM, will revert while markets are closed.

2.2.6 Router

The Router is the user-facing entry point for swaps and liquidity management across all Mento pools. It extends the Velodrome/Aerodrome router to handle FPMMs, VirtualPools, and any future pool type that registers an IRPoolFactory in the governance-controlled FactoryRegistry. Its key capabilities are:

- The router holds a `factoryRegistry` that lists every approved pool factory. When a user specifies a route hop, the Router resolves the correct pool address via the hop's factory by using `poolFor()` (falling back to the default FPMM factory if none is provided) and rejects pools from unapproved factories, ensuring that only official pools can be used. `poolFor()` uses each factory's `getOrPrecomputeProxyAddress()` to retrieve the pool address, providing deterministic lookups.
- When executing a swap, users submit an ordered list of Route structs describing each hop (token in/out plus optional factory address). The Router chains calls to each pool's swap function, forwarding the correct amount of the intermediate token at every hop. It offers the standard `swapExactTokensForTokens()` function, reverting if the final output is below `amountOutMin`.
- `addLiquidity()` and `removeLiquidity()` wrap the FPMM's minting and burning flows, calculating optimal token ratios based on current reserves and transferring the required assets.
- `quoteAddLiquidity()`, `quoteRemoveLiquidity()`, and `getAmountsOut()` help users or wallets precompute expected results for a given pool or multi-hop trade.
- Zap function `zapIn()` converts a single input token A into the precise amounts of tokens B and C needed to provide liquidity to pool(B,C) in one transaction (A can be equal to B or C). `zapOut()` provides the opposite functionality, removing liquidity from pool(B,C) and receiving a single output token A in return. Both functions expect a `Zap` struct as well as swapping routes as inputs, and

make use of the quoting functions to calculate the ideal amounts of tokens. Functions `generateZapOutParams()` and `generateZapInParams()` can be used to generate the parameters required for zapping in or out. However, these only provide rough estimates, since the actual execution can differ due to pool state changes (e.g. as result of the swap).

- Every state-changing function enforces a deadline, reverting if the transaction executes after the timestamp, and validates the caller's minimum amount conditions.

The Router inherits from `ERC2771Context`, ensuring compatibility with meta transactions.

FactoryRegistry is an upgradable and ownable contract acting as a whitelist of pool factories. The registry maintains a fallback factory (always approved) and only considers pools from approved factories valid, ensuring that the router can only use whitelisted implementations.

2.3 Changes in Version 2

The following changes were introduced in **Version 2** of the contracts:

- Rebalancing now rebalances to the threshold, instead of to a 50/50 value distribution.
- A new ReserveV2 contract was introduced. It is a simplified version of the Reserve contract, only supporting the functions needed by the ReserveLiquidityStrategy.
- All liquidity strategies are now upgradeable via proxy patterns.
- TradingLimitsV2 no longer counts swap fees in the trading limits.
- The rebalance incentive model was split into four separate parameters per pool: a protocol incentive and a liquidity source incentive, each configured independently for expansion and contraction. A protocol fee recipient receives the protocol share during rebalance callbacks.
- The rebalance callback was renamed from `hook` to `onRebalance`.
- Small swap amounts of tokens with more than 15 decimals can now round to zero in `TradingLimitsV2`.
- The separate `governance` address in `FPMMFactory` was removed. It now just uses `owner`. The maximum fee was increased from 1% to 2%.
- MarketHoursBreaker was modified to also revert from 22:00 UTC on Dec 24 and Dec 31.
- VirtualPoolFactory now uses CREATE3 via the CREATEX deployer contract for deterministic pool deployment. The `getOrPrecomputeProxyAddress()` function now precomputes the deterministic CREATE3 address for undeployed token pairs.

2.4 Trust Model

Roles and trusted components are enumerated below. All contracts in scope are upgradeable unless noted otherwise. Proxy admins (usually governance) can replace implementations without on-chain limitations.

- Governance (Fully trusted): Assumed to control all ownable contracts (e.g., FactoryRegistry, FPMMFactory, Liquidity Strategies, OracleAdapter, StableTokenV3/Spoke, etc.). Governance can deploy arbitrary pools, whitelist factories, upgrade proxies, assign strategy parameters, and appoint minters/burners/operators. Compromised governance could mint unbacked stablecoins, seize users' funds, approve malicious strategies, or swap oracle feeds to corrupt prices.
- ReserveV2 Owner (Fully trusted): Registers ReserveV2 roles. A compromised owner could register a malicious spender to drain all reserve assets.
- ReserveV2 Liquidity Strategy Spenders (Fully trusted): Can move collateral from the reserve to any address.

