CANTINA

mUSD
Security Review

Cantina Managed review by:

Alex the Entreprenerd, Security Researcher
High Byte, Security Researcher

April 15, 2025

Contents

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	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

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 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

Mezo, by Thesis, is a Bitcoin-centric platform designed to enhance Bitcoin's utility through seamless borrowing, spending, and earning.

From Mar 5th to Mar 15th the Cantina team conducted a review of musd on commit hash 0152248b. The team identified a total of **29** issues:

Issues Found

Severity	**Count**	**Fixed**	**Acknowledged**
Critical Risk	1	1	0
High Risk	6	6	0
Medium Risk	4	4	0
Low Risk	10	9	1
Gas Optimizations	1	1	0
Informational	7	5	2
Total	**29**	**26**	**3**

3 Findings

3.1 Critical Risk

3.1.1 `restrictedRefinance` can be spammed to make ones own liquidatable trove cause bad debt to the system

Severity: Critical Risk

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

Description: Liquity is a very delicate system. Every operation is constrained to make sure an attacker cannot negatively impact the system and to make liquidations profitable. `restrictedRefinance` allows us to break this key invariant in a unique way.

Since:

- There's no check nor cooldown to slow us from calling this multiple times.
- We can set the same rate.
- Every time we do we get a multiplicative amount of new debt.

We can use these properties to attack the system towards an unrecoverable state by:

- Self liquidating to trigger other liquidations (which we can profit from via liquidation incentives and caller incentives).

Proof of Concept: An attacker can simply call `restrictedRefinance` multiple times to inflate their debt by 2% each time, quickly triggering a self liquidation. Or if they so chose, can simply inflate the total debt to reach 2ˆ256 and permanently cause a DoS:

```
// forge test --match-test test_crytic -vvv
function test_crytic() public {
    // TODO: add failing property tests here for debugging
    cl3_setLatestRoundDataReturn(0, 2000e18, 0, 0, 0);

    // Mint something
    this.borrowerOperations_openTrove{value: 1_000_000e18}(500_000_000e18, address(0), address(0));

        _logStates();

    // Refinance spam
    uint256 x;

    while(x++ < 12267) {
        borrowerOperations_refinance(); // It's basically 10BPS
        console2.log("x", x);
    }

    _logStates();
}

// Desynch in values - CRIT -> Scary
// 2 Order of magnitude off, It's because it's 10 BPS -> 20% of total assets -> 50BPS on that = 10BPS on total
// Ability to spam is confirmed = CRIT


function _logStates() internal {
    uint256 price = priceFeed.fetchPrice();
    uint256 TCR = troveManager.getTCR(price);

    console2.log("TCR", TCR);
    console2.log("getTroveDebt", troveManager.getTroveDebt(_getActor()));
    console2.log("tm getEntireSystemColl", troveManager.getEntireSystemColl());
    console2.log("tm getEntireSystemDebt", troveManager.getEntireSystemDebt());
    // console2.log("sp getEntireSystemColl", stabilityPool.getEntireSystemColl());
    // console2.log("sp getEntireSystemDebt", stabilityPool.getEntireSystemDebt());
    console2.log("bo getEntireSystemColl", borrowerOperations.getEntireSystemColl());
    console2.log("bo getEntireSystemDebt", borrowerOperations.getEntireSystemDebt());
}
```

Logs:

```
[PASS] test_crytic() (gas: 1116117)
Logs:
  TCR 3980097918368987713
  getTroveDebt 5025002000000000000000000
  tm getEntireSystemColl 1000000000000000000000000
  tm getEntireSystemDebt 5025002000000000000000000
  bo getEntireSystemColl 1000000000000000000000000
  bo getEntireSystemDebt 5025002000000000000000000
  x 1
  x 2
  x 3
  x 4
  x 5
  TCR 3980097918368987713
  getTroveDebt 5050177310295150035002000000
  tm getEntireSystemColl 1000000000000000000000000
  tm getEntireSystemDebt 5025002000000000000000000
  bo getEntireSystemColl 1000000000000000000000000
  bo getEntireSystemDebt 5025002000000000000000000
```

Recommendation:

- Ensure that `refinance` cannot be called more than once per trove.
- Ensure that `refinance` will not trigger a self-liquidation.
- Ensure that `refinance` will not trigger RM.

Extra notes: I originally believed you could totally brick the system, I don't believe that's possible because `refinance` effectively charges a 10 BPS fee on total debt. If that fee was more than that (e.g. 2%) then the multiplicative impact would be bigger and it could most likely cause liquidations to no longer be possible.

Thesis: Fixed in PR 172.

Cantina Managed: Fix verified.

3.2 High Risk

3.2.1 Changing `musdGasCompensation` will break the system's accounting

Severity: High Risk

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

Description: The variable `musdGasCompensation` is used in key accounting:

- Sum of all compensations in the gas pool (BorrowerOperations.sol#L562-L569):

```
_withdrawMUSD(
    contractsCache.activePool,
    contractsCache.musd,
    gasPoolAddress,
    musdGasCompensation,
    musdGasCompensation
);
```

- In Closing a Trove (BorrowerOperations.sol#L637-L653):

```
activePoolCached.decreaseDebt( /// @audit TODO: Can this cause an overflow?? | TODO: How does accountinh
↪   work?
    debt - musdGasCompensation - interestOwed,
    interestOwed
);

// Burn the repaid mUSD from the user's balance
musdTokenCached.burn(_borrower, debt - musdGasCompensation);

// Burn the gas compensation from the gas pool
_repayMUSD(
    activePoolCached,
    musdTokenCached,
    gasPoolAddress,
    musdGasCompensation,
    0
);
```

- In Redemptions (TroveManager.sol#L1508-L1511):

```
singleRedemption.mUSDLot = LiquityMath._min(
    _maxMUSDamount,
    _getTotalDebt(_borrower) - redeemCollateralVars.gasCompensation
);
```

This can have one of the following impacts based on whether the gas compensation is increased or reduced.

