FAILSAFE

Prepared for
Canonic

Prepared by
Failsafe

4 February 2026

Canonic

Smart Contract Audit Report

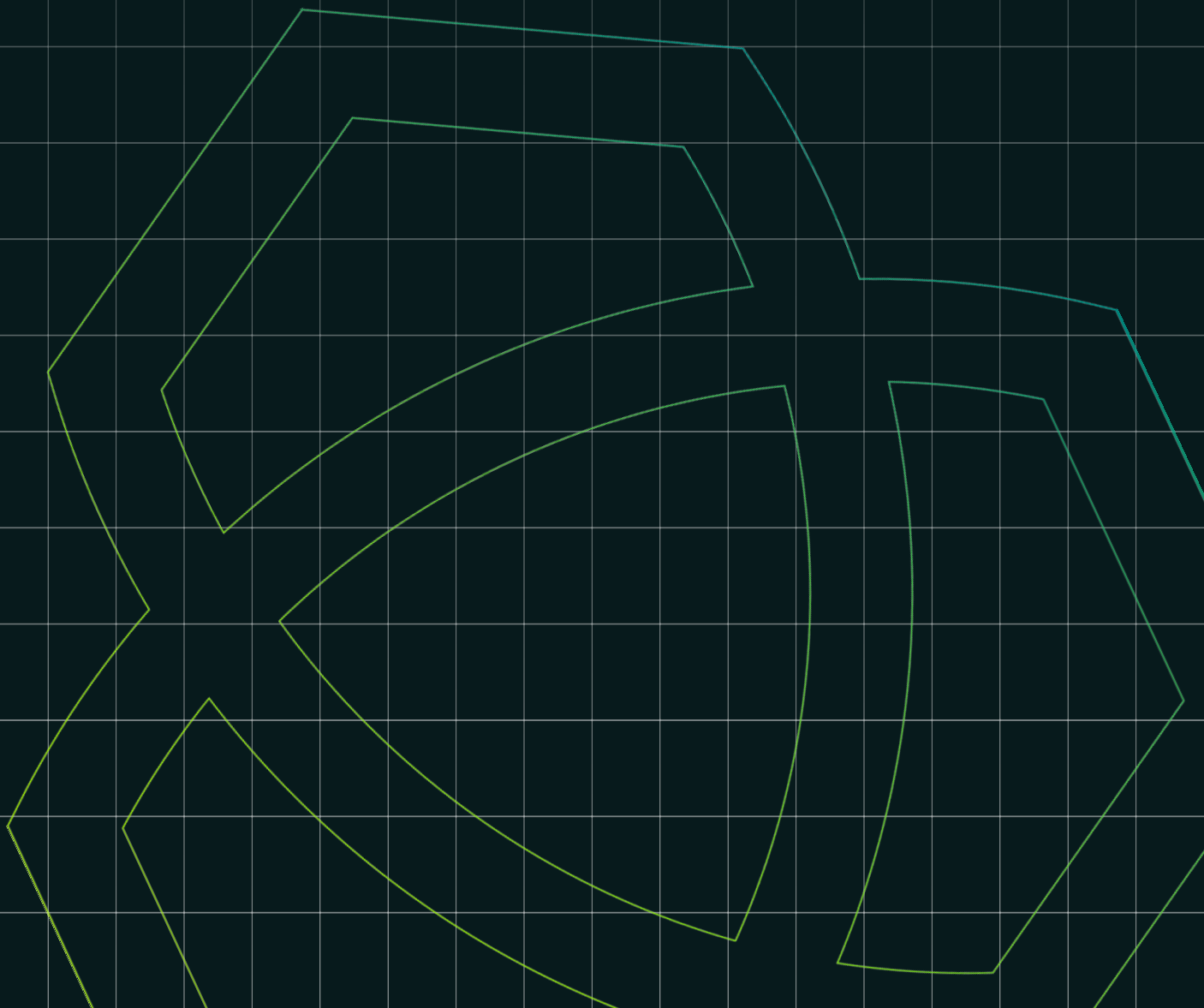

Table of Contents

Project Details

Project	Canonic
Website	https://github.com/0xnerdz/canonic-contracts
Blockchain	MegaEth
Audit Type	Smart Contract Audit Report
Initial Commit	09db16a07f87ccb6a0ecf16dab4b94f926f0489d
Final Commit	93d3b229d51f19a670fb778d4cc6c2690603b0a6
Timeline	18 January 2026 - 4 February 2026
	Final Report: 4 February 2026

Structure & Organization of The Security Report

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

- Open: The issue has been reported and is awaiting remediation from developer team.
- Acknowledged: The developer team has reviewed and accepted the issue but has decided not to fix it.
- Partially Resolved: Mitigations have been applied, yet some risks or gaps still remain.
- Resolved: The issue has been fully addressed and no further work is necessary.
- Closed: The issue is deemed no longer pertinent or actionable.

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

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

Executive Summary

FailSafe was engaged to perform a security audit of the Canonic smart contract suite, a Midpoint Anchored Order Book (MAOB) DeFi protocol deployed on MegaETH (L2) that enables on-chain limit order trading with prices anchored to discrete basis-point rungs around an oracle-provided midprice, accompanied by a liquidity provider vault (CLP) for automated market making.

Medium-severity issues include oracle staleness configuration where the code allows `maxStaleSeconds = 0` which would disable price freshness validation, token compatibility problems where fee-on-transfer tokens can be deposited into CLP but cannot be deployed to MAOB, economic vulnerabilities including withdrawal fee dilution due to incorrect calculation order, ineffective anti-spam protections for taker operations, an unconstrained asymmetric pause mechanism that lacks the time bounds and pre-defined states needed to ensure market integrity, and a sole LP fee bypass that was acknowledged as intended design. The low-severity finding addresses rebasing token insolvency risks as a design limitation requiring operational controls.

