# Threshold Nework

## Security Review

Cantina Managed review by:
**Kurt Barry**, Lead Security Researcher
**Alex the Entreprenerd**, Security Researcher
**Lucas Goiriz**, Junior Security Researcher

June 12, 2023

# 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** | *Directly* exploitable security vulnerabilities that need to be fixed. |
| **High** | Security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All high issues should be addressed. |
| **Medium** | Objective in nature but are not security vulnerabilities. Should be addressed unless there is a clear reason not to. |
| **Low** | Subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues. |
| **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. When determining the severity one first needs to determine whether the finding is subjective or objective. All subjective findings are considered of Minor severity.

Next it is determined whether the finding can be regarded as a security vulnerability. Some findings might be objective improvements that need to be fixed, but do not impact the project's security overall (Medium).

Finally, objective findings of security vulnerabilities are classified as either critical or major. Critical findings should be directly vulnerable and have a high likelihood of being exploited. Major findings on the other hand may require specific conditions that need to be met before the vulnerability becomes exploitable.

# 2 Security Review Summary

Threshold USD (thUSD) is a stablecoin soft-pegged against USD and backed by ETH and tBTC as collaterals with a minimum collateral ratio of 110%. Protocol-wise, it is a modified fork of Liquity Protocol built to be self-sustained through a PCV ("Protocol Controlled Value"), where all profits accrue into the PCV.

From May 22nd to Jun 9th the Cantina team conducted a review of Threshold USD on commit hash 6399a8. The team identified a total of **54** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 1
- Medium Risk: 3
- Low Risk: 12
- Gas Optimizations: 12
- Informational: 26

# 3 Findings

## 3.1 High Risk

### 3.1.1 `_requireValidAdjustmentInCurrentMode` bypass when not in `mintList` allows to run away with collateral without repaying

**Severity:** High Risk

**Context:** BorrowerOperations.sol#L555-L557

**Description:** `adjustTrove` is used to update the state of a Trove, invariants for collateralization are enforced in `_requireValidAdjustmentInCurrentMode`.

Due to the integration with `mintList` if the system is no longer part of the `mintList` it will not perform critical checks for the system collateralization.

The following code snippet extracted from BorrowerOperations.sol#L542-L557:

```
function _requireValidAdjustmentInCurrentMode
(
    bool _isRecoveryMode,
    uint256 _collWithdrawal,
    bool _isDebtIncrease,
    LocalVariables_adjustTrove memory _vars
)
    internal
    view
{
    /*
     * If contract has been removed from the thUSD mintlist remove the adjustment restrictions // TODO: Can we
↪  just run away once deprecated?
       */
    if (!thusdToken.mintList(address(this))) {
        return;
    }
    // ...
```

The lack of checks allows all depositors to withdraw the vast majority of their collateral without repaying their debt.

- Proof of concept (add this to `BorrowerOperationsTest.js`. Run it with `npx hardhat test --grep "UNDERCOLLATERALIZED"`):

```
it("UNDERCOLLATERALIZED (NO REPAY)", async () => {
  await openTrove({ ICR: toBN(dec(300, 16)), extraParams:{ from: alice } })
  await openTrove({ ICR: toBN(dec(120, 16)), extraTHUSDAmount: toBN(dec(300, 18)), extraParams:{ from: bob } })
  // remove mintlist
  await th.removeMintlist(contracts, owner, delay)

  // to compensate borrowing fees
  await thusdToken.transfer(alice, dec(300, 18), { from: bob })
  const collToWithdraw = await troveManager.getTroveColl(alice)

  // As defined in Dependencies/LiquityMath.sol
  const NICR_PRECISION = toBN('1000000000000000000');
  const debt = await troveManager.getTroveDebt(alice);

  /** Given that the NICR is computed as:
   * if (_debt > 0) {
   *     return _coll * NICR_PRECISION / _debt;
   * }
   *
   * We can compute the amount of collateral to withdraw to get a NICR of 1 (because of floor division)
   * Thus _coll = collInside - whatWeWantToWithdraw, then...
   *
   * whatWeWantToWithdraw = collInside - (debt / NICR_PRECISION)
   */

  const optimalWithdrawalAmnt = toBN(collToWithdraw.toString()).sub(
    toBN(debt.toString()).div(NICR_PRECISION)
  );

  const txAlice = await borrowerOperations.withdrawColl(optimalWithdrawalAmnt, ZERO_ADDRESS, ZERO_ADDRESS, {
↪  from: alice })
  assert.isTrue(txAlice.receipt.status)
})
```