Proof of Concept:

- If increased:
  - It will cause overflows and prevent closing and liquidation (as well as allow breaking redemptions due to overflows).
- If reduced:
  - It will cause some of the Trove debt to be stuck in the Trove (since when you repay debt), I'm unsure about the secondary effects on this, but I think it could cause issue especially if troves are opened and closed as you'd have ghost debt in the system.
  - This can also cause RM in a way that is not actionable, as there will be debt in the system with 0 collateral and no trove assigned to it.

```
// forge test --match-test test_overflow -vvv
function test_overflow() public {
    // Set Price
    cl3_setLatestRoundDataReturn(0, 2000e18, 0, 0, 0);

    // Open Second Trove so we can close
    switchActor(1);
    vm.deal(_getActor(), 1_000_000e18);
    this.borrowerOperations_openTrove{value: 1_000_000e18}(500_000_000e18, address(0), address(0));

    // Actor 0 is also governance, open
    switchActor(0);
    this.borrowerOperations_openTrove{value: 1_000_000e18}(500_000_000e18, address(0), address(0));

    // Faster than figuring it out | Now we have liquidity to close
    deal(address(musdToken), address(this), 1_000_000_000e18);

    // Change gas compensation to show bug
    borrowerOperations_proposeMusdGasCompensation(borrowerOperations.musdGasCompensation() * 10);

    vm.warp(block.timestamp + 7 days);
    borrowerOperations_approveMusdGasCompensation();

    console2.log("musdToken.balanceOf(address(gasPool))", musdToken.balanceOf(address(gasPool)));

    borrowerOperations_closeTrove();

    console2.log("musdToken.balanceOf(address(gasPool))", musdToken.balanceOf(address(gasPool)));
}
```

```
Failing tests:
Encountered 1 failing test in test/recon/CryticToFoundry.sol:CryticToFoundry
[FAIL: ERC20InsufficientBalance(0x756e0562323ADcDA4430d6cb456d9151f605290B, 400000000000000000000 [4e20],
↪  2000000000000000000000 [2e21])] test_overflow() (gas: 1497193)
```

Recommendation: I believe you should never change the `gasCompensation` after it has been set. Any change to `gasCompensation` should be thoroughly invariant tested and as of now should be considered unsafe.

Thesis: Fixed in PR 170.

Cantina Managed: Fix verified.

3.2.2 `restrictedRefinance` doesn't prevent triggering RM, allowing to liquidate victims

Severity: High Risk

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

Description: If the accounting wasn't broken this would be a bug. Every user operation in Liquity must ensure that:

- A trove cannot self liquidate.
- A trove doesn't trigger other troves liquidations.

Triggering RM does that, and as such is a dangerous transition that needs to be prevented, due to:

- Not accruing the total system debt.
- Not explicitly checking the CR of the system after refinancing.

The function can be used to trigger RM and liquidate all borrowers below the Critical Threshold.

Proof of Concept:

- Borrow to the Critical Threshold.
- Refinance to trigger RM.
- Liquidate Victims.

Recommendation: Ensure that the system debt has been accrued and from fresh value check the TCR that would result from refinancing. If that would trigger RM, revert.

Thesis: Fixed in PR 172.

Cantina Managed: Fix verified.

3.2.3 Unaccrued `_checkRecoveryMode` in `restrictedOpenTrove` may not report RM when the system is already below the threshold

Severity: High Risk

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

Description: Recovery Mode is an extremely delicate part of Liquity V1. `musd` introduced the concept of different interest rates, and these have to be accrued independently.

Assuming even only 1 interest rate is present, `restrictedOpenTrove` is not accruing that, causing the call to `getEntireSystemDebt` in `_checkRecoveryMode` to use a old, unaccrued total debt. This breaks the key invariant of "*you must accrue first*" in systems where debt has to be accrued, allowing thus to sidestep RM restrictions. Furthermore it can be weaponized to trigger RM and liquidate victims by first performing a deposit that brings the system down close to RM and then accruing the pending global interest.

Recommendation: Ensure `getEntireSystemDebt` is always accrued. You can either virtually accrue it or accrue first and then read it's value. This type of issue may require invariant testing to ensure it's not present in other parts of the code.

Thesis: Fixed in PR 187 and PR 176, which together make it so that all of the debt functions virtually accure and that system debt is accrued at the top of every external call.

Cantina Managed: Fix verified.

3.2.4 `restrictedRefinance` is not updating the total system debt, breaking a core invariant

Severity: High Risk

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

Description: `restrictedRefinance` performs this call to increase the Trove Debt (BorrowerOperations.sol#L673-L674):

```
troveManagerCached.increaseTroveDebt(_borrower, fee);
```

This is not consistent with how the rest of the system works, in which `activePool.increaseDebt` is also called. e.g. `_withdrawMUSD` (BorrowerOperations.sol#L884-L894):

```
function _withdrawMUSD(
    IActivePool _activePool,
    IMUSD _musd,
    address _account,
    uint256 _debtAmount,
    uint256 _netDebtIncrease
) internal {
    _activePool.increaseDebt(_netDebtIncrease, 0);
    _musd.mint(_account, _debtAmount);
}
```

This desynchs the system accounting, breaking a key invariant in the Liquity System that the `SUM(Debts) == getEntireSystemDebt`.

Proof of Concept:

```
// forge test --match-test test_crytic -vvv
function test_crytic() public {
    // TODO: add failing property tests here for debugging
    cl3_setLatestRoundDataReturn(0, 2000e18, 0, 0, 0);

    // Mint something
    this.borrowerOperations_openTrove{value: 1_000_000e18}(500_000_000e18, address(0), address(0));

        _logStates();

    // Refinance spam
    uint256 x;

    while(x++ < 1) {
        borrowerOperations_refinance(); // It's basically 10BPS
        console2.log("x", x);
    }

    _logStates();
}

// Desynch in values - CRIT -> Scary
// 2 Order of magnitude off, It's because it's 10 BPS -> 20% of total assets -> 50BPS on that = 10BPS on total
// Ability to spam is confirmed = CRIT


function _logStates() internal {
    uint256 price = priceFeed.fetchPrice();
    uint256 TCR = troveManager.getTCR(price);

    console2.log("TCR", TCR);
    console2.log("getTroveDebt", troveManager.getTroveDebt(_getActor()));
    console2.log("tm getEntireSystemColl", troveManager.getEntireSystemColl());
    console2.log("tm getEntireSystemDebt", troveManager.getEntireSystemDebt());
    // console2.log("sp getEntireSystemColl", stabilityPool.getEntireSystemColl());
    // console2.log("sp getEntireSystemDebt", stabilityPool.getEntireSystemDebt());
    console2.log("bo getEntireSystemColl", borrowerOperations.getEntireSystemColl());
    console2.log("bo getEntireSystemDebt", borrowerOperations.getEntireSystemDebt());
}
```

```
Logs:
  TCR 3980097918368987713
  getTroveDebt 5025002000000000000000000
  tm getEntireSystemColl 1000000000000000000000000
  tm getEntireSystemDebt 5025002000000000000000000
  bo getEntireSystemColl 1000000000000000000000000
  bo getEntireSystemDebt 5025002000000000000000000
  x 1
  TCR 3980097918368987713
  getTroveDebt 5030027002000000000000000
  tm getEntireSystemColl 1000000000000000000000000
  tm getEntireSystemDebt 5025002000000000000000000
  bo getEntireSystemColl 1000000000000000000000000
  bo getEntireSystemDebt 5025002000000000000000000
```

