Security Audit Report

Thema Protocol

v1.0

December 27, 2025

Table of Contents

License

THIS WORK IS LICENSED UNDER A CREATIVE COMMONS ATTRIBUTION-NODERIVATIVES 4.0 INTERNATIONAL LICENSE.

Disclaimer

THE CONTENT OF THIS AUDIT REPORT IS PROVIDED “AS IS”, WITHOUT REPRESENTATIONS AND WARRANTIES OF ANY KIND.

THE AUTHOR AND HIS EMPLOYER DISCLAIM ANY LIABILITY FOR DAMAGE ARISING OUT OF, OR IN CONNECTION WITH, THIS AUDIT REPORT.

THIS AUDIT REPORT WAS PREPARED EXCLUSIVELY FOR AND IN THE INTEREST OF THE CLIENT AND SHALL NOT CONSTRUE ANY LEGAL RELATIONSHIP TOWARDS THIRD PARTIES. IN PARTICULAR, THE AUTHOR AND HIS EMPLOYER UNDERTAKE NO LIABILITY OR RESPONSIBILITY TOWARDS THIRD PARTIES AND PROVIDE NO WARRANTIES REGARDING THE FACTUAL ACCURACY OR COMPLETENESS OF THE AUDIT REPORT.

FOR THE AVOIDANCE OF DOUBT, NOTHING CONTAINED IN THIS AUDIT REPORT SHALL BE CONSTRUED TO IMPOSE ADDITIONAL OBLIGATIONS ON COMPANY, INCLUDING WITHOUT LIMITATION WARRANTIES OR LIABILITIES.

COPYRIGHT OF THIS REPORT REMAINS WITH THE AUTHOR.

This audit has been performed by

Oak Security GmbH

https://oaksecurity.io/
info@oaksecurity.io

Introduction

Purpose of This Report

Oak Security GmbH has been engaged by Thema Labs Inc. to perform a security audit of Smart contract audit for Thema protocol.

The objectives of the audit are as follows:

1. Determine the correct functioning of the protocol, in accordance with the project specification.
2. Determine possible vulnerabilities, which could be exploited by an attacker.
3. Determine smart contract bugs, which might lead to unexpected behavior.
4. Analyze whether best practices have been applied during development.
5. Make recommendations to improve code safety and readability.

This report represents a summary of the findings.

As with any code audit, there is a limit to which vulnerabilities can be found, and unexpected execution paths may still be possible. The author of this report does not guarantee complete coverage (see disclaimer).

Codebase Submitted for the Audit

The audit has been performed on the following target:

Repository	https://github.com/thema-financial/thema-zero
Commit	`618d4a09ec826b4681b9f02ded3614085cce0085`
Scope	All contracts in `packages/core/contracts` were in scope.
Fixes verified at commit	`f3d763ea10e1bb998b286f1d1c8f7f026a29da51` Note that only fixes to the issues described in this report have been reviewed at this commit. Any further changes such as additional features have not been reviewed.

Methodology

The audit has been performed in the following steps:

1. Gaining an understanding of the code base's intended purpose by reading the available documentation.
2. Automated source code and dependency analysis.
3. Manual line-by-line analysis of the source code for security vulnerabilities and use of best practice guidelines, including but not limited to:
    a. Race condition analysis
    b. Under-/overflow issues
    c. Key management vulnerabilities
4. Report preparation

Functionality Overview

Thema is a DeFi protocol for issuing and managing asset-backed tokens (OMTs) that can be minted, redeemed, and transferred across chains.

On-chain contracts implement a token with atomic mint/redeem flows against configured asset baskets, NAV tracked via oracle inputs, and modular interfaces for fee models, oracle providers, and token set composition.

How to Read This Report

This report classifies the issues found into the following severity categories:

Severity	Description
Critical	A serious and exploitable vulnerability that can lead to loss of funds, unrecoverable locked funds, or catastrophic denial of service.
Major	A vulnerability or bug that can affect the correct functioning of the system, lead to incorrect states or denial of service.
Minor	A violation of common best practices or incorrect usage of primitives, which may not currently have a major impact on security, but may do so in the future or introduce inefficiencies.
Informational	Comments and recommendations of design decisions or potential optimizations, that are not relevant to security. Their application may improve aspects, such as user experience or readability, but is not strictly necessary. This category may also include opinionated recommendations that the project team might not share.

The status of an issue can be one of the following: **Pending, Acknowledged**, **Partially Resolved**, or **Resolved**.

Note that audits are an important step to improving the security of smart contracts and can find many issues. However, auditing complex codebases has its limits and a remaining risk is present (see disclaimer).

Users of the system should exercise caution. In order to help with the evaluation of the remaining risk, we provide a measure of the following key indicators: **code complexity**, **code readability**, **level of documentation**, and **test coverage**. We include a table with these criteria below.