Summary of Findings

Severity	Total	Open	Acknowledged	Partially Resolved	Resolved	Closed
High	0	-	-	-	0	-
Medium	7	-	1	-	6	-
Low	1	-	1	-	-	-
Total	8	-	2	-	6	-

#	Findings	Severity	Status
1	Malicious Oracle Feed Configuration	Medium	Resolved
2	Oracle Staleness Configuration	Medium	Resolved
3	Fee On Transfer Incompatibility	Medium	Resolved
4	Dust Rung Griefing	Medium	Resolved
5	Sole Lp Fee Bypass	Medium	Acknowledged
6	Asymmetric Pause State	Medium	Resolved
7	Withdrawal Fee Dilution	Medium	Resolved
8	Rebasing Token Insolvency	Low	Acknowledged

Finding 1: Malicious Oracle Feed Configuration

Severity: MEDIUM
Status: RESOLVED

Description

This is a privileged attack vector - it requires a compromised `OracleAdapter` owner (e.g., leaked private key, phishing, insider threat).

If the owner key is compromised, the attacker can immediately replace a legitimate price feed with a malicious contract via `setFeedAdvanced()`. This malicious feed can return arbitrary prices, repricing the entire MAOB order book and allowing the attacker to drain all maker liquidity (including CLP vault funds) at deeply unfavorable rates.

Critical Issue: No timelock exists on these functions. Without a timelock, there is no window for the team to detect and respond to a malicious configuration change before it takes effect. A timelock (e.g., 24-48 hours) would provide time for monitoring systems to alert the team and allow intervention before funds are at risk.

Source

OracleAdapter.sol - setFeedAdvanced:

`contracts/OracleAdapter.sol:99-114` - No timelock, immediate effect:

```
function setFeedAdvanced(address orderBook, FeedConfig calldata cfg) external onlyOwner {
    require(orderBook != address(0), "orderBook zero");
    require(cfg.sourceA != address(0), "sourceA zero");  // Only checks non-zero!
    require(cfg.baseDecimals <= 18 && cfg.quoteDecimals <= 18, "decimals too large");
    require(cfg.feedDecimals <= 18, "decimals too large");
    require(cfg.feedType == SourceType.Custom, "custom only");
    // ... decimal validation ...
    feeds[orderBook] = cfg;  // Malicious oracle stored IMMEDIATELY
    feeds[orderBook].exists = true;
    emit FeedSet(orderBook, cfg.feedType, cfg.sourceA, cfg.dataId);
    // NO timelock delay - change is IMMEDIATE
}
```

The function accepts **any non-zero address** as `sourceA` without validating that it implements a legitimate price oracle or returns reasonable prices. A single compromised owner key can redirect all price feeds instantly.

OracleAdapter.sol - setCompositeFeed:

`contracts/OracleAdapter.sol:121-162` - Same vulnerability with composite feeds:

```
function setCompositeFeed(address orderBook, uint8 baseDecimals, uint8 quoteDecimals, CompositeLeg[]
    calldata legs)
    external
    onlyOwner
{
    // ... validation ...
    compositeCount[orderBook] = uint8(legs.length);
    for (uint256 i = 0; i < 3; i++) {
        if (i < legs.length) {
            CompositeLeg calldata leg = legs[i];
            require(leg.sourceA != address(0), "sourceA zero");
            compositeLegs[orderBook][i] = leg;  // Leg with invert=true stored IMMEDIATELY
        }
        // ...
    }
    emit CompositeFeedSet(orderBook, uint8(legs.length));
    // NO timelock delay
}
```

The `invert` flag on composite legs can flip price direction (2000 USD/ETH becomes 0.0005 USD/ETH), enabling the same attack vector.

MAOB.sol - Price Consumption:

`contracts/MAOB.sol:1582-1595` - MAOB unconditionally trusts oracle:

```
function _getMidPrice() internal view returns (uint256 price, uint256 precision) {
    uint48 updatedAt;
    (price, precision, updatedAt) = oracle.getPrice(address(this));
    if (price == 0 || precision == 0) revert MAOB__OraclePriceMissing();
    // NO deviation check, NO sanity bounds, NO historical comparison
    // Accepts ANY price the oracle returns
}
```

Testnet Configuration (Verified):

OracleAdapter

- Address: `0xe3d2310da37f1a75bdcc6bbdbe8219242509a948`
- Owner: `0x88055628be8b2a855910d7c7860e3914cc777f89`

Verification:

```
cast call 0xe3d2310da37f1a75bdcc6bbdbe8219242509a948 "owner()" --rpc-url https://carrot.megaeth.com/
    rpc
```

Impact

- **100% of protocol TVL** at risk - drains all maker liquidity on affected MAOB
- **CLP vault funds** directly exposed as they hold MAOB orders

- **Immediate effect** - no timelock means team has no time to respond or intervene
- Attack executes in **2 transactions** (set feed → drain) or atomically via Multicall3
- Estimated profit: **>99% of TVL** minus gas costs

Attack Scenario:

1. Attacker deploys malicious oracle returning `price = 1` (instead of ~2000e18)
2. Attacker calls `setFeedAdvanced()` with malicious oracle address
3. Attacker calls `buyBaseExactOut()` to purchase ALL ask liquidity for ~$0.0001
4. Alternatively, set price extremely high and drain bid liquidity via `sellBaseTargetIn()`

Remediation

1. **Implement 24-48 hour timelock** for `setFeedAdvanced()` and `setCompositeFeed()` (see Timelock Governance Appendix)
2. Add price deviation circuit breaker in MAOB - pause if oracle deviates >X% from stored reference
3. Emit detailed events on feed changes for off-chain monitoring

Resolution

Fixed in commit: `c6c64315910fb811c3a298b9064df915e6da4026`

The fix implements:

- A timelock controller (`OZTimelockController`) that wraps OpenZeppelin's `TimelockController`
- The MAOB owner is set to the timelock, requiring a delay for all admin actions
- A `pauseGuardian` role that can pause instantly (emergency response) but cannot unpause
- Only the timelock can unpause, preventing guardian abuse

This ensures that malicious oracle configuration changes require a timelock delay, giving the team time to detect and respond.

Finding 2: Oracle Staleness Configuration

Severity: MEDIUM
Status: RESOLVED

Description

The OracleAdapter contract allows `maxStaleSeconds` to be configured as `0`, which disables staleness validation entirely. While the current deployment correctly uses non-zero values (3s for base tokens, 10s for quote tokens), the code does not prevent administrators from setting `maxStaleSeconds = 0`, which would disable freshness checks. Chainlink recommends validating price freshness for production deployments.