Recommendation: I believe the function needs to call the `ActivePool` to sync the debt. I'm not 100% confident on the invariant tied to total debt in the system, therefore I believe the bug is significant but I wasn't able to weaponize it into a critical exploit.

Thesis: Fixed in PR 171.

Cantina Managed: Fix verified.

3.2.5 SP Offset doesn't reduce interest, leading to accounting insolvency

Severity: High Risk

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

Description: SP Offset decreases the debt liquidated by a pro-rata portion of `debtToOffset = principalToOffset + interestToOffset;`. This value that mixes both `principal` and `interest` is subtracted from the `activePool` only on the `principal` variable, leading to insolvency over time. Liquidation debt amounts are computed as:

```
debtToOffset = principalToOffset + interestToOffset;
```

- TroveManager.sol#L1874-L1910:

```
function _getOffsetAndRedistributionVals(
    uint256 _principal,
    uint256 _interest,
    uint256 _coll,
    uint256 _MUSDInStabPool
)
    internal
    pure
    returns (
        uint256 debtToOffset,
        uint256 collToSendToSP,
        uint256 principalToRedistribute,
        uint256 interestToRedistribute,
        uint256 collToRedistribute
    )
{
    if (_MUSDInStabPool > 0) {
        /*
         * Offset as much debt & collateral as possible against the Stability Pool, and redistribute the
         ↪  remainder
         * between all active troves.
         *
         *  If the trove's debt is larger than the deposited mUSD in the Stability Pool:
         *
         *  - Offset an amount of the trove's debt equal to the mUSD in the Stability Pool
         *  - Send a fraction of the trove's collateral to the Stability Pool, equal to the fraction of
         ↪  its offset debt
         *
         */
        uint256 interestToOffset = LiquityMath._min(
            _interest,
            _MUSDInStabPool
        );
        uint256 principalToOffset = LiquityMath._min(
            _principal,
            _MUSDInStabPool - interestToOffset
        );
        debtToOffset = principalToOffset + interestToOffset;
        collToSendToSP = (_coll * debtToOffset) / (_principal + _interest); /// @audit a Pro-Rata
        ↪  percentage | This is locking in Losses
```

This includes accumulators on the principal and on the interest. However, offset will perform:

- StabilityPool.sol#L469-L470:

```
activePoolCached.decreaseDebt(_debtToOffset, 0); /// @audit Not liquidating interest = Breaking
↪  accounting
```

Proof of Concept:

- Open 2 Troves.
- Warp a bit and Accrue both.
- Check interest and principal.
- Liquidate 1.
- Check interest and principal (interest is unchanged and it was subtracted in the principal, leading to broken accounting).

```
// forge test --match-test test_liquidate -vvv
function test_liquidate() public {
    // TODO: add failing property tests here for debugging
    cl3_setLatestRoundDataReturn(0, 2000e18, 0, 0, 0);

    interestRateManager.proposeInterestRate(1000);
    vm.warp(block.timestamp + 7 days);
    interestRateManager.approveInterestRate();

    // Open Second Trove so we can close
    switchActor(1);
    vm.deal(_getActor(), 1_000_000e18);
    this.borrowerOperations_openTrove{value: 1_000_000e18}(500_000_000e18, address(0), address(0));

    // Actor 0 is also governance, open
    switchActor(0);
    this.borrowerOperations_openTrove{value: 1_000_000e18}(500_000_000e18, address(0), address(0));

    console2.log("activePool.getPrincipal()", activePool.getPrincipal());
    console2.log("activePool.getInterest()", activePool.getInterest());

    vm.warp(block.timestamp + 10000000000);

    cl3_setLatestRoundDataReturn(0, 0, 0, 0, 0);
    troveManager.updateSystemAndTroveInterest(_getActors()[0]);
    troveManager.updateSystemAndTroveInterest(_getActors()[1]);
    console2.log("activePool.getPrincipal()", activePool.getPrincipal());
    console2.log("activePool.getInterest()", activePool.getInterest());
    troveManager_liquidate(_getActors()[1]);

    console2.log("activePool.getPrincipal()", activePool.getPrincipal());
    console2.log("activePool.getInterest()", activePool.getInterest());
}
```

```
[PASS] test_liquidate() (gas: 1425418)
Logs:
  activePool.getPrincipal() 1005000400000000000000000000
  activePool.getInterest() 0
  activePool.getPrincipal() 1005000400000000000000000000
  activePool.getInterest() 31868353627600202942668696093
  activePool.getPrincipal() 502500200000000000000000000
  activePool.getInterest() 31868353627600202942668696093
```

Recommendation: Either change the liquidation logic to move all of the interest to the principal of the trove that will be closed, or change the SP Offset Function to subtract both the principal and the interest.

Thesis: Fixed in PR 177.

Cantina Managed: Fix verified.

3.2.6 Bad Debt Redistirbution logic in Recovery Mode is not redistributing the interest

Severity: High Risk

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

Description: The code for `_liquidateRecoveryMode` has a clause for when the trove ICR is below 100 PCT.