As shown, Alice can withdraw all collateral (except a small dust amount, to get her NICR to be greater than 0), which will make the Trove completely undercollateralized. This will in turn affect the peg of THUSD for all deprecated `BorrowerOperations`, putting all of that capital at risk.

**Recommendation:**

- **Short Term**: Maintain the same invariants for deprecated systems (do not have an early return).
- **Long Term**: Reconsider the purpose of the early return and change the check to enforce repayment, but relaxing other requirements.

**Threshold:** Special case removed, same checks for normal and deprecated states (see commit e05abc).

**Cantina:** Confirmed this is fixed in commit e05abc.

## 3.2 Medium Risk

### 3.2.1 Swapper can lose all `thusdAmount` if Price Feed is down

**Severity:** Medium Risk

**Context:** BAMM.sol#L256-L271

**Description:** The `swap` function is meant to allow purchasing of collateral as a way to automate auto-compounding of collateral into more THUSD. To allow the swap, `getSwapCollateralAmount` will query the Chainlink Price Feed for the Collateral / USD pair and use that value to compute the price.

In the case in which the call fails, `getSwapCollateralAmount` will return 0, which means that `msg.sender` will pay the `thusdAmount` but receives no collateral in return, as seen in BAMM.sol#L234:

```
if(collateral2usdPrice == 0) return (0, 0); // chainlink is down
```

The slippage check would prevent this from happening, but a distracted buyer (or an automated strategy) may just use 0, causing the transaction to go through.

**Recommendation:** Consider reverting if the price feed is down to avoid the caller losing funds.

**Threshold:** There is one check for swap function (in commit d5e7a5) that added recently. This is unrelated to suggestion but it will always lead to fail because `minReturn` can't be 0 and `collateralAmount` must be greater than 0.

**Cantina:** Verified that this is fixed with the changes shown in commit d5e7a5.

#### 3.2.2 Inability to change `thusd2UsdPriceAggregator` can lead to DOS or negative side effects

**Severity:** Medium Risk

**Context:** BAMM.sol#L88

**Description:** CL Price feeds are maintained, meaning that they could be deprecated.

Due to the check, the extra feed cannot be removed, and thus in case of deprecation, the BAMM contract will stop working.

Price Feeds could start reverting for following reasons:

- Logical Flaw
- Paused by the Owner (Chainlink Admin)

Price Feeds may also not be deprecated, hence they will return old prices. It would be best to allow to edit the PriceFeed as to allow the BAMM rebalancing to work.

More specifically, because the feed would be only to handle the peg for THUSD vs USD, it may be best to assume the price is 1 dollar (which creates opportunities for arbitrage), when the Feed Reverts.

**Recommendation:** Consider allowing to change the Price Feed or account for reverts, catching them and returning a safe value.

**Threshold:** Added returning safe value (`collateralAmount`, yielding a conversion rate of 1-1 if the CL feed is not working) in commit cc363a.

**Cantina:** It may be best to use a `constant` to flag for the 0 value since that's a flag for the feed being down, but the code in the linked commit cc363a is safe, and returns a conversion rate of 1-1 if the CL feed is not working.

#### 3.2.3 `TELLOR_DIGITS` will be incorrect for newer feeds

**Severity:** Medium Risk

**Context:** PriceFeed.sol#L34-L35

**Description:** The implementation will work only for the ETH / USD Feed (exclusively if Liquity's current Feed is used) but will not work for a BTC / USD Feed as new feeds have 18 decimals of precision

- Proof of concept By checking the following links, we can see how the Liquity Feed has been normalized on purpose by Tellor. So, newer feeds will instead have `18` decimals of precision:
- Etherscan address `0xD9157453E2668B2fc45b7A803D3FEF3642430cC0`:
  - See the implicated code here.
- Etherscan address '0xd3b9a1dcabd16c482785fd4265cb4580b84cded7
  - See the implicated code here.