- ReserveV2 Reserve Manager Spenders (Partially trusted): Can transfer collateral only to other whitelisted reserve addresses, and can transfer stable assets (collected fees) to any address.
- Liquidity Strategies (Partially trusted): Only addresses added via `FPMM.setLiquidityStrategy()` can call rebalance() on a given FPMM. The FPMM's `_rebalanceCheck()` enforces that the strategies do not extract more value than the incentive during a rebalance. Anyone can call rebalance on a registered pool, but the strategies and the FPMM logic enforce invariants that protect the LP's assets.
- Liquity Components (Partially trusted): The Liquity StabilityPool, CollateralRegistry, SystemParams, and overall governance must behave per Liquity's economic assumptions. A malicious Liquity deployment could block swaps, overcharge redemption fees, or steal debt/collateral handed to it by CDPLiquidityStrategy.
- StableToken Minters/Burners/Operators (Fully Trusted): Minters can inflate supply, burners can destroy user balances, operators can move funds between any addresses. These roles must be tightly controlled and are assumed to only be held by smart contracts.
- FPMM FeeSetter (Partially trusted): Can modify the LP fee, Protocol fee, and Rebalance Incentive, subject to safety bounds (0 to hard-coded limits).
- Users / LPs (Untrusted): LPs provide liquidity via FPMMs, and users trade through the Router or directly with pools. They can attempt to front-run trading limits or rebalances, but contract-level checks enforce invariants. Anyone can call the Router's swap/liquidity/zap functions.
- External oracles (Partially trusted): The SortedOracles contract and its price sources (such as Chainlink feeds) are assumed to provide accurate and timely FX rates. A compromised oracle could misprice swaps, leading to losses for LPs. The maximum loss within a short time is bounded by the trading limits.
- FX Market Assumptions: The system assumes FX markets are closed only during the periods encoded in MarketHoursBreaker. If the breaker is misconfigured or if markets experience extraordinary closures, oracle updates may not revert even if the price values should not be used.

All tokens used in Mento v3 are assumed to be non-malicious and to behave according to the specifications. They are assumed to be ERC20-compliant with no special features (no transfer hooks (reentrancy), rebasing, fees on transfer, etc.).

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Open Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	3

- MarketHoursBreaker Reports Inaccurate Market Hours Code Partially Corrected
- Rebalance Redemptions Can Hit Max Iterations Risk Accepted
- Rounding Should Favor Protocol LPs Risk Accepted

Low-Severity Findings	3

- Admin Reentrancy Possibilities Risk Accepted
- OZ ERC2771 Has a Known Issue Risk Accepted
- getAmountOut Missing Insufficient Liquidity Check Risk Accepted

5.1 MarketHoursBreaker Reports Inaccurate Market Hours

Correctness Medium Version 1 Code Partially Corrected

CS-MECO-002

In MarketHoursBreaker, the reported market hours do not align with the actual trading hours of global forex markets. The following discrepancies were identified:

1. Daylight Saving Time is not taken into account. As a result, the market is reported as closed during the last hour of trading on Fridays, if it is winter time in the U.S. Additionally, the market is reported as closed during the first two hours of trading on Sundays, if it is summer time in the U.S., and during the first one hour of trading if it is winter time.
2. The `_isHoliday` function does not take into account time zones. It determines holiday dates based on UTC time. This is inaccurate, as the market will actually close at 22:00 UTC on the previous day for both Christmas and New Year's Day. The market will be reported as open between 22:00 UTC and 00:00 UTC, even though it is closed. Similarly, the market will open again at 22:00 UTC on the holiday, but the code will report it as closed until 00:00 UTC.

For further notes on the topic, see Forex market low-liquidity considerations

---

Code partially corrected:

The holiday timing issue has been addressed. The `_isHoliday` function now correctly closes the market at 22:00 UTC on Christmas Eve (December 24th) and New Year's Eve (December 31st), matching actual forex market behavior.

However, the Daylight Saving Time issue remains unaddressed. Mento acknowledged this limitation and will accept the additional 1 hour downtime during the first release of V3.

5.2 Rebalance Redemptions Can Hit Max Iterations

Design | Medium | Version 1 | Risk Accepted

CS-MECO-003

In CDPLiquidityStrategy, if `_handleCallback()` does not get the full redemption amount that is expected, the function reverts. For example, this could happen due to hitting the configured `maxIterations`. There may be conditions where redemption rebalances can remain impossible until the redemption list evolves, e.g., if there are many small troves at its beginning.