- TroveManager.sol#L1330-L1346:

```
if (_ICR <= _100pct) {
    _removeStake(_borrower);
    _movePendingTroveRewardsToActivePool(
        _activePool,
        _defaultPool,
        vars.pendingColl,
        vars.pendingPrincipal,
        vars.pendingInterest
    );

    singleLiquidation.debtToOffset = 0;
    singleLiquidation.collToSendToSP = 0;
    singleLiquidation.principalToRedistribute = singleLiquidation
        .entireTrovePrincipal;
    singleLiquidation.collToRedistribute = vars.collToLiquidate;
    /// @audit Is this Missing Interest!
    _closeTrove(_borrower, Status.closedByLiquidation);
```

The struct `LiquidationValues` has fields for both a trove Interest and principal:

```
uint256 entireTrovePrincipal;
uint256 entireTroveInterest;
```

This means that in this logic case, the `singleLiquidation.totalInterestToRedistribute` is not being updated. This will lead to the interest being forgiven, technically making the system lock in bad debt that won't be resolved.

Proof of Concept:

- Trigger RM.
- Liquidate.
- Check the interest (you should see that the interest is still in the AP and that it was not redistributed).

Recommendation: Add the `totalInterestToRedistribute` to totals. Also ensure that the TM will properly update the total debt in the AP using those totals (for both principal and interest).

Thesis: This should be resolved as a result of removing special case liquidations in recovery mode (PR 173).

Cantina Managed: Fix verified.

3.3 Medium Risk

3.3.1 Seems like refinancing is better than opening a new trove, except during RM

Severity: Medium Risk

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

Description: Since the fee is a % of the debt, then this seems better. We could technically reduce the fee on refinance by reducing the debt, but that doesn't seem like a winning move, since this will trigger `_triggerBorrowingFee` outside of RM.

Proof of Concept: Whereas in RM it seems like this can be gamed:

- Reduce debt to max (to pay less).
- Raise Coll to max (to abuse the limit being cached).
- Then borrow from adjust (while improving CR, meaning we need even more collateral).

Recommendation: You could prevent refinancing during RM, which is probably a good move.

Thesis: Fixed in PR 167.

Cantina Managed: Fix verified.

3.3.2 Borrow Capacity can be gamed

Severity: Medium Risk

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

Description: `restrictedAdjustTrove` has the following snippet:

- BorrowerOperations.sol#L804-L809:

```
vars.maxBorrowingCapacity = contractsCache
    .troveManager
    .getTroveMaxBorrowingCapacity(_borrower);
if (_isDebtIncrease) {
    _requireHasBorrowingCapacity(vars);
}
```

Where the capacity is evaluated on `open` and `refinance`. This is gameable as we can `open` with a ton of temporary collateral, to get an outsized amount of `borrowCapacity`. Similarly, since `adjusting` doesn't reduce the capacity nor alter it in any way, we can simply increase the collateral via `adjust` and then `refinance` to get an outsized amount of `borrowCapacity`.

Proof of Concept:

- Open or Adjust to have an abnormally high Collateral amount (then refinance if you adjust).
- Remove the Collateral.
- The borrow capacity hasn't decreased.

Recommendation: In my opinion the logic for borrow capacity needs to be rethought, you may want to over time re-compute it and take the min of what was set and what's available, on each new `adjust`. Similarly, `refinance` is gameable unless you modify `adjustTrove` to reduce the capacity once a user withdraws some collateral.

Thesis: Lowering borrowing capacity on collateral withdrawal definitely makes sense. Fixed in PR 164.

Cantina Managed: Fix verified.

3.3.3 Deprecating the system can open up to RM Liquidations by closing Troves

Severity: Medium Risk

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

Description: `restrictedCloseTrove` has the following check:

- BorrowerOperations.sol#L598-L601:

```
uint256 price = priceFeed.fetchPrice();
if (canMint) {
    _requireNotInRecoveryMode(price);
}
```

Which ensures that RM cannot be triggered by closing a trove during normal operations. If the system starts being deprecated by setting `musdTokenCached.mintList(address(this)); = false` then this protection will be removed. This will create a scenario where possibly, any Trove that has it's CR below CCR could be liquidated during deprecation.

Proof of Concept:

- Close a healthy trove.
- Trigger RM.
- Liquidate victims.

Recommendation: I believe thUSD acknowledged this risk. As long as this is properly explained to end users this is acceptable, although can result in unexpected losses for end users.

Thesis: Fixed in PR 173 by removing special recovery mode liquidations.

Cantina Managed: Fix verified.

3.3.4 `getPendingDebt` is not adding the pending default debt interest, causing view functions and hints to be inaccurate

Severity: Medium Risk

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

Description: `getPendingDebt` looks as follows:

- TroveManager.sol#L877-L903:

```
function getPendingDebt(
    address _borrower
)
    public
    view
    override
    returns (uint256 pendingPrincipal, uint256 pendingInterest)
{
    uint256 principalSnapshot = rewardSnapshots[_borrower].principal;
    uint256 principalPerUnitStaked = L_Principal - principalSnapshot;

    uint256 interestSnapshot = rewardSnapshots[_borrower].interest;
    uint256 interestPerUnitStaked = L_Interest - interestSnapshot;

    if (
        principalPerUnitStaked == 0 ||
        Troves[_borrower].status != Status.active
    ) {
        return (0, 0);
    }

    uint256 stake = Troves[_borrower].stake;

    pendingPrincipal = (stake * principalPerUnitStaked) / DECIMAL_PRECISION;
    pendingInterest = (stake * interestPerUnitStaked) / DECIMAL_PRECISION;
}
```

It uses the storage values `L_Principal` and `L_Interest`. In various functions, that are not `view`, such as `_applyPendingRewards` a call to `updateDefaultPoolInterest` is performed before calling `getPendingDebt`, this results in the storage values `L_Principal` and `L_Interest` to be correctly updated.

In `hintHelpers` the calls to HintHelpers.sol#L97-L98

```
troveManager.getCurrentICR(currentTroveuser, _price) < MCR
```

And HintHelpers.sol#L117-L122

```
(uint256 pendingPrincipal, uint256 pendingInterest) = troveManager
    .getPendingDebt(currentTroveuser);

uint256 netDebt = borrowerOperations.getNetDebt(
    troveManager.getTroveDebt(currentTroveuser)
) +
```

are not calling `updateDefaultPoolInterest`, which will lead to returning incorrect values once some bad debt liquidations have happened.

Proof of Concept: Due to the complexity of setup I'm unable to write a coded proof of concept.

- Generate Bad Debt to be redistributed.
- See that a call to `updateDefaultPoolInterest` changes the TCR and the ICR of each trove.