Note that high complexity or low test coverage does not necessarily equate to a higher risk, although certain bugs are more easily detected in unit testing than in a security audit and vice versa.

Code Quality Criteria

The auditor team assesses the codebase's code quality criteria as follows:

Criteria	Status	Comment
Code complexity	**Medium**	-
Code readability and clarity	**Medium-High**	-
Level of documentation	**High**	The codebase is well documented, and the client hosted a session to explain the protocol in detail.
Test coverage	**Medium**	Test coverage reported by `forge coverage` is `68.21%`

Summary of Findings

No	Description	Severity	Status
1	Spot price from `slot0` is susceptible to short-term manipulation and is not recommended as a standalone production oracle	Critical	Resolved
2	Chainlink `answeredInRound` field is deprecated and is not recommended for staleness checks	Major	Resolved
3	Hardcoded one-day staleness threshold does not account for different Chainlink feed heartbeat intervals	Major	Resolved
4	Hardcoded 20-minute swap deadline enables TX to stall and MEV	Major	Resolved
5	Missing compound invocation when minting and redeeming tokens enables fee dilution of existing holders	Major	Partially Resolved
6	Spot-only oracle configuration in `setTwapConfig` exposes protocol to single-block price manipulation	Major	Resolved
7	Fabricated timestamp values in `latestRoundData` neutralize oracle freshness checks	Major	Resolved
8	Round ID artificially incremented on configuration changes breaks Chainlink compatibility	Major	Resolved
9	Permissionless `updatePrice` breaks Chainlink semantics	Major	Resolved
10	Inconsistent rounding between compound and transfer checks can brick the fee distribution	Major	Resolved
11	Double rounding down in `compound` inflates token quantities using the protocol backing	Major	Resolved
12	Lack of duplicate token handling in `updateTokens` function enables an inconsistent registry state	Minor	Resolved
13	Rounding discrepancy in `analyzeFees` function causes inconsistent fee breakdown reporting	Minor	Resolved
14	Fabricated historical data in `getRoundData`	Minor	Resolved

	compromises price feed integrity		
15	Missing fee cap validation in `FlatFeeModel` constructor enables configuration-induced Denial of Service	Minor	Resolved
16	Centralized price control in `setManualPrice` function introduces oracle trust risk	Minor	Acknowledged
17	Missing input validation in `setFeeReceivers` allows zero addresses and duplicate entries	Minor	Resolved
18	Immutable fee parameters prevent adjustment and lack sanity checks	Minor	Resolved
19	`SwapAndMint` reverts when duplicate token pools are used in the swap path	Minor	Resolved
20	`PrepareMintTx` lacks a minimum OMT mint amount check	Minor	Acknowledged
21	Price scaling uses hardcoded target decimals instead of actual Chainlink feed decimals	Informational	Resolved
22	Redundant zero initialization wastes gas	Informational	Acknowledged
23	Compound can't be paused	Informational	Resolved
24	Token decimals should be read from the contract instead of defaulting to 18	Informational	Resolved
25	Fee distribution uses a push pattern vulnerable to blacklisted receivers	Informational	Acknowledged
26	Zero-amount fee transfers can cause a revert with tokens that reject zero value transfers	Informational	Resolved

Detailed Findings

1. Spot price from `slot0` is susceptible to short-term manipulation and is not recommended as a standalone production oracle

Severity: Critical

In `packages/core/contracts/oracle/UniswapV3PriceFeed.sol:61-80`, within `latestRoundData` and in `packages/core/contracts/oracle/OMTPriceFeed.sol` within `_getTwapPrice`, both contracts read instantaneous spot prices directly from Uniswap V3's `slot0` function.

The `UniswapV3PriceFeed` calls `POOL.slot0` to retrieve `sqrtPriceX96` and immediately converts it to a price. The `OMTPriceFeed` function is named `_getTwapPrice` but directly calls `uniswapV3Pool.slot0`.

While spot pricing is responsive, it can be influenced over short time horizons by sufficiently large swaps executed within the same transaction or block, particularly when flash liquidity is available. Under such conditions, the pool price can temporarily diverge from broader market levels during protocol interactions that rely on the oracle.

The immediate exploitable impacts are fee manipulation and downstream integrator risk.

The `ArbitrageFeeModel` contract calculates mint and redeem fees based on the gap between the OMT's NAV and its market price as reported by the oracle. An attacker can manipulate the OMT spot price read from `slot0` to artificially narrow or widen this gap. By pushing the spot price toward NAV, the attacker pays minimum fees when minting or redeeming. Conversely, by pushing the spot price away from NAV, an attacker can grief legitimate users into paying maximum fees. Additionally, any external protocol integrating `OMTPriceFeed` or `UniswapV3PriceFeed` for collateral valuation, liquidation thresholds, or trading decisions becomes vulnerable to sandwich attacks and flash-loan price manipulation.