Here one can see that the adjustment is done exclusively for ETH / USD query. Another Tellor oracle, for example for tBTC, will not be using the same decimals.

**Recommendation:** Either consider making sure that older feeds are used, or changing the code on a deployment-by-deployment basis.

**Threshold:** Changed `constant TELLOR_DIGITS` to `immutable` variable to be able use different decimals with the same code (following the recommendation of changing the code on a deployment-by-deployment basis). See commit 04c683.

**Cantina:** Verified that this is fixed in commit 04c683.

## 3.3 Low Risk

### 3.3.1 If one oracle is permanently disabled, the other cannot fail even briefly, or the price is permanently frozen

**Severity:** Low Risk

**Context:** PriceFeed.sol#L236

**Description:** If the price contract reaches the `bothOraclesUntrusted` state, price updates can only resume if both oracles become live again and report approximately the same price. This means that if one of the oracles goes permanently offline, even a brief outage from the remaining one permanently locks the price at `lastGoodPrice`. Tellor in particular may be vulnerable to temporary outages due to its disputation mechanism.

If this happens, the price will quickly become stale, leading to various negative consequences (depending on whether the real price increases or decreases relative to the "locked-in" price). Fortunately, the probability of both oracles failing, one of them permanently, should be low.

**Recommendation:** Consider implementing a mechanism to allow one of the oracles to eventually become trusted again even if the other never comes back online. For example, requiring valid responses for a certain period of time, or a way for governance to change an oracle back to being trusted under special conditions.

**Threshold:** Added a function to allow the DAO to exit this broken state in commit fc4f49.

**Cantina:** The added function is reasonable; after discussion with the Threshold team agree that this action does not need to be behind a governance delay because: 1) The attack scenario (governance is malicious and has co-opted one of the oracles) is very unlikely compared to needing to exit such a state, and ... 2) a delay could prevent swift action in a case where this function needs to be called.

### 3.3.2 Liquidations Round in Favor of Troves, Against the Stability Pool

**Severity:** Low Risk

**Context:** TroveManager.sol#L414, TroveManager.sol#L440

**Description:** The divisions to calculate the collateral to send to the stability pool in liquidations round against the stability pool. At extreme values (e.g. very low trove debt, very high total debt, and very high collateral price), it's possible for the amount of collateral taken in a liquidation to round to zero. This would potentially allow for the extraction of value by actors that open small troves and liquidate them themselves when prices drop (so that they receive the gas compensation as well). Fortunately, this should not be possible at typical values. Even in an extreme example with a $10^3$ debt trove, $10^{10}$ total debt, and a $10^9$ collateral price, there are still about 5 orders of magnitude safety margin for the calculation on TroveManager.sol#L414. However, governance should be aware of this risk as the system grows and more collaterals are added.

**Recommendation:** Avoid adopting collaterals with extremely high prices. A more robust solution would be to use upwards-rounding division on these lines, but this requires further changes to avoid rounding up to amounts that exceed the collateral of a given trove (which would be unfair to other troves). Thus round-up semantics may add more complexity than they are worth.

**Threshold:** This issue has already been identified and internally discussed to find ways of mitigating such scenarios. The initial approach involves establishing a rigorous diligence mechanism for collateral addition from a governance perspective.

**Cantina:** The stated approach to managing this risk is reasonable.

### 3.3.3 Dangerous `int256` casting to `uint256`

**Severity:** Low Risk

**Context:** BAMM.sol#L129

**Description**: The `fetchPrice()` function in `BAMM.sol` employs Chainlink's pricefeed to retrieve a value of type `int256`. However, without any validation checks, this value is immediately cast to a `uint256` type, potentially leading to a silent overflow in the event that the oracle returns a negative value. Thus, the cast to `uint256` may produce unexpected behavior and introduce subtle bugs or vulnerabilities in subsequent operations.