Risk accepted:

Mento has steated they will configure `maxIterations` to a high value within the block gas limit such that a revert is very unlikely to happen.

5.3 Rounding Should Favor Protocol LPs

Security | Medium | Version 1 | Risk Accepted

CS-MECO-005

In FPMM, the `swap` function performs rounding on token amounts in multiple places. In some cases, this rounding can lead to a small loss of value for the protocol. It is considered best practice to always round in favor of the protocol's LPs to ensure that they do not lose value due to rounding errors. Small losses can potentially lead to critical exploits when combined with other vulnerabilities. While we have not identified a specific exploit that takes advantage of this issue, it is important to address it to prevent potential vulnerabilities.

The `swap` function does not round in favor of the protocol LPs in the following places:

1. When calculating the initial reserve value, the `_totalValueInToken1Scaled` function rounds down, which can overestimate the user's transferred amount. The same function rounds down later in `_swapCheck`, but there it benefits the protocol.
2. In `_swapCheck`, the `expectedAmountIn` is calculated using `_convertWithRate()`, which rounds down. Additionally, the fees are rounded down twice. Once when calculating `fee0` and `fee1`, and again when calculating `fee0InToken1`.

The sum of these rounding issues can lead to a user receiving slightly more tokens than they should. It is safer to round in a way such that protocol LPs always benefit from rounding.

Similar rounding happens in `rebalance()`. Swap and rebalance should have consistent rounding behavior.

It should be noted that changing the rounding behavior may have implications on other functions. For example, the `getAmountOut` function must never return an amount greater than what the user would actually receive after fees and rounding. This should be validated through extensive fuzz testing.

Risk accepted:

Mento acknowledged that fixing all rounding issues across the codebase would take more effort than originally anticipated and states:

```
Given that this doesn't present an immediate/significant
risk to the protocol we will provide a fix in a future
upgrade/follow up audit.
It's at the top of our list and we are actively working on it.
```

5.4 Admin Reentrancy Possibilities

Design Low Version 1 Risk Accepted

CS-MECO-007

The functions in the contracts are generally protected against reentrancy using reentrancy guards. However, admin functions do not have reentrancy guards. This could be problematic, even if the admin behaves correctly, as the admin contract is a governance timelock that allows for permissionless execution of initiated transactions after the delay has passed.

Depending on the parameters being changed, this may lead to vulnerabilities where an attacker could initiate a transaction with an arbitrary call (such as a swap hook), then trigger the admin function to change parameters, and then complete the transaction with the new parameters.

Note that reentrancy guards only protect against reentrancy within the same contract. To fully eliminate this risk (assuming the admin behaves correctly), the permissionless execution of timelocked transactions in the governance contract would need to be disabled.

Risk accepted:

Mento stated:

```
We will keep this in mind for future Governance proposals/parameters upgrade,
however permissionless execution is an important part of our governance system
which we intend to keep.
```

5.5 OZ ERC2771 Has a Known Issue

Security Low Version 1 Risk Accepted

CS-MECO-010

The ERC2771 contract from the OpenZeppelin library uses an outdated version (v4.7.0). This version has a known issue.

Generally, only the last minor version of a major release gets critical security fixes from OpenZeppelin, as stated in their security policy.

Risk accepted:

Mento reviewed the OpenZeppelin issue and determined it irrelevant to their use case. Mento therefore decided to keep the current version as is.

5.6 getAmountOut Missing Insufficient Liquidity Check

Design | Low | Version 1 | Risk Accepted

CS-MECO-013

In FPMM, `getAmountOut()` computes the oracle-priced output for a given input but never verifies that the pool actually has enough reserves to cover the result. For example, if reserves are zero (before any liquidity is added or after the pool has been emptied by burns), `getAmountOut()` still returns a positive amount even though the subsequent swap call would revert. Divergent behavior between the view helper and the executable path can be misleading for users and integrating contracts.