Chainlink's Official Recommendation:

Chainlink's official documentation explicitly states that consumers **must** validate price freshness using the `updatedAt` return value from `latestRoundData()`:

> "Your application should track the `latestTimestamp` variable or use the `updatedAt` value from the `latestRoundData()` function to make sure that the latest answer is recent enough for your application to use it. If your application detects that the reported answer is not updated within the heartbeat or within time limits that you determine are acceptable for your application, pause operation or switch to an alternate operation mode while identifying the cause of the delay."

Documentation Reference: https://docs.chain.link/data-feeds#monitoring-data-feeds

The code allowing `maxStaleSeconds = 0` creates a risk that future deployments or configuration changes could inadvertently disable staleness validation.

Source

OracleAdapter.sol - Conditional Staleness Check:

`contracts/OracleAdapter.sol:218-220` - Staleness check is skipped when `maxStaleSeconds = 0`:

```
1  if (leg.maxStaleSeconds != 0) {  // Skipped when maxStaleSeconds == 0
2      require(block.timestamp <= uint256(legUpdatedAt) + leg.maxStaleSeconds, "leg stale");
3  }
```

The staleness check is conditional on `maxStaleSeconds != 0`. If configured as 0, arbitrarily old prices would be accepted without revert.

OracleAdapter.sol - setFeedAdvanced (No Input Validation):

`contracts/OracleAdapter.sol:99-114` - No validation that `maxStaleSeconds > 0`:

```
function setFeedAdvanced(address orderBook, FeedConfig calldata cfg) external onlyOwner {
    require(orderBook != address(0), "orderBook zero");
    require(cfg.sourceA != address(0), "sourceA zero");
    require(cfg.baseDecimals <= 18 && cfg.quoteDecimals <= 18, "decimals too large");
    require(cfg.feedDecimals <= 18, "decimals too large");
    require(cfg.feedType == SourceType.Custom, "custom only");
    // NO validation that cfg.maxStaleSeconds > 0
    feeds[orderBook] = cfg;
    feeds[orderBook].exists = true;
}
```

OracleAdapter.sol - setCompositeFeed (No Input Validation):

`contracts/OracleAdapter.sol:121-162` - Same issue with composite legs:

```
function setCompositeFeed(address orderBook, uint8 baseDecimals, uint8 quoteDecimals, CompositeLeg[]
    calldata legs)
    external
    onlyOwner
{
    // ...
    for (uint256 i = 0; i < 3; i++) {
        if (i < legs.length) {
            CompositeLeg calldata leg = legs[i];
            require(leg.sourceA != address(0), "sourceA zero");
            // NO validation that leg.maxStaleSeconds > 0
            compositeLegs[orderBook][i] = leg;
        }
    }
}
```

Impact

- If `maxStaleSeconds` were ever set to 0, oracle staleness protection would be completely disabled
- This would allow arbitrageurs to trade against stale prices, extracting value from liquidity providers
- Current deployments are protected (verified: 3s base, 10s quote), but code-level protection is missing
- Future deployments or configuration updates could inadvertently disable protection

Remediation

Add input validation to reject `maxStaleSeconds = 0`:

In setFeedAdvanced:

```
function setFeedAdvanced(address orderBook, FeedConfig calldata cfg) external onlyOwner {
    require(orderBook != address(0), "orderBook zero");
    require(cfg.sourceA != address(0), "sourceA zero");
    require(cfg.maxStaleSeconds != 0, "maxStaleSeconds zero");  // ADD THIS
    // ...
}
```

In setCompositeFeed:

```
for (uint256 i = 0; i < legs.length; i++) {
    CompositeLeg calldata leg = legs[i];
    require(leg.sourceA != address(0), "sourceA zero");
    require(leg.maxStaleSeconds != 0, "maxStaleSeconds zero");  // ADD THIS
    compositeLegs[orderBook][i] = leg;
}
```

In _readCustom and _readComposite (defense-in-depth):

```
// Replace conditional check with mandatory check
require(cfg.maxStaleSeconds != 0, "maxStaleSeconds zero");
require(block.timestamp <= uint256(updatedAt) + cfg.maxStaleSeconds, "feed stale");
```

This ensures staleness validation cannot be accidentally or intentionally disabled.

Resolution

Fixed in commit: `ebd30d67c2ab5ab160e48d1648db7571d0a7eb35`

The fix adds validation at both configuration time and runtime:

- `setFeedAdvanced()` now requires `cfg.maxStaleSeconds != 0`
- `setCompositeFeed()` now requires `leg.maxStaleSeconds != 0` for each leg
- `_readCustom()` and `_readComposite()` now enforce the check unconditionally (defense-in-depth)

Finding 3: Fee On Transfer Incompatibility

Severity: MEDIUM
Status: RESOLVED

Description

There is an inconsistent handling of Fee-on-Transfer (FOT) tokens between CLP and MAOB contracts that creates a functional trap. CLP's `depositDual()` uses a balance-delta pattern that correctly handles FOT tokens by measuring actual received amounts. However, MAOB's `addLiquidityBatch()` uses `safeTransferFromExact()` which reverts if the received amount differs from the requested amount. Users can successfully deposit FOT tokens into CLP, but when the operator attempts to deploy liquidity via `placeOrders()`, the transaction reverts at the MAOB level, rendering the vault dysfunctional for such tokens.

Source

CLP.sol - depositDual Function (Accepts FOT):

`contracts/CLP.sol:176-182` - Uses balance delta pattern:

```
uint256 baseBefore = baseToken.balanceOf(address(this));
baseToken.safeTransferFrom(msg.sender, address(this), baseDeposit);
baseDeposit = baseToken.balanceOf(address(this)) - baseBefore;  // Uses actual received

uint256 quoteBefore = quoteToken.balanceOf(address(this));
quoteToken.safeTransferFrom(msg.sender, address(this), quoteDeposit);
quoteDeposit = quoteToken.balanceOf(address(this)) - quoteBefore;
```

The deposit function measures the actual received amount after transfer, which correctly handles FOT tokens. Shares are minted based on the actual received amount.