**Recommendation**: Although it is highly unlikely for the oracle to return negative values in this specific use case, it is advisable to implement a validation step to ensure that the returned value is non-negative. By performing this check before the cast, the code can detect any exceptional cases where negative values might be returned (such as when querying the price feed of futures). This preventive measure will mitigate the risk of silent overflow and help maintain the expected behavior and integrity of `BAMM`.

**Threshold:** Added check for non zero answer on commit 0ac4f4.

**Cantina:** Verified that this is fixed by commit 0ac4f4.

### 3.3.4 Tokens With Non-18 Decimals Are Unsupported

**Severity:** Low Risk

**Context:** LiquityMath.sol#L101 and collateral value handling throughout the codebase.

**Description:** Collateral values are never normalized based on their decimal values before being used in calculations where all other values are 18 decimals and the return value is expected to be 18 decimals (one example highlighted in the context). This means that non-18 decimal collaterals are incompatible with the system. Since governance controls the addition of collaterals, this issue is currently easy to avoid and thus `Low` severity.

**Recommendation:** No action is strictly necessary; if desired, non-18 decimal collaterals can be supported by normalizing collateral input and output amounts to 18 decimals appropriately.

**Threshold:** This has already been discussed internally, and considering that we are aiming only to have tBTC and ETH as collaterals initially, the team decided to add a disclaimer in the docs about this information. And If for instance, a new collateral with a different amount of decimals is to be added to the system, this validation can be added to the new deployment set of contracts.

**Cantina:** Sounds good, documentation will be beneficial to future governance participants.

### 3.3.5 Multiple BAMM and Stability Pools could cause a liquidity crunch

**Severity:** Low Risk

**Context:** BAMM.sol#L169

**Description:** Multiple Stability Pools may cause a lack of liquidity during liquidations. This could be caused due to BAMM Stakers not moving from one BAMM to another during periods of volatility. For Example:

- *Period A*
  - BTC price is extremely flat.
  - ETH price is more volatile.
  - All BTC minters end up staking in the ETH Stability Pool because the APR is Higher.
- *Period B*
  - BTC starts having liquidations (one day of excessive volatility).
  - Because stakers need time to move their funds (and they have to perform this themselves), liquidity is insufficient to handle the liquidation, causing debt and collateral redistribution.

This can create a scenario in which some liquidations may not be performed optimally, resulting in potentially more bad debt being formed.

**Recommendation:** It is worth weighing whether it would be best to have 1 SP that can handle liquidations on multiple systems, or whether the separate system is resilient enough. A yield farming vault that moves THUSD from one Stability Pool to the other may also be a way to offer higher APR to stakers as well as ensure that liquidity is available when necessary.

**Threshold USD:** It's economically feasible in this model to take "ever-larger loans", but it's not against future earnings per se.

The Initial Protocol Loan is scaled to a size intended to cover all cases. Currently set at 100M thUSD. It could be set at 1B, 10B, 100B or higher, the number is irrelevant and imaginary.

What truly happens is this:

When a liquidation occurs, roughly ~10% is profit to the stability pool (and thus the PCV). The stability pool will exchange thUSD for collateral worth ~10% more than the thUSD provided. This is instant profit to the PCV. Due to this, it can absorb an endless supply of collateral while taking no loss. E.g a 100M liquidation would wipe out the stability pool, but leave PCV with ~$10M in more value (profit). We could just as easily have seeded the pool with 1B if more appropriate.

The main risk (of loss) becomes the balancing act rather than the draw from stability pool itself. With ~10% profit, there is room for ~10% fluctuations to the downside until the balancing is complete before the PCV runs at a loss. In bitcoin, 10% movement usually take days. Given that BAMM is set up to constantly offer liquidated collateral for sale, it's intended that the collateral will be returned back to PCV as thUSD in due time, with exceptions being rare circumstances of very high fluctuation in the markets / black swans.