Risk accepted:

Mento has accepted the risk, but has decided to keep the code unchanged.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Open Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	4

- Excessive Approval in CDPLiquidityStrategy._handleCallback() Code Corrected
- Incorrect Redemption Fee Formula in CDPLiquidityStrategy Code Corrected
- Redemption Fee Calculation Does Not Round Correctly Code Corrected
- Zap Parameters Helpers Ignore Price/State Changes Specification Changed

Low-Severity Findings	6

- Redemption Incentive Depends on Protocol Fee Code Corrected
- Inconsistent Denominators in Rate Feed Functions Code Corrected
- FPMMFactory Redundant Governance Role Code Corrected
- Missing FeeCurrency Interface Code Corrected
- TradingLimits Incorrect Rounding Correction Code Corrected
- Unnecessary Price Fetching Code Corrected

Informational Findings	13

- Deprecated Comments in _rebalanceCheck Code Corrected
- Deprecated Fee-on-transfer Tokens Functions Code Corrected
- FPMM Mint Event Omits Recipient Address Code Corrected
- Incorrect ERC20Permit Initialization Code Corrected
- Missing Token Decimals Check in FPMM Code Corrected
- Outdated FPMM Invariant Code Corrected
- Overlapping Hook Function Signatures Code Corrected
- Repeated Pool Lookups in Router _Swap Code Corrected
- StableTokenV3.initializeV3 Does Not Update EIP712 Version Code Corrected
- TradingLimitsV2 Accepts Unknown Flag Bits Code Corrected
- Unindexed Address Parameters in FPMM Configuration Events Code Corrected
- VirtualPoolFactory Returns Zero for Undeployed Pools Code Corrected
- getAmountsOut Returns Zero on Invalid Swap Code Corrected

6.1 Excessive Approval in CDPLiquidityStrategy._handleCallback()

Correctness | Medium | Version 2 | Code Corrected

CS-MECO-029

In `CDPLiquidityStrategy._handleCallback()`, the function approves the CollateralRegistry contract for `collAmount` collateral, but later only consumes `collAmount - protocolIncentive` when calling `swapCollateralForStable()`. As such, a residual approval of `protocolIncentive` will persist after the first time this function is called.

The project uses OpenZeppelin's `safeApprove` function as a wrapper to support uncommon collateral types. However, this function is considered deprecated and has some footguns. One of them is that it reverts when the approval is to be set to a non-zero value if it is currently non-zero.

```
function safeApprove(IERC20 token, address spender, uint256 value) internal {
  // safeApprove should only be called when setting an initial allowance,
  // or when resetting it to zero. To increase and decrease it, use
  // 'safeIncreaseAllowance' and 'safeDecreaseAllowance'
  require(
    (value == 0) || (token.allowance(address(this), spender) == 0),
    "SafeERC20: approve from non-zero to non-zero allowance"
  );
```

As a result of the residual approval of `protocolIncentive` after the first call to `swapCollateralForStable()`, the `_handleCallback()` function will revert on any subsequent call, blocking rebalancing operations.

Code corrected:

The function was modified to approve for `collAmount - protocolIncentive` instead of `collAmount` collateral.

6.2 Incorrect Redemption Fee Formula in CDPLiquidityStrategy

Correctness | Medium | Version 1 | Code Corrected

CS-MECO-001

In CDPLiquidityStrategy, `_calculateMaxRedeemableDebt` mirrors Liquity's redemption-fee logic, so that the strategy can clamp contractions when the Liquity fee would exceed the FPMM incentive budget. However, the implementation multiplies by `REDEMPTION_BETA` when computing the fee:

```
uint256 redemptionFee = decayedBaseFee + (contractionAmount * 1e18 * redemptionBeta) / totalDebtTokenSupply;
```

Whereas the formula used in `._getUpdatedBaseRateFromRedemption()` Liquity's CollateralRegistry's divides by `REDEMPTION_BETA`:

```
uint256 redeemedBoldFraction = _redeemAmount * DECIMAL_PRECISION / _totalBoldSupply;
uint256 newBaseRate = decayedBaseRate + redeemedBoldFraction / systemParams.REDEMPTION_BETA();
```

This discrepancy causes the strategy to overestimate the redemption fee, leading to unnecessarily restrictive contraction limits or reverts.

Code corrected:

The CDPLiquidityStrategy no longer obtains collateral by calling the standard redeem function. Instead, it uses the entrypoint `CollateralRegistry.redeemCollateralRebalancing`, which takes the redemption fee charged to the strategy as an argument. The redemption fee is independently calculated by the strategy and does not depend on the base rate or other Liquity-internal parameters.

6.3 Redemption Fee Calculation Does Not Round Correctly

Correctness | Medium | Version 1 | Code Corrected

CS-MECO-004

In CDPLiquidityStrategy, the expected collateral received from a redemption is calculated. If the received amount is lower than expected, the redemption will revert.