A concrete attack scenario for fee evasion works as follows:

- Attacker flash loans large amounts of the paired token (for example USDC)
- Executes massive swap in Uniswap V3 pool to push OMT spot price exactly to NAV
- Calls `OMT.mintFromTokens`, which queries `ArbitrageFeeModel.calculateMintFeeRate`
- Fee model reads manipulated price showing zero deviation from NAV, returns `MIN_FEE`
- Attacker mints OMT, paying only the minimum fee instead of the appropriate market-based fee

- Swaps back to restore pool price and repays flash loan

The attacker profits by avoiding fees that should have been charged based on true market conditions. Even with large liquidity pools, flash loans provide sufficient capital to move prices significantly.

Recommendation

We recommend replacing direct `slot0` spot price reads with time-weighted average price (TWAP) calculations. Uniswap V3 provides the `observe` function specifically for this purpose, enabling the computation of manipulation-resistant prices using historical tick accumulators.

Status: Resolved

The client addressed this issue through a comprehensive reengineering of the oracle design, aligning the implementation more closely with established oracle robustness and interface expectations.

2. Chainlink `answeredInRound` field is deprecated and is not recommended for staleness checks

Severity: Major

In `packages/core/contracts/oracle/ChainlinkOracle.sol`, the `getPrice` function uses the `answeredInRound` field from Chainlink's `latestRoundData` to validate price freshness by checking if `answeredInRound` is less than `roundId`.

The `answeredInRound` field was deprecated by Chainlink because it does not reliably indicate stale data across all aggregator types, particularly with newer Chainlink architectures where the relationship between roundId and answeredInRound may not follow the expected pattern.

Relying on this deprecated field means the staleness check may fail to detect actual stale prices, allowing the oracle to return outdated price data that could be exploited in mint and redeem operations. An attacker could potentially trade against stale prices to extract value from the OMT contract.

Recommendation

We recommend removing the `answeredInRound` comparison and relying solely on the `updatedAt` timestamp check against `block.timestamp`.

Status: Resolved

The client addressed this issue through a comprehensive reengineering of the oracle design, aligning the implementation more closely with established oracle robustness and interface expectations.

3. Hardcoded one-day staleness threshold does not account for different Chainlink feed heartbeat intervals

Severity: Major

In `packages/core/contracts/oracle/ChainlinkOracle.sol`, the `getPrice` function uses a hardcoded 1-day threshold to determine if a Chainlink price is stale by checking `block.timestamp - updatedAt > 1 days`.

Different Chainlink price feeds have different heartbeat intervals that determine how frequently they update.

For example, `ETH/USD` on mainnet updates every hour, while some less liquid pairs may update every 24 hours, and highly volatile assets might update every few minutes. A hardcoded one-day threshold is too permissive for feeds with short heartbeats, like one hour, meaning the oracle could accept prices that are 23 hours stale when they should be rejected after just a few hours.

Conversely, for feeds with longer heartbeats, the one-day threshold could incorrectly mark valid prices as stale. This mismatch allows the OMT contract to operate with inappropriately stale prices during mints and redeems, potentially enabling arbitrage opportunities or incorrect NAV calculations.

Recommendation

We recommend replacing the hardcoded staleness threshold with a per-token configurable heartbeat mapping.

Add a `mapping(address => uint256) public heartbeatThresholds` to store the maximum allowed staleness for each token's feed, and add an owner-only function `setHeartbeatThreshold(address token, uint256 threshold)` to configure it.

Then modify the staleness check to `block.timestamp - updatedAt > heartbeatThresholds[_token]`.

Each threshold should be set to match the documented heartbeat of the corresponding Chainlink feed plus a small safety buffer.

Status: Resolved

4. Hardcoded 20-minute swap deadline enables TX to stall and MEV

Severity: Major

In `packages/core/contracts/intent/SwapAndMint.sol`, the `swapAndMint` loop hardcodes a 20-minute validity window by passing `block.timestamp + 1200` to `UNISWAP_ROUTER.swapExactTokensForTokens`.

This allows a submitted transaction to remain valid in the mempool for a long time and be mined under significantly different prices than when prepared, increasing exposure to sandwich/backrun attacks and stale execution.

Because the contract is the caller and there is no oracle-anchored slippage guard or caller-controlled expiry, execution can occur under adverse market conditions.

Recommendation

We recommend accepting a caller-provided `deadline` (bounded by a sensible maximum) instead of hardcoding `1200`, and adding price protection beyond user-supplied `swapAmountsOut[i]` (e.g., validate expected outputs against an oracle/TWAP, or use tighter per-hop slippage limits).

Status: Resolved

5. Missing compound invocation when minting and redeeming tokens enables fee dilution of existing holders

Severity: Major

In `packages/core/contracts/OMT.sol`, the `mintFromTokens` and `redeemToTokens` functions execute operations without first invoking `compound`.