B. Protocol's own analysis shows that these events will average out over time, but, in rare instances where the PCV would incur a real loss (debt) such as during large liquidation and a fast dropping market, it would first cancel out debt against already built up profits incurred in the protocol (from loans, earlier liquidations, redemptions), and if loss exceeds this sum, then no withdraws can be made from the PCV until debts are repaid, and this promise of future earnings is where we can ask what limits the market is willing to accept before losing trust in the peg. I argue, quite a lot, if we look at fractional reserve stablecoins.

As stated earlier, PCV is the main depositor in stability pool and there's no yield farming. Since all profits from loans accrue into the PCV, it will have a steady stream of thUSD entering the pools and the IPC will work as sufficient buffer in times of high volatility.

BAMM offers collateral up for sale, naturally arbitragers should step in to fulfill this order once profitable. Liquidity would need to be strong in either the tBTC / thUSD pair and ETH / thUSD pair or by proxy thUSD/USDC pair, tBTC/WBTC pair or similar.

I expect these pairs (tBTC/WBTC and thUSD/stablecoin) to be commissioned by the DAO with yield farming APR's to incentivize sufficient liquidity.

**Cantina:** Threshold team's plans are deemed adequate to mitigate any concerns regarding liquidity crunches.

#### 3.3.6 `closeTrove` can create a Bank Run when `canMint` is set to false

**Severity:** Low Risk

**Context:** BorrowerOperations.sol#L365-L368

**Description:** Recovery Mode exists with the goal of bringing the TCR back up to a higher value. In the original Liquity Model, during Recovery Mode liquidations can be performed at a higher CR, and Debt Redistribution is performed by preventing Troves from being closed.

If `canMint` is set to `false`, `closeTrove` will allow closing a Trove even during recovery mode. This may cause a bank run, in which early withdrawers can avoid getting bad-debt being redistributed to them, while slower withdrawers will be forced to take on those losses.

- Proof of concept:
  - Anyone can call `closeTrove` for their own Trove.
  - Then call `batchLiquidateTroves` to trigger the bad-debt redistribution.
  - Other Troves will receive the Bad Debt, the caller has avoided taking a direct loss.

**Recommendation:** Consider if `closeTrove` should be allowed once a deployment is deprecated. It may be best to ensure that deprecation happens gradually, reducing TVL and as such reducing the impact of bad debt.

**Threshold:** I believe "Undercollateralized Troves" will be a very rare event. In normal circumstances a trove is liquidated at 110%, which means the market has to drop an additional 10% from the time it reaches liquidation threshold to undercollaterialization state. Since incentives are put in place for bots to profit from liquidations I would expect this to occur at 110%, seconds or at most minutes after reaching this threshold.

If for some reason that does not happen, and trove(s) enter the undercollaterialized state, it seems like an acceptable risk to take, in fact I think there's an argument to be made that it's beneficial to "reward" (or at least not punish) people for closing early since that's getting us closer to the goal of deprecating.

**Cantina:** Agreed that the scenario above is low likelihood as it requires multiple external conditions. It's worth noting that from our own research, the fastest ETH had a 10% downturn was in around 4 hours. Those time-frames should be considered safe as long as any bot is calling liquidations through the Stability Pool.

### 3.3.7 Missing stale price validation in chainlink's agreggator `latestRoundData()` call

**Severity:** Low Risk

**Context:** BAMM.sol#L219

**Description:** Chainlink's `AggregatorV3` could return stale prices, which in turn results in wrong price assumptions. It is necessary to verify that the returned price is fresh enough before using it.

**Recommendation:** Consider reusing the logic implemented in `BAMM's fetchPrice()`.

**Threshold:** Fetching a price was unified for both aggregators in commit cc363a.

**Cantina:** Verified that this is fixed by commit cc363a.

### 3.3.8 `receive` function should revert if the collateral is an ERC20 token

**Severity:** Low Risk

**Context:** BAMM.sol#L295, PCV.sol#L241

**Description:** `BAMM.sol` and `PCV.sol` are set with a specific collateral at construction time. This collateral could be an ERC20 token or Ether. In the case that the collateral is set to an ERC20 token, the contract should block the `receive()` function, since any Ether sent to that contract would become locked.