As multiple troves can be redeemed in a single redemption, there can be rounding errors from each trove that accumulate. This accumulation of rounding errors is not taken into account in CDPLiquidityStrategy.

As a result, there can be cases where the redemption should succeed, but reverts due to the received amount being slightly lower than expected because of accumulated rounding errors.

Code corrected:

A tolerance mechanism was added to handle precision loss in redemptions. The contract now allows redemptions to succeed even if there is a small shortfall (up to `REDEMPTION_SHORTFALL_TOLERANCE`). When a shortfall occurs within the tolerance, the contract subsidizes it from its own balance and emits a `RedemptionShortfallSubsidized` event. Surpluses from redemptions are retained to offset future shortfalls.

In case shortfalls are more common that surpluses, a small amount of external funds must be added to cover them.

Note on the audit process:

This issue was identified both by the auditors and by the Mento team in parallel during the audit.

6.4 Zap Parameters Helpers Ignore Price/State Changes

Design | Medium | Version 1 | Specification Changed

CS-MECO-006

In Router, `generateZapInParams()` and `generateZapOutParams()` build recommended `amountOutMin/amountAMin` values by chaining `getAmountsOut` and `quote{Add,Remove}Liquidity` calls, assuming pool prices and routing state stay constant between the view call and the actual zap.

In practice, pool state can change between parameter generation and execution. Also, `zapIn()` can trade in the pool, which changes the reserves and will lead to reverts, as this change is not taken into

account in the minimum amounts for providing liquidity. Additionally, some limitations in the pools are not taken into account, such as reduced available liquidity for the swap after removing liquidity in `zapOut()`. Also, trading limits (from TradingLimitsV2) could be hit and cause a revert, as they are not taken into account.

Due to these factors, the suggested values can be stale or outright incorrect. Rather than returning optimistic guesses, a more robust solution would be for off-chain integrators to simulate the actual zap call via `eth_call` to capture the precise values under current pool conditions. Documenting this limitation would prevent users from relying on numbers that don't reflect real execution.

Specification changed:

The limitations of these helper functions have been documented in the contract.

6.5 Redemption Incentive Depends on Protocol Fee

Design Low Version 3 Code Corrected

CS-MECO-027

In CDPLiquidityStrategy, _calculateRedemptionFee() intentionally inflates the incentive fee rate so that trove owners are compensated based on the original debt amount (including the protocol's cut).

However, the trove owner only experiences a redemption of the reduced amount.

As a result, the trove owner's compensation is dependent on the protocol fee.

For example, if 1000 BOLD leaves the pool and the protocol takes 10 BOLD, only 990 BOLD is redeemed from the trove. The trove owner's incentive fee is applied to the full 1000.

The Stability Pool is also affected by a similar coupling.

Code corrected:

The CDPLiquidityStrategy has been updated to no longer inflate the fee rate in Version 4. The fee of liquidity sources (i.e., trove owners or stability pool) is now calculated on the amount minus the protocol fee. As such, the amount after applying protocol and incentive fees is now calculated as $(1 - f_{protocol}) \times (1 - f_{liquidity})$ instead of $1 - f_{protocol} - f_{liquidity}$.

6.6 Inconsistent Denominators in Rate Feed Functions

Correctness Low Version 2 Code Corrected

CS-MECO-030

The `FPMM._getRateFeed()` function accepts denominators other than 1e18. However, the price feed used in Liquity v2 fork (i.e. `FxPriceFeed.fetchPrice()`) does not support this flexibility and reverts with a denominator other than 1e18.

The protocol assumes throughout the code that both protocols are using the same price (i.e. when rebalancing with the Liquity V2 fork). If the denominator were set to anything other than 1e18, the Mento core could continue operating while the Liquity v2 fork breaks.

Code corrected:

The following assertion was added in `OracleAdapter._getOracleRate()`:

```
assert(denominator == 1e24);
```

As both the numerator and denominator are then divided by 1e6, this ensures that the denominator received by the `FPMM` is 1e18.

6.7 FPMMFactory Redundant Governance Role

Design | Low | Version 1 | Code Corrected

CS-MECO-008

In FPMMFactory, there is a `governance` role. However, the `owner` role is already expected to be held by governance, and covers all needed governance actions. As a result, the `governance` role is redundant and can be removed to simplify the contract.

Code corrected:

The redundant `governance` role has been removed from FPMMFactory. The contract now uses only the `owner` role for governance actions.