MAOB.sol - addLiquidityBatch Function (Rejects FOT):

`contracts/MAOB.sol:366-371` - Uses strict transfer validation:

```
if (totalBaseIn != 0) {
    baseToken.safeTransferFromExact(msg.sender, address(this), totalBaseIn);  // Reverts if fee taken
}
if (totalQuoteIn != 0) {
    quoteToken.safeTransferFromExact(msg.sender, address(this), totalQuoteIn);
}
```

SafeTokenTransfer.sol - safeTransferFromExact Function:

`contracts/libraries/SafeTokenTransfer.sol:69-86` - Enforces exact amounts:

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

    uint256 balanceBefore = safeBalanceOf(token, to);
    token.safeTransferFrom(from, to, amount);
    uint256 balanceAfter = safeBalanceOf(token, to);
    unchecked {
        received = balanceAfter - balanceBefore;
    }
    if (received != amount) revert SafeTokenTransfer__TransferFromAmountMismatch();
}
```

The explicit check `received != amount` causes revert for any token where the transfer amount differs from the requested amount, including FOT tokens.

CLP.sol - placeOrders Function:

`contracts/CLP.sol:251-264` - Calls MAOB which will revert for FOT:

```
function placeOrders(
    MAOB.LiquidityOrder[] calldata withdrawOrders,
    MAOB.LiquidityOrder[] calldata balanceOrders
) external nonReentrant onlyRole(OPERATOR_ROLE) returns (uint256[] memory orderIds) {
    // ...
    uint256[] memory balanceOrderIds =
        balanceOrders.length == 0 ? new uint256[](0) : maob.addLiquidityBatch(balanceOrders);
    // ...
}
```

Impact

- **Denial of Service**: CLP vault cannot perform its primary function with FOT tokens
- **No Direct Fund Loss**: Users can still withdraw via `withdrawDual()` which uses standard `safeTransfer`
- **Lost Opportunity**: Deposited funds remain idle, no yield generation possible
- **Recovery Available**: `exitAndWithdrawAll()` works for emergency recovery
- Affects any vault deployed with FOT tokens such as PAXG, STA, or similar 1-2% fee tokens

Remediation

1. **Option A (Fail Fast)**: Modify `CLP.depositDual()` to use `safeTransferFromExact()`, rejecting FOT tokens at deposit time with a clear error message
2. **Option B (Full Support)**: Modify MAOB to support FOT tokens with balance-delta pattern and adjust order sizes based on actual received amounts

3. Add explicit documentation that FOT tokens are not supported
4. Implement token validation in CLP constructor that tests transfer behavior before deployment

POC

File: `Fee_On_Transfer_Incompatibility/FOTIncompatibilityPoC.t.sol`

Attack Scenario (Operational Failure):

1. Admin deploys CLP/MAOB with a fee-on-transfer token (e.g., PAXG with 0.02% fee)
2. Users deposit funds via `depositDual()` - succeeds, shares minted based on actual received amount
3. Operator calls `placeOrders()` to deploy idle liquidity to MAOB rungs
4. Transaction reverts at `MAOB.addLiquidityBatch()` with `SafeTokenTransfer__TransferFromAmountMismatch`
5. Vault is operationally dysfunctional - cannot generate yield
6. Users must manually withdraw; opportunity cost incurred

How to Run:

```
cd canonic-contracts-main-latest
forge test --match-contract FOTIncompatibilityTest -vvv
```

Test Functions:

- `test_CLPAcceptsFOTDeposit`: Shows CLP accepts FOT tokens on deposit
- `test_MAOBRejectsFOTOnPlaceOrders`: Shows MAOB rejects FOT tokens during deployment
- `test_FullOperationalFailureScenario`: End-to-end demonstration of the operational trap
- `test_WithdrawalWorksForRecovery`: Confirms recovery path via withdrawDual

Resolution

Fixed in commit: `dd3ec825377d447bbe113bce6636761d901d5d0d`

The fix implements Option A (Fail Fast):

- `CLP.depositDual()` now uses `SafeTokenTransfer.safeTransferFromExact()` instead of the balance-delta pattern

- If a FOT token is deposited, the transaction reverts with `SafeTokenTransfer__TransferFromAmountMismatch`
- This provides a clear error at deposit time rather than a confusing failure during `placeOrders()`

Finding 4: Dust Rung Griefing

Severity: MEDIUM
Status: RESOLVED

Description

If `minQuoteMaker` is configured too low, an attacker can spam liquidity across all 64 available rungs with minimal value. Takers attempting to fill orders will iterate through these dust rungs, incurring high gas costs due to expensive Fenwick tree update operations per rung (approximately 50-100k gas each). This can cause 10-20x gas amplification for takers and potential transaction failures during network congestion.

Why Takers Cannot Avoid Dust Rungs:

While a `maxRung` parameter exists on all taker functions, it does **not** solve this problem. The `maxRung` parameter is designed as a **price boundary control** (limiting how far from mid-price a taker will trade), not a **liquidity quality control**. Takers cannot:

- **Know which rungs contain dust**: There's no on-chain view function to check liquidity quality per rung before submitting a transaction
- **Skip specific rungs**: `maxRung` only sets an upper bound (e.g., "trade rungs 0-10"), it cannot say "skip rung 5"
- **Avoid dust at desirable rungs**: Attackers place dust at LOW rungs (near mid-price) - exactly where takers want to trade

The `maxRung` parameter would only help if dust were isolated to HIGH rungs (far from mid-price), but rational attackers place dust at the most-used rungs to maximize impact.

The gas cost scales with the **number of rungs processed**, not the **value of liquidity obtained**. This means:

- Normal: 3 rungs with 3.3 ETH each = 10 ETH, ~3 rung iterations
- Griefed: 100 rungs with 0.1 ETH each = 10 ETH, ~100 rung iterations (33x more gas)

The taker receives the same 10 ETH but pays dramatically more gas in the griefed scenario.

Asymmetric Attack Economics:

This attack is viable because the damage is **asymmetric**:

	Attacker	Victims (All Takers)
Cost	One-time gas to place dust orders	10-20x gas on EVERY trade
Duration	Single transaction	Until dust is cleared
Capital at risk	Minimal (dust amounts)	N/A

Example: If an attacker spends $1,000 in gas to place dust across all rungs, and 1,000 trades occur before the dust is cleared, each trade might incur ~$10 extra gas cost. The attacker spent $1,000 but caused $10,000+ in aggregate damage to protocol users.

Attacker Motivations:

1. **Competitor Sabotage**: Competing DEXs/AMMs directly benefit when MAOB becomes unusable due to gas costs. Users migrate to platforms with normal gas costs.

2. **Market Maker Warfare**: A dominant market maker could grief the protocol to drive away smaller competitors who cannot absorb the gas overhead, then remove the dust and capture the market.

3. **Short Position Exploitation**: If any metric is tied to Canonic's success (TVL, volume, governance token), an attacker could short that metric, grief the protocol to reduce usage, and profit from the short.

4. **Extortion**: Attack the protocol, then privately demand payment to stop. A legitimate threat since clearing dust costs money.

5. **Setup for Secondary Attack**: Grief protocol to thin liquidity, making price manipulation or other attacks easier.

This attack has low attribution risk since dust orders can appear indistinguishable from normal low-value trading activity.

Source

MAOB.sol - _fillAsks Function:

`contracts/MAOB.sol:1070-1114` - Loop iterates through all active rungs:

```
function _fillAsks(...) internal returns (uint256 totalQuoteUsed) {
    FillAskState memory state;
    state.remainingBase = baseAmount;
    state.mask = _activeMask(maxRung, activeAskRungs);
    uint256 precision = denom / baseScale;
```

```
6
7       while (state.mask != 0 && state.remainingBase > 0) {
8           uint16 rung = _lsbIndex(state.mask);
9           state.mask &= state.mask - 1;
10
11          RungTotals storage totals = askRungs[rung];
12          state.availableBase = totals.volume;
13          if (state.availableBase == 0) continue;
14
15          uint256 rungPrice = _rungPrice(midPrice, bpsRungs[rung], true);
16          state.priceQ = _roundPriceQ(rungPrice * quoteScale);
17
18          // ... expensive operations per rung
19          state.quoteNeeded = _recordFill(uint16(rung), true, state.fillBase, state.priceQ, denom);
20          // Each _recordFill performs multiple SSTOREs
21      }
22  }
```

The while loop processes each rung with active liquidity, performing expensive storage writes via `_recordFill()` for Fenwick tree updates.

MAOB.sol - minQuoteMaker Check:

`contracts/MAOB.sol:1640-1642` - Minimum order value validation:

```
1  function _checkMinQuoteMaker(uint256 quoteValue) internal view {
2      if (quoteValue < minQuoteMaker) revert MAOB__QuoteAmountTooLow();
3  }
```

Testnet Configuration (Verified - Partial Mitigation):

Parameter	Contract	Value	Implication
minQuoteMaker	MAOB BTC.b/USDm	10e18 ($10)	Attack cost: 64 × $10 = $640
minQuoteMaker	MAOB USDT0/USDm	10e18 ($10)	Provides economic barrier
minQuoteMaker	MAOB WETH/USDm	10e18 ($10)	May still be viable for griefing

Verification command:

```
1  cast call 0xd1B3581797972638 61bA4f1f18A94002596b5db "minQuoteMaker()" --rpc-url https://carrot.
       megaeth.com/rpc
```

Impact

- Takers may experience 10-20x gas amplification when filling orders spread across many dust rungs
- Potential transaction failures during network congestion due to gas limit exceeded
- Economic griefing against specific users who need to fill large orders

- Current $10 minimum provides meaningful barrier ($640 to fill all 64 rungs) but may still be viable for targeted griefing in high-value markets

Remediation

Why `maxRung` Does Not Mitigate This Issue:

The `maxRung` parameter exists on all taker functions but does **not** address dust griefing:

`maxRung` Capability	Dust Griefing Requirement
Sets upper price bound (rungs 0 to N)	Need to skip specific dusty rungs
Requires advance knowledge of dust locations	No on-chain visibility into rung liquidity quality
Would help if dust only at high rungs	Attackers target low rungs (near mid-price)

Example: If rungs 0, 2, 4, 6 have dust and rungs 1, 3, 5 have real liquidity, `maxRung=6` still processes all dusty rungs. There's no way to say "process rungs 1, 3, 5 only."

Recommended Actions:

1. **Implement minimum fill threshold per rung**: Add logic to automatically skip rungs where the available liquidity is below a configurable dust threshold (e.g., `minRungLiquidity`). The fill loop would check `if (totals.volume < minRungLiquidity) continue;` - this removes the burden from takers and makes dust attacks ineffective.

2. **Implement minimum rung liquidity requirement**: Add a mechanism that automatically clears or consolidates orders when a rung's total liquidity falls below a minimum threshold. This prevents dust from accumulating in the first place.

Resolution

Fixed in commit: `f97d0f2913340425c9bc3b64325f0b4f93e4d88e`

The fix implements option 1 (minimum fill threshold per rung):

- A new `minQuotePerRung` parameter is added to all taker functions
- When `minQuotePerRung != 0`, rungs with liquidity below this threshold are skipped
- This gives takers the ability to avoid dust rungs and mitigate the gas amplification attack
- The parameter is optional (0 disables) to maintain backward compatibility

Finding 5: Sole Lp Fee Bypass

Severity: MEDIUM
Status: ACKNOWLEDGED

Description

The withdrawal fee logic in `CLP.withdrawDual()` contains an off-by-one error that allows any **sole LP** (or last remaining LP) to withdraw their entire position without paying the withdrawal fee.

Due to the `MINIMUM_LIQUIDITY` mechanism, 1000 shares are permanently locked when a vault is created. This means the maximum shares any user can hold is `totalSupply - 1000`. When a user holds this maximum amount (i.e., they are the sole LP), the fee condition evaluates to `FALSE` and the fee is bypassed.

Why this happens:

```
if (shares + MINIMUM_LIQUIDITY < supply) {
    // charge fee
}
```

For a sole LP:

- User holds `supply - 1000` shares (the maximum possible)
- Condition: `(supply - 1000) + 1000 < supply` → `supply < supply` → **FALSE**
- Fee block is skipped entirely

This is not a manipulation or edge case - **any sole shareholder automatically bypasses the fee** simply by being the only LP in the vault.

Source

CLP.sol - withdrawDual Function:

`contracts/CLP.sol:225-231` - Withdrawal fee bypass condition:

```
uint256 feeShares = Math.mulDiv(shares, withdrawalFeeBps, BPS);
if (feeShares > 0 && feeShares < shares && shares + MINIMUM_LIQUIDITY < supply) {
    _burn(msg.sender, feeShares);
    emit WithdrawalFeeSharesBurned(msg.sender, feeShares);
    shares -= feeShares;
    supply -= feeShares;
}
```

CLP.sol - MINIMUM_LIQUIDITY Constant:

contracts/CLP.sol:20:

```
uint256 private constant MINIMUM_LIQUIDITY = 1000;
```

Affected Vaults

Each CLP vault is independent and affected:

CLP Vault	withdrawalFeeBps	Fee Bypassed
WETH/USDm	10 (0.1%)	Yes, for sole LP
BTC.b/USDm	10 (0.1%)	Yes, for sole LP
USDT0/USDm	10 (0.1%)	Yes, for sole LP

Impact

- **Guaranteed fee bypass for sole LPs**: No manipulation required - any user who is the only shareholder automatically pays no withdrawal fee
- **Most impactful scenarios**:
  - Early depositors in new vaults (first LP is sole LP)
  - Low-activity vaults where LPs exit leaving one remaining
  - Whale-dominated vaults where one LP holds majority
- **Protocol revenue loss**: For a $10M position with 0.1% fee, $10K in fees bypassed
- **Incentive misalignment**: The largest withdrawer (sole LP) pays no fee, while smaller LPs in multi-user vaults pay full fee

Remediation

Change the condition from < to <=:

```
if (feeShares > 0 && feeShares < shares && shares + MINIMUM_LIQUIDITY <= supply) {
    _burn(msg.sender, feeShares);
    emit WithdrawalFeeSharesBurned(msg.sender, feeShares);
    shares -= feeShares;
    supply -= feeShares;
}
```

This ensures the fee is applied even when the user holds the maximum possible shares (`supply - MINIMUM_LIQUIDITY`).

POC

File: `Sole_LP_Fee_Bypass/WithdrawalFeeBypassPoC.t.sol`

Scenario:

1. User deposits into CLP vault, becoming the sole LP
2. User holds `totalSupply - 1000` shares (the maximum possible)
3. User calls `withdrawDual()` with all their shares
4. Condition `shares + 1000 < supply` evaluates to `supply < supply` → FALSE
5. Fee block is skipped, user withdraws without paying 0.1% fee

How to Run:

```
cd canonic-contracts-main-latest
forge test --match-contract WithdrawalFeeBypassTest -vvv
```

Acknowledgment

Status: Intended Design

The team clarified that this behavior is intentional:

> "The withdrawal fee goes to the remaining LPs, not to the protocol. The purpose of this withdrawal fee is to prevent gaming behavior, such as users timing deposits and withdrawals to extract arbitrage at the expense of other LPs. When a user withdraws, they pay a fee that is redistributed to the other LPs for this reason. However, if a user is the last LP to withdraw, there are no remaining LPs to be harmed or gamed. Since there is no one left to protect, the 0.1% withdrawal fee is not charged to the final LP."

This explanation is economically sound:

- The fee mechanism works via share burning, which benefits remaining shareholders
- The `MINIMUM_LIQUIDITY` (1000 shares) is permanently locked and cannot be withdrawn
- If a fee were charged to the sole LP, the burned value would accrue to shares that can never be claimed
- Charging a fee that benefits no one would be economically wasteful

The behavior is acknowledged as intended design rather than a vulnerability.

Finding 6: Asymmetric Pause State

Severity: MEDIUM
Status: RESOLVED

Description

The MAOB contract implements independent pause controls for maker and taker operations via `setMakerPaused()` and `setTakerPaused()`. This design allows the contract to enter asymmetric pause states where one side of the market is frozen while the other remains active.

Asymmetric Pause Analysis:

Asymmetric pause capabilities can serve legitimate purposes, but require careful constraints to avoid market integrity risks:

Pause State	Legitimate Use	Risk Level	Notes
Both paused	Emergency shutdown	Low	Standard, unambiguous
Neither paused	Normal operation	None	Expected state
Makers paused, takers active	Emergency unwind	Medium	Allows users to exit positions; prevents new potentially problematic orders
Takers paused, makers active	No clear use case	High	Allows book shaping without price discovery; resembles market manipulation

"Makers paused, takers active" can be defensible as an emergency unwind mode—for example, if a bug is discovered in order validation logic, halting new orders while allowing users to exit existing positions is reasonable. However, this requires:

- Time bounds (auto-expire after N blocks)
- Public reason codes
- Pre-defined as an explicit emergency state, not arbitrary admin discretion

“Takers paused, makers active” is difficult to justify. It allows the order book to be reshaped without market clearing, creating conditions where insiders could seed favorable orders before execution resumes.

Current Implementation Concerns:

The current implementation provides unconstrained discretionary control rather than a bounded emergency state machine:

1. **No time limits**: Asymmetric states can persist indefinitely
2. **No reason codes**: No on-chain record of why the state was entered
3. **Arbitrary combinations**: Any pause configuration is possible at admin discretion
4. **No pre-defined states**: Not structured as explicit emergency modes

The Atomic Transaction Problem:

Beyond market integrity concerns, the asymmetric state interacts problematically with standard market maker operations. Professional liquidity providers update orders atomically (cancel old order + place new order in single transaction) to avoid leaving capital idle.

When makers are paused:

1. Atomic update transaction begins
2. `cancelOrder()` executes successfully (not pause-protected)
3. `addLiquidity()` reverts (pause-protected)
4. **Entire transaction reverts, including the cancel**
5. Maker’s old order at stale price remains active
6. Takers (not paused) can execute against the stale order

This creates a scenario where makers cannot update stale positions but those positions remain executable by takers.

Source

MAOB.sol - Independent Pause Controls:

`contracts/MAOB.sol:753-763` - Separate pause functions allow arbitrary asymmetric states:

```
function setTakerPaused(bool paused) external onlyOwner {
    takerPaused = paused;
    emit TakerPaused(paused);
}

function setMakerPaused(bool paused) external onlyOwner {
    makerPaused = paused;
    emit MakerPaused(paused);
}
```

MAOB.sol - Asymmetric Enforcement:

`contracts/MAOB.sol:1659-1665` - Pause checks are applied independently:

```
function _checkTakerNotPaused() internal view {
    if (takerPaused) revert MAOB__TakerPaused();
}

function _checkMakerNotPaused() internal view {
    if (makerPaused) revert MAOB__MakerPaused();
}
```

Impact

- **Market Integrity Risk**: Unconstrained asymmetric pause allows states that could be perceived as or enable market manipulation
- **Atomic Update Trap**: Standard market maker patterns (cancel + replace) fail in a way that leaves stale orders active and executable
- **Value Extraction Window**: The “makers paused, takers active” state without time bounds allows extended periods where makers cannot protect their positions
- **Governance Trust**: Arbitrary pause combinations without reason codes make post-mortems difficult and reduce user trust

Remediation

Two approaches, depending on whether asymmetric pause capability is desired:

Option 1 - Single Pause Flag (Simplest):

If asymmetric pause is not required, replace independent flags with a single unified pause:

```
bool public paused;

function setPaused(bool _paused) external onlyOwner {
    paused = _paused;
    emit Paused(_paused);
}

function _checkNotPaused() internal view {
    if (paused) revert MAOB__Paused();
}
```

This is the most defensible default—simple, auditable, and eliminates ambiguity.

Option 2 - Constrained Emergency State Machine:

If asymmetric pause serves a legitimate emergency function, constrain it to pre-defined states with bounds:

```
enum MarketState {
    Active,
    Halted,          // Both paused
    UnwindOnly       // Makers paused, takers active (time-limited)
}

MarketState public marketState;
uint256 public stateExpiresAt;

function setMarketState(MarketState _state, string calldata reason) external onlyOwner {
    if (_state == MarketState.UnwindOnly) {
        stateExpiresAt = block.timestamp + 1 hours;  // Auto-expire
    }
    marketState = _state;
    emit MarketStateChanged(_state, reason);
}
```

Key constraints:

- **Pre-defined states only**: No arbitrary combinations
- **Time bounds**: Asymmetric states auto-expire
- **Reason codes**: On-chain record of intent
- **No "takers paused, makers active"**: This combination is excluded as it lacks legitimate use

Either approach improves market integrity and governance trust.

Resolution

Fixed in commit: `0d66b87aa5f0c830be534d27b5ecae9c17477c43`

The fix implements Option 2 (Constrained Emergency State Machine):

- Replaced `takerPaused` and `makerPaused` booleans with a `MarketState` enum: `Active`, `Halted`, `UnwindOnly`

- `UnwindOnly` state has a 1-hour auto-expiry (`UNWIND_DURATION = 1 hours`)
- Guardian can set `Halted` or `UnwindOnly` instantly, but only owner (timelock) can set `Active`
- The “takers paused, makers active” combination is explicitly excluded
- Reason codes are required via a `string calldata reason` parameter

Finding 7: Withdrawal Fee Dilution

Severity: MEDIUM
Status: RESOLVED

Description

The CLP withdrawal fee mechanism suffers from an order-of-operations flaw that causes large liquidity providers to pay significantly less than the configured fee rate. The fee is implemented by burning shares before calculating the withdrawal payout, but because the payout calculation uses the post-burn supply, users capture a portion of the value accretion caused by their own fee burn.

The Mathematical Problem:

When shares are burned, the total supply decreases, which increases the value per remaining share. Since the withdrawing user still holds their remaining shares at the moment of payout calculation, they benefit from this supply reduction—effectively "buying back" part of their own fee.

Effective Fee Formula:

```
Effective Fee = Nominal Fee × (1 - userShareOfPool)
```

User Pool Share	Configured Fee	Effective Fee	Fee Reduction
10%	10%	9.0%	10% less
25%	10%	7.5%	25% less
50%	10%	5.0%	50% less
75%	10%	2.5%	75% less
90%	10%	1.0%	90% less

Worked Example:

Setup:

- Pool holds 1,000 Base tokens, 1,000 Quote tokens
- Total supply: 1,000 shares
- User owns 500 shares (50% of pool)
- Withdrawal fee: 10%

Current implementation:

1. `feeShares = 500 × 10% = 50`
2. Burn 50 shares → supply becomes 950
3. User's remaining shares: 450
4. `baseOut = 1000 × (450 / 950) = 473.68 Base`

Correct implementation:

1. `feeShares = 500 × 10% = 50`
2. Net shares: 450
3. `baseOut = 1000 × (450 / 1000) = 450 Base`

Result: User receives 473.68 Base instead of 450 Base—an effective fee of 5.26% instead of 10%.

Whale Example - Extreme Case:

Setup:

- Pool holds 1,000 Base tokens
- Total supply: 1,000 shares
- Whale owns 990 shares (99% of pool)
- Small LPs own 10 shares (1% of pool)
- Withdrawal fee: 10%

Current implementation:

1. Whale withdraws 990 shares
2. `feeShares = 990 × 10% = 99` shares burned
3. New supply = 901, whale's remaining shares = 891
4. `baseOut = 1000 × (891 / 901) = 989 Base`

Correct implementation:

1. `baseOut = 1000 × (891 / 1000) = 891 Base`

Outcome	Buggy	Correct
Whale receives	989 Base	891 Base
Pool remaining	11 Base	109 Base
Small LPs gain from fee	1 Base	99 Base
Whale's effective fee	**0.1%**	10%

The whale reclaims 98 of the 99 Base fee value. Due to the calculation order, the fee value that should benefit remaining LPs (99 Base) is largely retained by the withdrawing user, leaving small LPs with only 1 Base instead of 99 Base.

Source

CLP.sol - withdrawDual Fee Calculation:

`contracts/CLP.sol:225-234` - Fee shares burned before payout calculation:

```
uint256 feeShares = Math.mulDiv(shares, withdrawalFeeBps, BPS);
if (feeShares > 0 && feeShares < shares && shares + MINIMUM_LIQUIDITY < supply) {
    _burn(msg.sender, feeShares);
    emit WithdrawalFeeSharesBurned(msg.sender, feeShares);
    shares -= feeShares;
    supply -= feeShares;  // Supply reduced BEFORE payout calculation
}

baseOut = Math.mulDiv(totalBase, shares, supply);  // Uses reduced supply
quoteOut = Math.mulDiv(totalQuote, shares, supply);
```

The vulnerability is in lines 229-230: `supply` is decremented by `feeShares` before being used as the denominator in the payout calculation. This inflates the payout per remaining share, allowing the user to recapture value from their burned fee shares.

Impact

- **Protocol Revenue Loss**: CLP vaults collect less fee revenue than configured, scaling with LP concentration
- **Whale Advantage**: Large liquidity providers pay disproportionately lower effective fees than small LPs
- **Defeated Economic Purpose**: If withdrawal fees are intended to discourage short-term liquidity or protect remaining LPs, the mechanism fails for the users it matters most (large withdrawals)
- **Equity Concern**: Small LPs pay close to the full fee rate while whales pay a fraction

Remediation

Calculate the payout using the original supply value, ensuring the fee deduction is not diluted by the share burn:

Option 1 - Calculate with original supply:

```
uint256 feeShares = Math.mulDiv(shares, withdrawalFeeBps, BPS);
uint256 netShares = shares;

if (feeShares > 0 && feeShares < shares && shares + MINIMUM_LIQUIDITY < supply) {
    _burn(msg.sender, feeShares);
    emit WithdrawalFeeSharesBurned(msg.sender, feeShares);
    netShares = shares - feeShares;
    // Do NOT update supply variable used in calculation
}

// Use original supply, not post-burn supply
baseOut = Math.mulDiv(totalBase, netShares, supply);
quoteOut = Math.mulDiv(totalQuote, netShares, supply);
```

Option 2 - Calculate gross payout first, then deduct fee:

```
// Calculate gross payout first
uint256 grossBase = Math.mulDiv(totalBase, shares, supply);
uint256 grossQuote = Math.mulDiv(totalQuote, shares, supply);

// Apply fee to payout amounts
uint256 feeBase = Math.mulDiv(grossBase, withdrawalFeeBps, BPS);
uint256 feeQuote = Math.mulDiv(grossQuote, withdrawalFeeBps, BPS);

baseOut = grossBase - feeBase;
quoteOut = grossQuote - feeQuote;

// Handle share burning separately if needed for accounting
```

Either approach ensures that the effective fee rate matches the configured `withdrawalFeeBps` regardless of user's pool share.

POC

File: `Withdrawal_Fee_Dilution/WithdrawalFeeDilutionPoC.t.sol`

Scenario:

1. Whale owns 99% of CLP vault (990 of 1000 shares)
2. Withdrawal fee is configured at 10%
3. Whale calls `withdrawDual()` with all shares
4. Fee shares (99) are burned, supply drops to 901
5. Payout calculated: `1000 × (891 / 901) = 989` instead of `891`

6. Whale receives 989 instead of 891, effectively paying 0.1% fee
7. Small LPs receive only 1 of the 99 fee value

How to Run:

```
cd canonic-contracts-main-latest
forge test --match-contract WithdrawalFeeDilutionTest -vvv
```

Test Functions:

- `test_50PercentLP_PaysHalfFee`: 50% LP pays ~5% instead of 10%
- `test_99PercentWhale_PaysAlmostNoFee`: 99% whale pays ~0.1% instead of 10%
- `test_FeeDilutionScalesWithPoolShare`: Verifies formula across pool shares

Resolution

Fixed in commit: `93d3b229d51f19a670fb778d4cc6c2690603b0a6`

The fix implements Option 1 (calculate with original supply):

- Introduced `netShares` variable to track shares after fee deduction
- The `supply` variable is no longer modified after fee share burning
- Payout is calculated as `Math.mulDiv(totalBase, netShares, supply)` using the original supply
- This ensures the effective fee rate matches the configured `withdrawalFeeBps` regardless of user's pool share

Finding 8: Rebasing Token Insolvency

Severity: LOW
Status: ACKNOWLEDGED

Description

MAOB uses internal accounting (`askRungs[].volume`, `bidRungs[].volume`) to track liabilities. If a negative rebasing token (e.g., stETH during slashing events) is used as base or quote token, the contract's actual balance will drop below its internal liabilities, causing insolvency where last withdrawers cannot retrieve their funds.

Source

MAOB.sol - Internal Accounting:

`contracts/MAOB.sol` - Volume tracked separately from actual balance:

```
// Rung storage tracks expected volumes
struct RungTotals {
    uint256 volume;  // Internal accounting of expected funds
}

mapping(uint16 => RungTotals) public askRungs;
mapping(uint16 => RungTotals) public bidRungs;
```

The contract tracks `volume` as internal accounting but never reconciles this against actual token balances. For standard ERC20 tokens, these values remain synchronized. However, rebasing tokens can change balance independent of transfers.

MAOB.sol - Withdrawal Logic:

`contracts/MAOB.sol:1597-1612` - Withdraws based on internal accounting:

```
function _withdraw(address user) internal returns (uint256 baseOut, uint256 quoteOut) {
    baseOut = withdrawableBase[user];
    quoteOut = withdrawableQuote[user];
    if (baseOut == 0 && quoteOut == 0) return (0, 0);

    if (baseOut != 0) {
        withdrawableBase[user] = 0;
        baseToken.safeTransfer(user, baseOut);  // May fail if actual balance < internal accounting
    }
    if (quoteOut != 0) {
        withdrawableQuote[user] = 0;
        quoteToken.safeTransfer(user, quoteOut);
    }
}
```

CLP.sol - Same Issue:

`contracts/CLP.sol` - CLP inherits this vulnerability through its interaction with MAOB and its own internal accounting for share valuations.

Impact

- Complete loss for last withdrawers, potentially up to 100% of their deposited tokens
- No recovery mechanism exists within the contract
- Protocol becomes insolvent without admin intervention
- Particularly dangerous for stETH-like tokens that can experience negative rebases during validator slashing events
- The insolvency is permanent and irreversible without external fund injection

Remediation

Maintain a blessed whitelist of approved non-rebasing tokens. Admins should only deploy MAOB and CLP instances with tokens from this whitelist to protect against rebasing token insolvency. Tokens such as stETH, AMPL, or other rebasing/elastic supply tokens must be explicitly excluded from the whitelist.

Acknowledgment

Status: Known Design Limitation

Team Response:

> "Noted. We can only deploy markets with tokens that have fast oracles. This set of tokens is very small. Operationally the team will verify that each token is non-rebasing before deploying."

The protocol is intended to work with standard ERC20 tokens only. The team has committed to verifying that each token is non-rebasing as part of their deployment procedures. Given that the supported token set is constrained by oracle availability, the operational risk is mitigated through deployment-time verification.

Disclaimer

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

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

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

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

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

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

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

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

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

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

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

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

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

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