**Recommendation:** Consider blocking the `receive()` function, as shown below:

```
- receive() external payable {}
+ receive() external payable {
+     require(address(collateralERC20) == address(0), "ETH receive not allowed");
+ }
```

**Threshold:** Added suggested checks in BAMM (at commit cf9b0d and PCV (at commit d7076b), plus posterior error string renaming (at commit 00b730).

**Cantina:** Verified that the changes have been correctly applied at commits cf9b0d, d7076b and 00b730.

### 3.3.9 `BAMM.withdraw` may burn shares but return nothing

**Severity:** Low Risk

**Context:** BAMM.sol#L188-L191

**Description:** The `withdraw()` function contains the following check:

```
if(thusdAmount > 0) thusdToken.transfer(msg.sender, thusdAmount);
```

which is to account for the case of a 0 value transfer. In that case, it may be best to just revert as the shares would be burned for nothing in return.

**Recommendation:** Consider reverting instead.

### 3.3.10 Tellor Caller may use older values

**Severity:** Low Risk

**Context:** TellorCaller.sol

**Description:** Due to Tellor Disputes, it is possible for the Price Feed to use older values if the newer ones are disputed (this is explained in the Tellor Docs).

To mitigate this issue, Tellor has created a demo repository that ensures values have had time for disputes, and caches the last valid value to avoid Dispute Attacks.

**Recommendation:** Consider employing the suggested version (TellorCaller.sol) instead.

**Threshold:** Used suggested version of TellorCaller. See commit 9d5890.

**Cantina:** Verified that the code has been changed at commit 9d5890 to use the best practice of caching the last accepted answer and return it in case Tellor starts returning older values, mitigating this way dispute attacks.

### 3.3.11 Cannot use `USDT` and similar

**Severity:** Low Risk

**Context:** SendCollateral.sol#L22

**Description:** The function `IERC20.transfer`

```
bool success = _collateralERC20.transfer(_recipient, _amount);
```

uses an interface that assumes that the ERC20 will return a `bool`. For some tokens, such as `USDT` that do not return anything, this will cause the call to revert (because solidity performs safety checks based on the interface). Since governance controls the addition of collaterals, however, this is relatively easy to avoid in practice if there is awareness of it.

- Proof of concept (works on any Foundry Repository):

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.19;

import {Test} from "forge-std/Test.sol";

interface IERC20 {

    function transfer(address, uint256) external returns (bool);
}

contract USDTLike {
    uint256 balance;
    function transfer(address, uint256 amount) external {
        balance += amount;
        // Return nothing
    }
}
contract UsdtRevertDemo is Test {

    IERC20 usdt;
    function setUp() public {
        usdt = IERC20(address(new USDTLike()));
    }