6.8 Missing FeeCurrency Interface

Design | Low | Version 1 | Code Corrected

CS-MECO-009

According to Celo's specification, fee tokens are expected to implement the IFeeCurrency interface:

```
creditGasFees(address[] calldata recipients, uint256[] calldata amounts)
```

StableTokenV3 does not implement this interface.

Code corrected:

StableTokenV3 has been modified to implement the IFeeCurrency interface.

6.9 TradingLimits Incorrect Rounding Correction

Correctness | Low | Version 1 | Code Corrected

CS-MECO-011

In TradingLimitsV2, if the `scaledDelta` value rounds down to zero, it is set to `1e3`, to avoid rounding to zero. However, it should actually be set to `1`.

Code corrected:

The trading limits implementation was updated to track only the first 15 decimals of precision. Values below this threshold are now rounded to zero.

6.10 Unnecessary Price Fetching

Design | Low | Version 1 | Code Corrected

CS-MECO-012

In FPMM, the `_rebalanceCheck` function fetches the price a second time, even though the price was already fetched in the `rebalance` function. The code assumes that the price is the same in both calls, so there is no need to fetch it again. Currently, there is no untrusted external call made in the liquidity strategies, so the price cannot change between the two calls. However, if in the future a strategy is added that makes an untrusted external call before the second price fetch, the price could change between the two calls, which would be incorrect behavior.

Code corrected:

The second price fetch has been removed from the code.

6.11 Deprecated Comments in _rebalanceCheck

Informational | Version 2 | Code Corrected

CS-MECO-028

The `_rebalanceCheck` function contains outdated comments that reference a "10 wei difference" related to rounding and precision loss. These comments no longer accurately reflect the current implementation.

Code corrected:

The comments were removed.

6.12 Deprecated Fee-on-transfer Tokens Functions

Informational | Version 1 | Code Corrected

CS-MECO-014

In `Router.sol`, the functions `swapExactTokensForTokensSupportingFeeOnTransferTokens` and `_swapSupportingFeeOnTransferTokens` are deprecated and should be removed, as fee-on-transfer tokens are explicitly not supported.

Code corrected:

The deprecated functions for fee-on-transfer tokens have been removed from the Router contract.

6.13 FPMM Mint Event Omits Recipient Address

Informational | Version 1 | Code Corrected

CS-MECO-015

In FPMM, the `mint` function emits a `Mint` event with (sender, amount0, amount1, liquidity) but does not include the address that receives the LP tokens:

```
emit Mint(msg.sender, amount0, amount1, liquidity);
```

In contrast, the `Burn` event includes the `to` address that received the withdrawn tokens.

Code corrected:

The recipient address has been added to the `Mint` event.

6.14 Incorrect ERC20Permit Initialization

Informational | Version 1 | Code Corrected

CS-MECO-016

In `StableTokenV3.sol`, the `initialize` function incorrectly initializes the `ERC20Permit` with the token symbol instead of the token name. According to OZ's ERC20Permit documentation:

```
It's a good idea to use the same `name` that is defined as the ERC20 token name.
```

StableTokenV2 also incorrectly initialized ERC20Permit with the symbol, so it must be weighed if changing it now is worth it, or if compatibility is more important.

Code corrected:

The initialization has been updated to pass the token name instead of the token symbol, following the ERC20Permit documentation.

6.15 Missing Token Decimals Check in FPMM

Informational | Version 1 | Code Corrected

CS-MECO-018

`FPMM` does not include checks to ensure that the token's decimals do not exceed 18.

However, it is assumed throughout the contract that tokens have a maximum of 18 decimals. For example, in `_totalValueInToken1Scaled()`:

```
amount1 = amount1 * (1e18 / $.decimals1);
```

If `$.decimals1` is greater than 18, the division will round down to zero, leading to `amount1` being equal to zero.

Code corrected:

A decimals validation check has been added to the FPMM initialize function to ensure tokens have not more than 18 decimals.

6.16 Outdated FPMM Invariant

Informational Version 1 Code Corrected

CS-MECO-019

In FPMM, the contract comment states the following invariant:

```
Rebalance does not change the direction of the price difference
```

However, the design was changed to allow rebalances that change the direction of the price difference, as long as the price difference after the rebalance is small enough.

Version 2:

The comment was updated to reflect the behavior of the code in Version 1:

```
Rebalance can change the direction of the price difference but
not by more than the rebalance incentive
```

Code corrected:

In Version 3, the comment was updated again to reflect this version's behavior of the code (rebalance can no longer change the direction of the price difference).