Recommendation: Ideally you would change the code to accrue virtually and formally verify this property (should be doable with Halmos).

Alternatively you can monitor if the system ever locks in bad debt and then consider deploying a new HintHelpers, which can be non-view but `eth_call` from offchain via a mechanism similar to the Uniswap V3 Quoter (QuoterV2.sol#L80-L97).

Thesis: Fixed in PR 187.

Cantina Managed: Fix verified.

3.4 Low Risk

3.4.1 Lack of staleness check

Severity: Low Risk

Context: PriceFeed.sol#L44-L49

Description: The price feed is not checking if the oracle is stale. This can open up to:

- Incorrect Liquidations.
- Overborrowing.
- Value leak through redemptions.

Recommendation: Add a check to ensure the oracle was updated around it's intended update frequence.

Thesis: Fixed in PR 178.

Cantina Managed: Fix verified.

3.4.2 Discrepancy between value sent and reported in `restrictedAdjustTrove`

Severity: Low Risk

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

Description: The functon `restrictedAdjustTrove` receives `_assetAmount`.

- BorrowerOperations.sol#L703-L714:

```
function restrictedAdjustTrove(
    address _borrower,
    address _recipient,
    address _caller,
    uint256 _collWithdrawal,
    uint256 _mUSDChange,
    bool _isDebtIncrease,
    uint256 _assetAmount,
    address _upperHint,
    address _lowerHint
) public payable {
```

and is `payable`. There's no check to ensure that `_assetAmount` == `msg.value`. In the vast majority of cases this can lead to user losses due to their mistake. If a reentrancy is found, in which some value can be left in BO, then this could be used to escalate that into a critical severity.

Recommendation: See the finding "Suggested Architectural Change to fix to desynch in asset amount and parameter".

Thesis: Fixed in PR 165.

Cantina Managed: Fix verified.

3.4.3 `refinanceWithSignature` is not verifying which rate will be used

Severity: Low Risk

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

Description: `refinanceWithSignature` looks as follows:

- BorrowerOperationsSignatures.sol#L494-L514:

```
function refinanceWithSignature(
    address _borrower,
    bytes memory _signature,
    uint256 _deadline
) external {
    Refinance memory refinanceData = Refinance({
        borrower: _borrower,
        nonce: nonces[_borrower],
        deadline: _deadline
    });

    _verifySignature(
        REFINANCE_TYPEHASH,
        abi.encode(refinanceData.borrower), /// @audit QA: Non Deterministic, you could sign the
        ↪  previous, current or next rate
        _borrower,
        _signature,
        _deadline
    );

    borrowerOperations.restrictedRefinance(refinanceData.borrower);
}
```

There is no check for which rate will be used, this gives the ability to the caller to chose which rate the `borrower` will receive.

Recommendation: Signers should use strict deadlines to avoid their signature being used for the wrong rate.

Thesis: Fixed in PR 175.

Cantina Managed: Fix verified.

3.4.4 `openTroveWithSignature` can be used with an unspecified collateral amount

Severity: Low Risk

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

Description: `openTroveWithSignature` looks as follows:

- BorrowerOperationsSignatures.sol#L363-L403:

```
function openTroveWithSignature(
    uint256 _debtAmount,
    address _upperHint,
    address _lowerHint,
    address _borrower,
    address _recipient,
    bytes memory _signature,
    uint256 _deadline
) external payable {
    OpenTrove memory openTroveData = OpenTrove({
        debtAmount: _debtAmount,
        upperHint: _upperHint,
        lowerHint: _lowerHint,
        borrower: _borrower,
        recipient: _recipient,
        nonce: nonces[_borrower],
        deadline: _deadline
    });

    _verifySignature( /// @audit Doesn't specify the value, so it can result in unintended ICR
        OPEN_TROVE_TYPEHASH,
        abi.encode(
            openTroveData.debtAmount,
            openTroveData.upperHint,
            openTroveData.lowerHint,
            openTroveData.borrower,
            openTroveData.recipient
        ),
        openTroveData.borrower,
        _signature,
        openTroveData.deadline
    );

    borrowerOperations.restrictedOpenTrove{value: msg.value}(
        openTroveData.borrower,
        openTroveData.recipient,
        openTroveData.debtAmount,
        openTroveData.upperHint,
        openTroveData.lowerHint
    );
}
```

There's no specific `collateralAmount` which is decided by the caller. This can allow a caller to open a trove with any amount they choose. This could for example allow a front-runner to open a trove with a lower than intended amount of collateral.

Recommendation: Add a specific check for the collateral amount.

Thesis: Fixed in PR 168.

Cantina Managed: Fix verified.

3.4.5 Refactor: `BorrowerOperationsSignatures` should not have hint signed by the borrower

Severity: Low Risk

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

Description: Liquity V1's hint system is easily griefable to either cause a tx to revert or consume more gas. Many `structs` in `BorrowerOperationsSignatures` have `upperHint` and `lowerHint`. Generally speaking, permit like signatures will have a short delay between them being signed and the tx happening. However, given how quickly hints can be incorrect, having them signed in each permit-like request opens up to griefing and reverts due to hints becoming stale.

Recommendation: Remove hints from the signatures, allow the caller to pass fresh hints as they choose.

Thesis: Fixed in PR 166.

Cantina Managed: Fix verified.

3.4.6 Must accrue first in Liquidations - delayed call to `updateDefaultPoolInterest` can avoid triggering RM liquidations

Severity: Low Risk

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

Description: The SP will call `updateDefaultPoolInterest` after checking for RM, meaning the logic is incorrect. Since I can't find any rational reason for this that benefits a caller, and since `updateDefaultPoolInterest` can be called externally. I think this has a Low impact.

Recommendation: Call `updateDefaultPoolInterest` before checking if the system is in RM.

Thesis: Should be handled by a combination of PR 176. We accrue system interest at the top of `batchLiquidateTroves`.

Thesis: Fixed in PR 176.

Cantina Managed: Fix verified.

3.4.7 Economic concerns around the PCV

Severity: Low Risk

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

Description: The PCV is effectively an idle position that can be managed. My assumption in writing this is that the position will be managed slowly and won't be able to participate in sophisticated MEV activities. Given these assumption 3 interesting economic consideration arise:

1. The PCV is reducing the yield of other stakers.

   This is creating an economic incentive not to use the SP, as the yield from liquidations has to be split with the PCV. I believe this can be seen in THUSD where the PCV is 99% of the depositors in the SP.