Both functions rely on the current per-OMT quantity values, which are used by `calculateNav`, but these values are not updated to account for accumulated protocol fees before the operations are performed.

As a result, surplus fees held by the contract are excluded from the NAV calculation, allowing new users to mint OMT at an un-compounded value while immediately gaining proportional ownership of the entire vault, including the preexisting surplus.

When `compound` is later invoked, the accumulated surplus is distributed across all token holders based on the updated total supply, effectively diluting the economic value of existing holders.

Similarly, redemptions processed before compounding can allow users to exit at an undervalued NAV, leaving residual value behind for others and creating inconsistent accounting between holders.

Recommendation

We recommend invoking `compound` at the beginning of both `mintFromTokens` and `redeemToTokens` to ensure that all accumulated protocol fees are realized and reflected in the per-OMT NAV before any minting or redemption occurs.

Status: Partially Resolved

6. Spot-only oracle configuration in `setTwapConfig` exposes protocol to single-block price manipulation

Severity: Major

In `packages/core/contracts/oracle/OMTPriceFeed.sol:104-114`, the `setTwapConfig` function enforces a spot-only mode by requiring the `twapInterval` parameter to be zero.

This design hardcodes the oracle to use instantaneous Uniswap V3 prices retrieved directly from `slot0`, without any form of time-weighted averaging or observation-based smoothing. As a result, the price feed becomes highly susceptible to transient, single-block manipulation.

A malicious actor can momentarily distort the pool price using a large swap or flash loan, trigger protocol functions that read from the manipulated oracle within the same block, and then revert or unwind the trade to restore the original price.

Because the oracle disallows nonzero TWAP intervals, there is no built-in mechanism to filter out such short-term distortions. This configuration introduces a structural market-manipulation weakness that undermines the reliability of the feed and could be exploited to extract value or influence protocol behavior based on falsified spot data.

Recommendation

We recommend allowing configurable, nonzero TWAP intervals and integrating Uniswap V3's built-in observation-based pricing to compute time-weighted average prices over several blocks.

Status: Resolved

7. Fabricated timestamp values in `latestRoundData` neutralize oracle freshness checks

Severity: Major

In `packages/core/contracts/oracle/OMTPriceFeed.sol:71-77`, the `latestRoundData` function incorrectly assigns both the `startedAt` and `updatedAt` fields to the current `block.timestamp` on every call, regardless of when the underlying price was last updated.

This implementation violates Chainlink's freshness semantics by effectively making the feed appear perpetually current.

Downstream consumers, such as `ChainlinkOracle.getPrice`, use these timestamps to perform staleness checks based on the age of the data. Because `updatedAt` is always equal to the current block time and `answeredInRound` always matches `roundId`, these freshness guards are rendered ineffective.

As a result, stale, frozen, or manually set prices can be misinterpreted as valid and up to date.

The issue is particularly severe in `MANUAL` mode, where a static price may never refresh yet still passes freshness validation, and in `NAV` mode, where the true update time of underlying component feeds is not reflected.

This behavior undermines the reliability of time-based safeguards and allows protocol operations to proceed using outdated or manipulated data.

Recommendation

We recommend updating the oracle logic to record and return accurate timestamps corresponding to the actual last update time of the reported price.

Status: Resolved

8. Round ID artificially incremented on configuration changes breaks Chainlink compatibility

Severity: Major

In `packages/core/contracts/oracle/OMTPriceFeed.sol:91-95`, the `setPriceMode` function increments `latestRoundId` whenever the price mode changes, even though no actual oracle data update has occurred. The same flawed pattern appears in `setManualPrice` and `setTwapConfig`.

According to Chainlink documentation, round IDs represent periodic data updates from oracles and should only increment when new price observations arrive. The `OMTPriceFeed` contract misuses round IDs as a generic version counter for any configuration change. This fundamentally breaks the Chainlink aggregator model, where round IDs allow contracts to query historical oracle data from specific update rounds.

When an integrator calls `getRoundData(5)` expecting the price from the fifth oracle update, they will instead get whatever the current price happens to be with no indication that round five was actually a mode configuration change rather than a legitimate price observation.

The contract provides no way to distinguish between rounds that contain real price data versus rounds that were just configuration updates.

Additionally, Chainlink explicitly states that round ID increases may not be monotonic because aggregators can be upgraded, but this contract guarantees strict monotonic increases which contradicts expected behavior and may cause integrators to make incorrect assumptions about data availability.

Recommendation

We recommend removing all `latestRoundId++` statements from configuration functions like `setPriceMode` and `setTwapConfig`. Only increment the round ID when actual price data changes in a meaningful way that represents a true oracle update. For manual mode, increment only when `setManualPrice` is called with a price different from the current value.