6.17 Overlapping Hook Function Signatures

Informational Version 1 Code Corrected

CS-MECO-020

In FPMM, the `swap` and `rebalance` functions both call the `hook` interface. This may be a dangerous pattern, as the swap hook can be called on an arbitrary `to` address, while the rebalance hook is only called on registered liquidity strategies. The current strategies do sufficient validation to ensure that their hook cannot be called through `swap()` with arbitrary data, so there is no immediate issue. However, if a new strategy added in the future does not perform sufficient validation, or if it allows calling `swap()` from the strategy, it may be vulnerable to attacks through the swap hook.

Using separate function signatures for the swap and rebalance hooks could help prevent this class of issues in the future, by making it impossible to call the rebalance hook through `swap()`.

Code corrected:

The rebalance hook has been renamed from `hook` to `onRebalance` to prevent any potential collision.

6.18 Repeated Pool Lookups in Router _Swap

Informational | Version 1 | Code Corrected

CS-MECO-021

In Router, `_swap()` computes two pool addresses in each hop using `poolFor()`, one for the destination address and another for executing the swap:

```
address to = i < routes.length - 1 ? poolFor(routes[i + 1]
    .from, routes[i + 1].to, routes[i + 1].factory) : _to;
IRPool(poolFor(routes[i].from, routes[i].to, routes[i].factory))
    .swap(amount0Out, amount1Out, to, new bytes(0));
```

Since `poolFor()` deterministically derives the address from the token pair and factory, calling it twice in the same loop iteration wastes gas. Caching the result of `address to` and reusing it for the next loop iteration could eliminate the duplicate lookup and slightly reduce per-hop costs without changing behavior.

Code corrected:

A cache has been added to the router to store pool addresses and reduce redundant `poolFor()` lookups, optimizing gas usage during multi-hop swaps.

6.19 StableTokenV3.initializeV3 Does Not Update EIP712 Version

Informational | Version 1 | Code Corrected

CS-MECO-022

In `StableTokenV3.sol`, the `initializeV3` function, called when upgrading from `StableTokenV2`, does not update the EIP712 version used in `ERC20Permit`. This allows for permits created for the previous implementation to remain valid.

Code corrected:

The `initializeV3` function has been updated to set the EIP712 version to 3.

6.20 TradingLimitsV2 Accepts Unknown Flag Bits

Informational | Version 1 | Code Corrected

CS-MECO-023

In TradingLimitsV2, `validate()` checks that `limit0` and `limit1` are nonzero when their respective bits are set, and that `limit1` exceeds `limit0` when both limits are enabled.

However, the function does not reject flag values containing bits outside `L0` and `L1`. As a result, callers can pass `flags > 3` without reverting, even though such values have no defined meaning and could mask configuration mistakes.

Code corrected:

The TradingLimitsV2 implementation was refactored to remove the flag-based validation system. Limits are now derived directly from the configuration, eliminating the possibility of passing invalid flag values.

6.21 Unindexed Address Parameters in FPMM Configuration Events

Informational Version 1 Code Corrected

CS-MECO-024

Several FPMM configuration events emit address parameters without indexing them:

```
event ReferenceRateFeedIDUpdated(address oldRateFeedID, address newRateFeedID);
event OracleAdapterUpdated(address oldOracleAdapter, address newOracleAdapter);
event ProtocolFeeRecipientUpdated(address oldRecipient, address newRecipient);
```

Indexing would let indexers and monitoring systems query configuration changes more efficiently.

Code corrected:

The address parameters in FPMM configuration events have been indexed.

6.22 VirtualPoolFactory Returns Zero for Undeployed Pools

Informational Version 1 Code Corrected

CS-MECO-025

In `VirtualPoolFactory`, `getOrPrecomputeProxyAddress` simply returns the entry in `_pools[token0][token1]`.

When no virtual pool has been deployed for the pair, this mapping entry is zero, so the function reports `address(0)` instead of a deterministic precomputed address (or an explicit error). Callers that treat `getOrPrecomputeProxyAddress` like the FPMM factory's version (which always returns a usable or predictable address) may misinterpret the zero value and attempt to interact with a non-existent pool.

Code corrected:

The VirtualPoolFactory now returns the precomputed address for undeployed pools instead of returning zero.