2. Management of the PCV can cause insolvency in the system.

   If the PCV was always staked in the SP and rebalanced and no fees were taken, then the PCV could be viewed as a tool to generate some fees for the protocol. However, the PCV can withdraw the debt and also has a mechanism to take a fee. This can cause scenarios of insolvency for the broader system. In order to discuss these, we first have to agree that the PCV is effectively an unbacked position. Where each liquidation then retroactively backs each musd with a (ideally) 1.X to 1 backing of Collateral.

   This backing could actually break to a 0.X to 1 if:

   - The price of the collateral changes too quickly when trying to rebalance it.
   - The liquidation is unprofitable due to the CR dropping below 100.
   - The admin makes a mistake in processing the musd that the PCV contains (e.g. they claim fees on the musd that is "retroactively backed").

   All of these scenarios are possible over time, and become a lot more dangerous if an attacker is able to find a way to profitably self-liquidate (see spamming `refinance`).

   It's important you keep in mind that:

   - SP Liquidations can be unprofitable, meaning the PCV unbackedness can increase over time (low likellihood).
   - The PCV needs to quickly rebalance from Coll back to musd as to ensure it's backed (medium likelihood).
   - The admin has to separate between the musd and coll generated by the loan (should not be withdrawable), vs the excess fees (can be withdrawn or ideally are used to repay the debt faster).

   Once the debt has been repaid, the system wide invariant of backedness can no longer be broken. Whereas, until the debt has been repaid, the PCV can cause the system to be fully unbacked based on a combination of the above. For those reasons, these ratios should be monitored over time to ensure no insolvency is introduced.

3. The PCV is a passive LP in the SP, and the SP can have unprofitable liquidations.

   If we assume the PCV to be passive, we can imagine a scenario in which the other SP stakers are all active (perhaps through a managed position that aggregates their deposits), this allows the following attack:

   - For all liquidations that will be profitable, stake as intended.
   - For all liquidations that will be unprofitable, immediately withdraw and let the PCV take the loss as the sole passive depositors in the SP.

   If sufficient incentives were present this type of behaviour is a winning one, and I believe a rational actor would perform this.

Recommendation: Reflect on the above, discuss them with other Economic advisors.

Thesis: Acknowledged.

> "The PCV is reducing the yield of other stakers. This is creating an economic incentive not to use the SP, as the yield from liquidations has to be split with the PCV. I believe this can be seen in THUSD where the PCV is 99% of the depositors in the SP".

This is definitely the case. Depending on how large the protocol grows (billions of dollars?) $100m might be low, so folks might show up anyway. Further, over time, the SP might drain from liquidations (especially large ones). The default behavior is for governance to withdraw their earned BTC, sell it back for MUSD, and redeposit, but if that doesn't happen, the pool will slowly drain and buying in will be easier.

> "Management of the PCV can cause insolvency in the system".

Totally correct. More abstractly, every time the PCV earns fees, half of those fees are used to repay the $100m loan. Every time this happens, the system has less money available to pay back debts than existing debt. This means that outstanding loans need to be paid by newly created debt.

> "The PCV is a passive LP in the SP, and the SP can have unprofitable liquidations".

Also totally correct. I don't expect coordinated action like this to happen, and we'll also have a bot monitoring the chain that liquidates troves the moment they dip below 110% ICR. There's no grace period, our oracle updates quickly, and our block times are fast (especially relative to bitcoin's).

More broadly, we're putting a public document together that'll detail how our governance / PCV will behave in the important edge cases. It's not going to be automated on-chain on day 1, but we're working towards it

Cantina Managed: Acknowledged.

3.4.8 Advisory on Liquity V1's SP

Severity: Low Risk

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

Description: Please be wary that Liquity V2 is being re-audited and re-deployed because the SP had a bug. That said it also contained a fix to a low likelihood bug for V1 (StabilityPool.sol#L521-L544). I have extensivelly fuzzed Liquity V1's SP and have never ran into a scenario in which the fix was necessary.

Proof of Concept: See StabilityPool.sol#L521-L544.

Recommendation: I highly recommend you research the advisory, discuss internally, test the code and then decide what to do.

Thesis: Fixed in PR 196.

Cantina Managed: Fix verified.

3.4.9 `getDebt` is a stale value

Severity: Low Risk

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

Description: We need to review deeper implications. Liquity V1 also had a lot of stale values, which can break invariants.

Proof of Concept: See ActivePool.sol#L133-L135:

```
function getDebt() external view override returns (uint) {
    return principal + interest;
}
```

Recommendation: Virtual accruals are generally best and will have to be accompanied by testing for all cases Invariant testing or Halmos can help get this done fairly elegantly.

Thesis: Fixed in PR 201.

Cantina Managed: Fix verified.

3.4.10 Discrepancy between `msg.value` and `assetAmount`

Severity: Low Risk

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

Description: The function `addCollWithSignature` is forwarding `msg.value` byt passing `addCollData.assetAmount` which can be different.

- BorrowerOperationsSignatures.sol#L208-L218:

```
borrowerOperations.restrictedAdjustTrove{value: msg.value}(
    addCollData.borrower,
    addCollData.borrower,
    msg.sender,
    0,
    0,
    false,
    addCollData.assetAmount,
    addCollData.upperHint,
    addCollData.lowerHint
);
```

There's an additional Instance at BorrowerOperations.sol#L704-L723:

```
function restrictedAdjustTrove(
    address _borrower,
    address _recipient,
    address _caller,
    uint256 _collWithdrawal,
    uint256 _mUSDChange,
    bool _isDebtIncrease,
    uint256 _assetAmount,
    address _upperHint,
    address _lowerHint
) public payable {
    _requireCallerIsAuthorized(_borrower);
    ContractsCache memory contractsCache = ContractsCache(
        troveManager,
        activePool,
        musd,
        interestRateManager
    );

    contractsCache.troveManager.updateSystemAndTroveInterest(_borrower);
```

Recommendation: Ensure that the value forwarded matches the `assetAmount`.

Thesis: Fixed in PR 165.

Cantina Managed: Fix verified.