Status: Resolved

9. Permissionless `updatePrice` breaks Chainlink semantics

Severity: Major

In `packages/core/contracts/oracle/UniswapV3PriceFeed.sol`, the `updatePrice` function is external and unprotected, allowing anyone to increment the round ID and update the timestamp without any actual price observation or state change.

Since the contract reads spot price directly from the Uniswap pool's `slot0`, the price itself is never stored or updated by this function.

Any caller can arbitrarily bump `roundId` and `lastUpdateTime` as many times as they want in a single block or across multiple blocks. This breaks the Chainlink aggregator semantics, where round IDs should represent distinct oracle updates with meaningful new data.

Recommendation

We recommend removing the `updatePrice` function entirely. If round tracking is needed for compatibility, either remove round ID management completely and return a constant value, or implement a time-based automatic increment in `latestRoundData` itself.

Status: Resolved

10. Inconsistent rounding between compound and transfer checks can brick the fee distribution

Severity: Major

In the `packages/core/contracts/OMT.sol` contract, the `compound` and `_safeTransferFromContract` functions apply inconsistent rounding when converting expected balances from 18-decimal precision to each token’s native decimals.

This discrepancy can cause the contract to appear correctly collateralized during compounding while later failing invariant checks during fee transfers, resulting in transaction reverts.

Within `compound`, the expected balance is calculated using `convertToTokenDecimals`, which rounds down when scaling from 18 decimals to a lower decimal token (e.g., 6-decimal USDC).

In contrast, `_safeTransferFromContract` computes the required balance using `convertToTokenDecimalsRoundUp`, which rounds up.

This mismatch causes an off-by-one unit error in cases where token quantities include fractional precision beyond the token's native decimals.

For example, with a 6-decimal token like USDC and `quantity = 1.0000005e18`, `compound` passes validation (truncating to `1.000000`), but `_safeTransferFromContract` later rounds up to `1.000001`, making the invariant check `require(currentBalance >= requiredBalance + amount)` fail, even for zero amount. This results in a full denial-of-service for fee-distribution operations such as `mintFromTokens` and `redeemToTokens`.

The bug is triggered only when the OMT contains ow-decimal tokens (6 decimals like USDC or USDT) and fractional quantities that are truncated during conversion.

Recommendation

We recommend using consistent rounding in both places. The safer approach is to always round UP in both checks to ensure backing is never violated, even by dust amounts.

Status: Resolved

11. Double rounding down in `compound` inflates token quantities using the protocol backing

Severity: Major

In `packages/core/contracts/OMT.sol`, the `compound` function performs two sequential rounding-down operations when calculating the expected token balance, causing it to systematically understate the required backing. This leads the function to misclassify legitimate backing tokens as "fees," artificially inflating token quantities.

Over time, this creates a spiralling insolvency issue where redemptions require more tokens than the protocol holds.

The `compound` function calculates the expected backing balance to determine if there are excess fees to compound.

The error occurs due to sequential truncation:

- In line 385, `(quantity * totalSupply) / 1e18` performs integer division, truncating fractional wei.
- In line 388, `convertToTokenDecimals` again divides by a scaling factor for tokens with fewer than 18 decimals, truncating remaining precision.

These two rounding-down steps cause the computed expectedBalance to fall short by 1–2 `wei` per token, falsely classifying those missing units as available "fees." When compounded repeatedly, this leads to a gradual loss of backing.

The issue is amplified when combined with the rounding-up behavior in `_safeTransferFromContract`, which applies stricter invariants. Tokens with lower decimal precision, such as WBTC (8 decimals), are particularly affected due to higher rounding loss per operation.

Recommendation

We recommend rounding up the `expectedBalance`.

Status: Resolved

12. Lack of duplicate token handling in `updateTokens` function enables an inconsistent registry state

Severity: Minor

In `packages/core/contracts/OMT.sol:534-558`, the `updateTokens` function clears the existing `tokenList` and repopulates it with new token entries.

However, the implementation does not include any mechanism to detect or prevent duplicate token symbols in the `_tokens` array. If multiple tokens share the same symbol, the `tokenList` array will contain multiple entries for that symbol, while the `tokenBySymbol` mapping will retain only the most recently processed value.

This discrepancy creates an inconsistent registry state where lookups through the mapping may return unexpected results, and functions relying on a synchronized token list may behave incorrectly.

Recommendation

We recommend implementing a validation step within the `updateTokens` function to ensure that no duplicate token symbols are processed. The function should revert if a token with an existing symbol is encountered during the update process.

Status: Resolved

13. Rounding discrepancy in `analyzeFees` function causes inconsistent fee breakdown reporting

Severity: Minor

In `packages/core/contracts/interfaces/IThemaTx.sol:64-110`, the `analyzeFees` function calculates the total fee in basis points and then distributes it among different receiver roles based on their assigned weights.