6.23 getAmountsOut Returns Zero on Invalid Swap

Informational Version 1 Code Corrected

CS-MECO-026

The function `Router.getAmountsOut` calculates the amount of tokens returned from multiple swaps. However, when a user provides a route without a pool deployed by the factory, the function will return zero tokens instead of reverting.

```
amounts[0] = amountIn;
uint256 _length = routes.length;
for (uint256 i = 0; i < _length; i++) {
    address factory = routes[i].factory == address(0) ? defaultFactory : routes[i].factory;
    address pool = poolFor(routes[i].from, routes[i].to, routes[i].stable);
    if (IPoolFactory(factory).isPool(pool)) {
        amounts[i + 1] = IPool(pool).getAmountOut(amounts[i], routes[i].from);
    }
}
```

As a result, any other application calling `getAmountsOut` could continue with zero amount out when provided an invalid route. Provided an invalid route, the function will continue execution and only revert later when a call to the `swap` function on a non-existent pool address reverts due to Solidity's extcodesize check.

Code corrected:

The Router now reverts when an invalid pool is passed to `getAmountsOut`.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Initializers Must Be Called on Deployment

Informational Version 1 Risk Accepted

CS-MECO-017

It must be ensured that all initializers are called immediately during deployment or upgrade of upgradable contracts.

In particular, when upgrading to StableTokenV3, the `initializeV3` function must be called by the owner immediately, otherwise the permissionless `initialize` function will be callable by anyone to take over the contract.

Risk accepted:

Mento acknowledged the risk and provided the following statement:

```
[Initialization] will be correctly handled during deployment.
```

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 FPMM Profitability Critically Depends on Fees

Note Version 1

The profitability of an FPMM depends critically on the fees it charges for swaps and the incentive it gives for rebalances. In particular:

1. The swap fee must be high enough that the oracle price rarely lags the true market price by more than the swap fee. Whenever the deviation is greater than the fee, an arbitrageur can profit by swapping against the FPMM up to the TradingLimit.
2. The swap fee must be high enough compared to the rebalance incentive. If it is not, it can be profitable to intentionally make the FPMM imbalanced, and then rebalance it to earn the rebalance incentive repeatedly. The redemption side of the CDPLiquidityStrategy is more susceptible to this, as redemptions pay the rebalance incentive to a single trove owner. On the stability pool side, the rebalance incentive is shared among all stability pool depositors. This makes the strategy less profitable, as the attacker will pay the full trading fee to imbalance the pool, but only receive part of the incentive, unless they are the only Stability Pool depositor.

The profitability also depends on the limits set in TradingLimitsV2. Lower limits reduce the potential profit from a single arbitrage, limiting the downside risk. The rebalance cooldown also limits the frequency of rebalances, reducing the potential profit from repeated imbalancing and rebalancing.

8.2 Forex Market Low-Liquidity Considerations

Note Version 1

Liquidity can be non-uniform throughout the trading day. For example, the first 2 hours of the Sydney session tend to have lower liquidity than the rest of the session, when Tokyo joins.

There are regional holidays that will affect certain trading sessions of certain days. For example, the New York session is closed on US public holidays, such as Good Friday, Memorial Day, and others. On these days, the liquidity during the New York session will be significantly reduced, though other sessions on the same day may remain open. Regional holidays are not taken into account in the MarketHoursBreaker, although they may cause liquidity to drop to very low levels during affected sessions.

For Christmas and New Year's Eve, liquidity tends to drop significantly during the last few hours of trading before the holiday closure, as some markets close early. For example, the U.S. Stock market closes at 18:00 UTC on Christmas Eve, which can lead to reduced liquidity during the remainder of the forex trading day (18:00 UTC to 22:00 UTC).

For each currency used, it should be carefully evaluated how its liquidity is affected by different times. Especially for less commonly traded currencies, liquidity can be very low during certain sessions or times of the day.

If the FPMM offers trades at forex oracle prices during times of low liquidity, there is a greater risk that those trades happen at unfavorable prices.

8.3 Liquidity Strategies Should Not Have Arbitrary Calls

Note Version 1

Currently, there are only two liquidity strategies. They do not implement arbitrary calls. Any future liquidity strategies should also avoid implementing arbitrary calls, as this could lead to reentrancy issues. In the current audit, it was assumed there cannot be arbitrary calls in liquidity strategies. In case this is a requirement in the future, the security implications on the rest of the codebase must be carefully considered.

8.4 Oracle Staleness Threshold

Note Version 1

The oracle staleness threshold for FX prices must be smaller than a weekend, to avoid using the market close price when the market opens again after the weekend.

It should also be larger than the heartbeat time of the Chainlink oracle, as it can take some time for the price to be available on-chain after the heartbeat time has passed.