3.5 Gas Optimization

3.5.1 Gas Optimizations for `TroveManager`

Severity: Gas Optimization

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

Description: `hasPendingRewards` checks for trove being active (TroveManager.sol#L935-L936):

```
if (hasPendingRewards(_borrower)) { /// @audit Gas: Can remove the
    _requireTroveIsActive(_borrower); /// Check here
```

`hasPendingRewards` already calls `_requireTroveIsActive`. Call to `updateDefaultPoolInterest` is done in `batchLiquidateTroves` (TroveManager.sol#L1026-L1037):

```
function _redistributeDebtAndColl(
    IActivePool _activePool,
    IDefaultPool _defaultPool,
    uint256 _principal,
    uint256 _interest,
    uint256 _coll
) internal {
    if (_principal == 0 && _interest == 0) {
        return;
    }

    updateDefaultPoolInterest(); /// @audit seems like it's done in `batchLiquidate`
```

Thesis: Fixed in PR 179.

Cantina Managed: Fix verified.

3.6 Informational

3.6.1 Suggested Architectural Change to fix to desynch in asset amount and parameter

Severity: Informational

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

Description: Assets and msg.value can be desynched in a few parts of the codebase. This can be mitigated by removing the parameter and validating the msg.value == param in the Signatures check.

Recommendation:

- Remove the parameter.
- Make it one of the first thing declared in each function off of `msg.value`.
- Use only the `param` everywhere.
- Change permit logic to check the `msg.value == param` in every call.

This is probably the simplest.

Thesis: Fixed in PR 165.

Cantina Managed: Fix verified.

3.6.2 Refactor: remove `assert(vars.compositeDebt > 0)`

Severity: Informational

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

Description: The call to `assert(vars.compositeDebt > 0)`. Is unnecessary since you have `_requireAtLeastMinNetDebt` above.

Recommendation: Delete the redundant check.

Thesis: Fixed in PR 163.

Cantina Managed: Fix verified.

3.6.3 Missing event in PCV (TODO left in code)

Severity: Informational

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

Description: PCV.sol#L308-L309:

```
// TODO Emit event
```

Recommendation: Add the event.

Thesis: Fixed in PR 162.

Cantina Managed: Fix verified.

3.6.4 Suggested Next Steps

Severity: Informational

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

Description: These are recommendations based on my experience working on a DAO for 3 years, as well as being an SR for the past 2. My experience is condensed in this video which I highly recommend watching this video.

Per our original discussion the goal of the team was to launch right after the audit. This type of process indicates that you had a reasonable expectation that no major bugs would be found.

After around 2 weeks we have identified quite a few bugs that would have caused the system to reach a state of insolvency and broken accounting. In spite of the extensive testing done many of these issues have been missed, I believe for 2 key reasons:

1. The testing would have required modelling for these specific cases that don't fit an happy path.
2. The Liquity design makes each new test extremely verbose and tedious.

While 1) can be attributed to a lack of adversarial thinking, 2) is a tradeoff of the Liquity model (multi contract that ping-pong calls) that is rarely spoken about.

My point is if it was hard to find these bugs, and the "deck is stacked against" finding more, you should re-evaluate your security process as to reduce the likelihood that something else will go through.

Recommendation: Given the above, and the fact that the code is effectively specced out. 2 key security exercises are missing:

1. Invariant Testing, to thoroughly explore edge cases, and review them to ensure that they have a limited impact.
2. A Security Contest or a Guarded Launch with a Bug Bounty, as to ensure that your security process was thorough and didn't have blind spots.

In security reviews, the biggest issue we can face is having a false sense of security, that stems by "proxy for security", the only usable metric we have is the impact and the complexity of the bugs that were found in the last review.

Given this, I believe that via Invariant Testing a non negligible number of bugs would have been found automatically.

Whereas a contest will help ensure that once you think the code is ready to go live (i.e. you believe it's so unlikely for the code to lose funds that you're willing to have 100 MLN TVL in it), you get the different perspectives of many motivated researchers.

If you could only pick one, you should do the invariant testing in-house and do a paid contest, the downside of skipping the tests is that if too many bugs are found in the contest you'll find yourself in this same spot.

Whereas the obvious tradeoff of doing a lot of engagements is the price.

Update: The team has implemented invariant testing: `https://github.com/mezo-org/musd/tree/echidna/solidity/contracts/echidna`.

And is organizing a security contest for the codebase.

Thesis: Acknowledged.

Cantina Managed: Acknowledged.

3.6.5 Loose Collection of Invariants

Severity: Informational

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

Summary: These are quick notes I wrote while doing the review. I believe all of these should be implemented in invariant tests. If you prefer, you could use Halmos with symbolic storage to test most of these (although I fear Halmos will timeout on the codebase).

- After all accruals the sum of debt and interest matches the one in the Interest Rate Manager.
  - Sum of all debt invariant - BROKEN due to compounding interest (TO TEST).
- Sum of Interest in AP and Default Pool vs Sum of interest for all troves:
  - You cannot have a trove with 0 principal and some interest. Because you always pay the interest first.
- You can never get 0 total stakes, after one trove was opened:
  - `updateSystemInterest` only mints debt from when a rate is actually used, It will not mint debt from 0 to start. From manual review this looks good, but testing is better.
- SortedTrove Insertion can never run out of gas when there are just a few troves:

    - Pools, solvency invariant. For all pools, the assets in the pool are gte than the storage value.

- Repaying debt always zeros out the interest first: After any repayment, if the debt is below the interest (+ pending) then the interest is reduced. If the repayment is above, then the debt in the trove is equal to the `left = principal + interest - repayment`. The interest is 0. The principal is the result of left.

Thesis: Acknowledged.

Cantina Managed: Acknowledged.

3.6.6 Collection of QA Findings

Severity: Informational

Context: *See each particular case below*

- QA: Technically it's 365.25 (InterestRateMath.sol#L6-L7):

```
uint256 private constant SECONDS_IN_A_YEAR = 365 * 24 * 60 * 60; /// @audit QA: Technically it's 365.25
```

- QA - Could use constant BPS = 10_000 for legibility (InterestRateMath.sol#L17):

```
(10000 * SECONDS_IN_A_YEAR); /// @audit Interest rate in BPS?
```

- QA: These are not the latest dependencies: (package.json#L47-L51):

```
"dependencies": {
    "@openzeppelin/contracts": "^5.0.2",
    "@openzeppelin/contracts-upgradeable": "^5.0.2",
    "solidity-coverage": "^0.8.12"
}
```

- QA: `governanceTimeDelay` being one and immutable could be an issue: You may want different delays per operations e.g. a revoke should be faster than enabling. Also a 1 hour delay may not be good long term but useful at the beginning. I think this is ok, but can be a bit of a nuisance.