Each receiver's share is computed independently using the formula `(totalFeeBps * weight) / 1e18`, which implicitly floors the result during integer division. This rounding behavior can cause the sum of the calculated per-role basis points to be less than the original `totalFeeBps`.

For example, if three roles each have weight `0.333333333333333333e18` (33.33%), the sum is `0.999999999999999999e18`, leaving a 1 wei gap, and each per-role calculation floors independently, compounding the loss.

While no funds are actually lost, since the corresponding transfer logic in `OMT.addFeeTransfers` assigns the rounding remainder to the final receiver, the `FeeAnalysis` output may appear inconsistent, reporting `returnsBps + lpFundBps + treasuryBps` as less than `totalFeeBps`.

This inconsistency can lead to confusion for integrators or monitoring systems that expect the reported fee distribution to sum precisely to the total fee percentage.

Recommendation

We recommend aligning the reporting logic in `analyzeFees` with the actual distribution mechanism used in `OMT.addFeeTransfers`.

Status: Resolved

14. Fabricated historical data in `getRoundData` compromises price feed integrity

Severity: Minor

In `packages/core/contracts/oracle/OMTPriceFeed.sol:62-70`, the `getRoundData` function returns fabricated historical price data instead of retrieving stored per-round values.

When called with any `_roundId` less than or equal to `latestRoundId`, the function generates a response using the current price from `_getPrice` and the current block timestamp, while simply echoing the input `_roundId` as both the `roundId` and `answeredInRound`.

This design results in all historical queries returning identical data to the most recent round, effectively erasing temporal distinctions between price samples. Consumers who depend on accurate historical round data, such as those enforcing time-based validations, detecting stale feeds, or calculating moving averages, receive misleading results.

This undermines the integrity of the oracle interface and may allow dependent systems to make decisions based on falsified or inconsistent historical information.

Recommendation

We recommend implementing persistent storage for per-round data, recording both price and timestamp at the time each round is produced.

The `getRoundData` function should then retrieve these stored values to accurately reflect the state of the feed at the specified round. If historical data is unavailable, the function should revert with an appropriate error message rather than returning fabricated information.

Status: Resolved

15. Missing fee cap validation in `FlatFeeModel` constructor enables configuration-induced Denial of Service

Severity: Minor

In `packages/core/contracts/fee/FlatFeeModel.sol:11-14`, the constructor assigns the provided `_fee` value directly to the `FEE` variable without enforcing an upper bound.

The contract treats this parameter as a percentage in 18-decimal fixed-point format, where `1e18` corresponds to 100%. Without validation, an administrator or deployer can set a value exceeding 1e18.

Downstream calculations that assume the fee is a valid percentage will compute incorrect results that exceed the available token amount.

Recommendation

We recommend enforcing input validation within the constructor and any subsequent fee-setting functions to ensure that the configured fee does not exceed 100%.

Status: Resolved

16. Centralized price control in `setManualPrice` function introduces oracle trust risk

Severity: Minor

In `packages/core/contracts/oracle/OMTPriceFeed.sol:97-103`, the `setManualPrice` function allows the contract owner to unilaterally set an arbitrary price value for the oracle.

This design introduces a centralization and trust risk, as the accuracy and integrity of the price feed depend entirely on a single privileged account.

A compromised owner, or even in non-malicious scenarios, false or accidental misconfiguration could result in severe pricing inaccuracies affecting user positions and protocol stability.

Recommendation

We recommend implementing decentralized or multi-signature governance for manual price updates, or restricting manual intervention to emergency fallback scenarios only.

Status: Acknowledged

17. Missing input validation in `setFeeReceivers` allows zero addresses and duplicate entries

Severity: Minor

In `packages/core/contracts/OMT.sol:430-451`, the `setFeeReceivers` function enables the contract owner to configure the list of fee receivers that determine how collected fees are distributed. While the function verifies that the total weight equals `1e18`, it performs no validation on individual receiver entries.

As a result, configurations containing duplicate receiver entries or zero addresses are accepted without restriction. This omission introduces potential inconsistencies and operational risks, as a zero address would cause fees to be effectively burned or trapped, and duplicated entries could lead to misallocation or skewed fee distribution.

Recommendation

We recommend enforcing stricter validation within the `setFeeReceivers` function.

Status: Resolved

18. Immutable fee parameters prevent adjustment and lack sanity checks

Severity: Minor

In `packages/core/contracts/fee/ArbitrageFeeModel.sol:23-26`, the constructor sets `MIN_FEE`, `MAX_FEE`, `ARBITRAGE_CAPTURE`, and `DEVIATION_MAX` as immutable values with no validation or ability to change them post-deployment. The contract inherits `Ownable` but provides no setter functions. This creates two issues.

First, there are no sanity checks during construction, allowing nonsensical configurations such as `MAX_FEE` exceeding 1e18 (one hundred percent fees or higher), `MIN_FEE` greater than `MAX_FEE`, or `ARBITRAGE_CAPTURE` set above 1e18. A deployment error could set fees that consume all user value or brick the fee model logic entirely.

Second, the parameters are permanently locked after deployment. If market conditions change or the initial calibration proves suboptimal, there is no way to adjust the fee curve without redeploying the entire contract and migrating all OMTs via `updateFeeModel`.

Similarly, in `packages/core/contracts/fee/FlatFeeModel.sol`, the `FEE` variable is declared `immutable` and set once in the constructor with no ability to change it afterward.

Recommendation

We recommend Add constructor validation to ensure `MIN_FEE <= MAX_FEE <= 1e18`, `ARBITRAGE_CAPTURE <= 1e18`, and `DEVIATION_MAX` is reasonable. Consider making these parameters mutable by removing `immutable` and adding `onlyOwner` setter functions with the same validation checks.

Status: Resolved

19. `SwapAndMint` reverts when duplicate token pools are used in the swap path

Severity: Minor

In `packages/core/contracts/intent/SwapAndMint.sol:93`, users can supply a custom swap path to acquire the tokens required for minting an OMT basket.

However, when multiple swaps target the same output token within a single transaction, subsequent approvals overwrite prior ones, leading to incomplete token acquisition and mint failure.

Consider a scenario where the OMT basket requires WETH and WBTC, and a user provides USDC as input with the following swap path configuration:

- USDC → WBTC
- USDC → WETH
- USDC → WBTC

In this case:

- The first USDC → WBTC swap successfully yields 2 WBTC (limited by pool liquidity).
- The second swap converts USDC → WETH, yielding 10 WETH.
- The third swap converts USDC → WBTC again, producing the remaining 3 WBTC.

However, the contract's `forceApprove` logic overwrites the previous token allowance when the same token appears multiple times in the swap sequence.

As a result, only the last WBTC approval (3 WBTC) remains valid, effectively discarding the first 2 WBTC swap outcomes. When the minting logic executes, it detects insufficient WBTC (only 3 instead of the required 5), causing the transaction to revert.

Recommendation

We recommend aggregating the total required token amounts prior to approval to prevent partial mints and reverts

Status: Resolved

20. `PrepareMintTx` lacks a minimum OMT mint amount check

Severity: Minor

In `packages/core/contracts/OMT.sol:172-174`, the `prepareMintTx` function does not enforce a minimum OMT mint amount, allowing users to mint tokens with insufficient or zero collateral backing.

The issue occurs because the calculation of each token's contribution uses integer division that truncates fractional values, as shown in the expression `(amountOmt * tokens[i].quantity) / 1e18`.

For high-priced, low-decimal tokens such as WBTC, the quantity per OMT is typically a very small number (for example, 1e14 units, roughly $10 worth). When a user mints a small amount of OMT, this truncation rounds down the fractional result to zero, producing no corresponding

WBTC allocation. This effectively allows users to mint small quantities of OMT without contributing proportional collateral.

In practice, a user could mint approximately 9.99e3 OMT tokens “for free” if the WBTC component rounds to zero. Although each instance of this exploit produces only a minor discrepancy, repeated use can accumulate over time, especially on lower-fee networks such as Layer 2s. As the price of WBTC or other high-value assets increases, the magnitude of this rounding error grows, compounding the protocol’s collateral deficit and leading to a gradual loss of backing.

Recommendation

We recommend implementing a minimum mint amount check (e.g., `minAmountOmt = 1e12`) to prevent zero-value mints.

Status: Acknowledged

21. Price scaling uses hardcoded target decimals instead of actual Chainlink feed decimals

Severity: Informational

In `packages/core/contracts/oracle/OMTPriceFeed.sol:182-190`, the `_toChainlinkScale` function accepts a `chainlinkDecimals` parameter, but this parameter is always passed as `TARGET_DECIMALS`, which is a constructor-set immutable value, not the actual decimals returned by real Chainlink price feeds.

The function is called from both `_getNavPrice` and `_getTwapPrice`, always passing `TARGET_DECIMALS` hardcoded at contract deployment. Real Chainlink feeds can have varying decimal precision, commonly eight decimals for USD pairs but sometimes eighteen decimals for ETH pairs or other values depending on the specific feed. When this contract is configured to work with actual Chainlink infrastructure or when integrating protocols query its decimals, they will receive `TARGET_DECIMALS` from the `decimals` function, but the actual price scaling may be incorrect if `TARGET_DECIMALS` does not match what downstream consumers expect.

More critically, the function name and parameter suggest it should scale to match Chainlink feed decimals dynamically, but it actually scales to whatever arbitrary value was set at construction. This creates an inconsistent mismatch between the contract's advertised decimal precision and its actual output.