- QA - This pattern prevents bulk removal: (MUSD.sol#L175-L201):

```
function startRevokeBurnList(address _account) external onlyOwner {
    require(burnList[_account], "Incorrect address to revoke");

    // solhint-disable-next-line not-rely-on-time
    revokeBurnListInitiated = block.timestamp;
    pendingRevokedBurnAddress = _account;
}

function cancelRevokeBurnList() external onlyOwner {
    require(
        revokeBurnListInitiated != 0,
        "Revoking from burn list is not started"
    );

    revokeBurnListInitiated = 0;
    pendingRevokedBurnAddress = address(0);
}

function finalizeRevokeBurnList()
    external
    onlyOwner
    onlyAfterGovernanceDelay(revokeBurnListInitiated)
{
    burnList[pendingRevokedBurnAddress] = false;
    revokeBurnListInitiated = 0;
    pendingRevokedBurnAddress = address(0);
}
```

  This could be an issue if you need to deprecate the system or if you extend the system (e.g. PSM) and need to revoke those.

- PCV should not take a split of fees if the `feeRecipient` is address 0 (PCV.sol#L106-L111):

```
if (feeRecipient != address(0) && feeSplitPercentage > 0) { /// @audit QA: if this fails the fees are
↪  lost
    require(
        musd.transfer(feeRecipient, feeToRecipient),
        "PCV: sending mUSD failed"
    );
}
```

- QA - Seems like we shouldn't be claiming fees before the whole debt is repaid (PCV.sol#L79-L82):

```
function payDebt( /// @audit TODO I Feel it should first repay the debt and then allow cashing out |
↪  This is currently allowing for odd behaviour
    uint256 _musdToBurn
) external override onlyOwnerOrCouncilOrTreasury {
    require(
```

  I think when it comes to fee streams, the logic as is is ok. But if you allow withdrawing from the SP, then this logic is pretty risky as you could (either mistakenly or due to private key leak) effectively:

  - Mint `100e6` musd.
  - Withdraw from the SP.
  - Claim 50% (unbacked tokens) to the recipient.

- QA: Use `calculateDebtAdjustment` (InterestRateManager.sol#L214-L230):

```
function calculateDebtAdjustment(
    uint256 _interestOwed,
    uint256 _payment
)
    public
    pure
    returns (uint256 principalAdjustment, uint256 interestAdjustment)
{   /// If payment more than interest, hit the principal, else interest only
    if (_payment >= _interestOwed) {
        principalAdjustment = _payment - _interestOwed;
        interestAdjustment = _interestOwed;
    } else {
        principalAdjustment = 0;
        interestAdjustment = _payment;
    }
}
```

- QA: Remove `nonce` from the structs in `BorrowerOperationsSignatures` (BorrowerOperationsSignatures.sol#L20-L27):

```
struct AddColl {
    uint256 assetAmount;
    address upperHint;
    address lowerHint;
    address borrower;
    uint256 nonce;
    uint256 deadline;
}
```

- QA: Unused `uint256 public constant MAX_BORROWING_FEE = (DECIMAL_PRECISION * 5) / 100; // 5%`.

  Thesis Fixed in PR 183.

- Liquidations can become unprofitable due to coll incentive:

```
uint256 collToLiquidate = singleLiquidation.entireTroveColl -
```

  Even by a little. Something faily underexplored is the fact that the SP in normal mode is taking on any liquidation. The Coll/debt is pro-rata of Coll/debt meaning that in some scenarios the value of the collateral can be below the value of the debt. I think the incentives to prevent this are there, although I find it pretty interesting how the SP is taking the L on normal liquidations, while Troves are taking the L during RM.

I'm thinking this is because the system expect RM to be a very short transitory period, whereas having possible bad debt in Normal Mode is somewhat intended. I guess this can be one of the downsides of removing RM as now you can have people that are unwilling to ever stake in the SP since they would have to accept a non negligible loss, whereas with the RM logic that loss is socialized to other Troves and is unavoidable.

- Breaks CEI (TroveManager.sol#L418-L422):

```
contractsCache.activePool.sendCollateral( /// @audit TODO: Breaks CEI
    address(contractsCache.pcv),
    totals.collateralFee
);
```

  The call should be moved below to maintain CEI compliance.

Thesis:

- Technically it's 365.25: PR 184.
- Could use constant BPS = 10_000 for legibility: PR 184.
- These are not latest dependencies: *(unaddressed)*.
- governanceTimeDelay being one and immutable could be an issue: PR 192.
- This pattern prevents bulk removal: PR 192.
- PCV should not take a split of fees if the feeRecipient is address 0: See comment 240f64a4.
- Seems like we shouldn't be claiming fees before the whole debt is repaid: **won't fix**.
- Use calculateDebtAdjustment: PR 191.
- Remove nonce from the structs in BorrowerOperationsSignatures: PR 160.
- Unused: PR 183.
- Liquidations can become unprofitable due to coll incentive: **won't fix**.
- Breaks CEI: PR 190.

Cantina Managed: Fixes verified.

3.6.7 High Level: Discrepancy between who pays in `BorrowerOperationsSignatures`

Severity: Informational

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

Description:

- Burn → Borrower.
- Receives → Receiver.
- Pays with Coll → Caller.
- Pays with Debt → Borrower.

This is not an issue per se but I think it will inform how you write zaps. For eBTC we ended up having the caller pay and receive since that's what a zap would do. I think your choices are fine but you should fully document them and decide if these will be great once you start rolling out the protocol.

Recommendation: Document and standardize the token flows based on your needs.

Thesis: In PR 153 we generalized how this works. The signer specifies a receiver (who could be themselves or the caller) that receives any mUSD or BTC emitted from the operations. I think that makes it flexible enough that we can accomplish whatever set of behavior we want.

Cantina Managed: Fix verified.