Recommendation

We recommend removing the chainlinkDecimals parameter from `_toChainlinkScale` since it always receives `TARGET_DECIMALS` anyway. Rename the function to `_toTargetScale` to accurately reflect that it scales to the constructor-configured target rather than dynamically adapting to Chainlink standards.

Status: Resolved

22. Redundant zero initialization wastes gas

Severity: Informational

In `packages/core/contracts/fee/ArbitrageFeeModel.sol:37-38`, the `_calculateFeeRate` function explicitly initializes the `nav` variable to zero before immediately overwriting it with the result of `_oft.calculateNav`.

However, it serves no functional purpose, as `nav` is reassigned in the next statement. This redundant initialization introduces a minor but unnecessary gas cost during execution.

Recommendation

We recommend removing the line `uint256 nav = 0;` and changing the next line to `uint256 nav = _oft.calculateNav;` to declare and assign in one statement.

Status: Acknowledged

23. Compound can't be paused

Severity: Informational

In `packages/core/contracts/OMT.sol:367` the `compound` function can't be paused.

When the protocol is paused, it is usually due to volatility or potential insolvency issues. These situations are typically handled by adding a surplus amount of funds to the protocol's backing. However, this surplus can still be compounded for rewards, even though the intended purpose might be different. In most cases, if the protocol provides additional funds from the treasury, it is meant to restore exact backing ratios for the supported assets and the code defends due to the require/expected values.

However, if the intention is not to include the surplus in the reward mechanism, this behavior could inadvertently lead to treasury funds being distributed as rewards, effectively transferring tokens from the treasury to users.

Recommendation

We recommend adding the `whenNotPaused` modifier to the `compound` function to prevent it from being called while the protocol is paused.

Status: Resolved

24. Token decimals should be read from the contract instead of defaulting to 18

Severity: Informational

In `packages/core/contracts/OMT.sol:551`, the `updateTokens` function defaults a token's decimals field to 18 whenever it is provided as zero, instead of querying the actual token contract for its true decimal value. This behavior silently accepts misconfigured or incomplete admin input, leading to incorrect decimal scaling in all subsequent balance and conversion calculations.

Because many stablecoins and wrapped assets (such as USDC or WBTC) use fewer than 18 decimals, a mistaken default of 18 can mis-scale all calculations by several orders of magnitude.

For instance, if USDC (6 decimals) is incorrectly treated as an 18-decimal token, every conversion performed by `convertToTokenDecimals` and `convertToTokenDecimalsRoundUp` will be off by a factor of 10^{12}.

This disrupts the accuracy of minting, redemption, NAV computation, and fee distribution, and can render the protocol unusable until the token configuration is manually corrected.

Recommendation

We recommend querying decimals directly from the token contract instead of accepting them as admin input.

Status: Resolved

25. Fee distribution uses a push pattern vulnerable to blacklisted receivers

Severity: Informational

In `packages/core/contracts/OMT.sol:611-614`, the `_executeTx` function distributes fees using a push-based model, where the contract directly transfers fee amounts to receiver addresses during mint and redeem operations. The loop executes `safeTransferFromContract` for each fee transfer, requiring that all transfers succeed for the transaction to complete.

This design introduces a denial-of-service risk: if any fee receiver is blacklisted by a token that enforces transfer restrictions (such as USDC or USDT), or if a token in the basket implements a pause mechanism and is currently paused, the `safeTransfer` call will revert. Because all fee transfers occur in a single atomic transaction, the failure of one transfer prevents the entire mint or redeem operation from completing.

In effect, a single blacklisted or paused token, or a blacklisted receiver for any one token, can freeze all user-facing mint and redeem operations, locking liquidity and preventing entry or exit from the OMT system.

Recommendation

We recommend implementing a pull pattern where fees accumulate in the contract and receivers claim them separately.

Status: Acknowledged

26. Zero-amount fee transfers can cause a revert with tokens that reject zero value transfers

Severity: Informational

In `packages/core/contracts/OMT.sol:672-682`, the `addFeeTransfers` function always creates a fee transfer entry for the last receiver to handle rounding remainders, even when the computed remainder is exactly zero. The function constructs and appends a `ThemaTransfer` object for the last fee receiver using remainingAmount and remainingValue, regardless of whether these values are nonzero.

Later, during execution in `_executeTx`, all fee transfers, including zero-value ones, are processed via `_safeTransferFromContract`. Some ERC-20 tokens, particularly those with strict compliance logic (e.g., certain stablecoins or wrapped assets), revert on zero-value transfers. As a result, when all fee allocations divide evenly among receivers (a common, not edge-case scenario), the zero-value transfer for the remainder will revert, causing the entire mint or redeem transaction to fail.

This issue introduces a compatibility risk and potential denial of service: OMT baskets containing tokens that reject zero-value transfers will intermittently fail whenever fee splits are perfectly divisible by the receiver weights.

Recommendation

We recommend skipping the remainder transfer when the amount is zero.

Status: Resolved