    function test_usdt() public {
        usdt.transfer(address(1), 123);
    }
}
```

**Recommendation:** Consider using `safeTransfer` or ensuring each token you use in the future is compatible with the direct usage of `transfer returns (bool)`.

**Threshold:** `SendCollateral` updated to use `SafeERC20` library in commit 21fefc.

**Cantina:** Verified that this is fixed in commit 21fefc by using `safeTransfer`.

### 3.3.12 Redemptions Can Be Prevented Due to an Unsound Assertion

**Severity:** Low Risk

**Context:** TroveManager.sol#L915

**Description:** TroveManager.sol#L915 asserts that the THUSD balance of the caller of `redeemCollateral` is less than the total debt drawn against collateral type being redeemed; this is no longer an invariant in a multicollateral deployment (in fact, the existence of the PCV loan technically violates the invariant as well). In some cases, redemptions may unexpectedly fail as a result. The worst-case scenario would be a smart contract design to redeem THUSD for collateral; in an admittedly costly griefing, an attacker could donate sufficient THUSD to it to prevent it from redeeming against its target.

**Recommendation:** Consider removing this unsound assertion.

**Threshold:** Assertion removed in commit 1420b3.

**Cantina:** Confirmed this was fixed in commit 1420b3.

## 3.4 Gas Optimization

### 3.4.1 `pcvAddress` and `pcv` stores the same value in storage and one can be removed

**Severity:** Gas Optimization

**Context:** BorrowerOperations.sol#L27-L28

**Description:** Within `BorrowerOperations.sol` there are two variables (`pcvAddress` and `pcv`) that represent the same stored value since the casting `PCV(pcvAddress)` has not runtime impact.

**Recommendation:** Consider removing one of such variables and to cast when necessary in the code. This saves one storage slot and thus incurs in gas savings.

**Threshold:** Fixed in commit 58478e.

**Cantina:** Verified this has been fixed in commit 58478e.

### 3.4.2 Redundant getter functions for the same property

**Severity:** Gas Optimization

**Context:** CropJoinAdapter.sol#L11, CropJoinAdapter.sol#L18-L20

**Description:** `CropJoinAdapter` includes two getter functions for the `total` property: `totalSupply()` and `total()`, leading to redundancy and potential confusion. `totalSupply()` is explicitly defined by the user, while `total()` is automatically generated by the Solidity compiler due to the property being defined as `public`. This duplication of functionality increases code complexity, incurs in a slightly larger creation bytecode size and slightly higher deployment gas costs.

**Recommendation:** Consider removing one of the getter functions. The simplest approach is declaring `total` as `internal` instead of `public`, and ensuring that all callees within the codebase are using `CropJoinAdapter.totalSupply()` instead of `CropJoinAdapter.total()`.

**Threshold:** Fixed in commit 58478e.

**Cantina:** Verified that this is fixed by declaring `total` as `internal` in commit 58478e.

### 3.4.3 Replace division by multiples of 2 for right bit shifts for gas savings

**Severity:** Gas Optimization

**Context:** LiquityMath.sol#L40, LiquityMath.sol#L74, LiquityMath.sol#L78

**Description:** `LiquityMath` currently employs division by a multiple of 2 using the division operator (`/`). Given that Solidity performs integer division, this operation is equivalent of performing an appropriate amount of right bit shifts (`>>`), being the latter slightly more gas efficient.

**Recommendation:** Consider replacing all instances of `/ 2` by `>> 1` for minor gas savings. See below:

- LiquityMath.sol#L40

```
- decProd = (prod_xy + (DECIMAL_PRECISION / 2)) / DECIMAL_PRECISION;
+ decProd = (prod_xy + (DECIMAL_PRECISION >> 1)) / DECIMAL_PRECISION;
```

- LiquityMath.sol#L74

```
- n = n / 2;
+ n = n >> 1;
```

- LiquityMath.sol#L78

```
- n = (n - 1) / 2;
+ n = (n - 1) >> 1;
```

### 3.4.4 Favor local variable usage over redundant storage reads

**Severity:** Gas Optimization

**Context:** SortedTroves.sol#L331, SortedTroves.sol#L355

**Description:** `SortedTroves` unnecessarily reads the same value repeatedly from storage instead of utilizing a locally declared variable, even though both approaches yield the same value. This practice introduces inefficiency and incurs higher gas costs.

**Recommendation:** Consider refactoring the code to utilize the locally declared variable instead of reading said variable again from storage. To achieve this, one could perform the following changes

- SortedTroves.sol#L331

```
- prevId = data.nodes[prevId].nextId;
+ prevId = nextId;
```

- SortedTroves.sol#L355

```
- nextId = data.nodes[nextId].prevId;
+ nextId = prevId;
```

### 3.4.5 Basis Point Range Checks Can Be Removed In BAMM

**Severity:** Gas Optimization

**Context:** BAMM.sol#L201-L202

**Description:** The function `addBps` is invoked twice for every swap, so the `require` statements found at BAMM.sol#L201-L202 are evaluated twice. However, the values passed as the `bps` argument are `fee` and `maxDiscount`. The former is validated in its setter method, and the latter is `immutable` (a check could be added for it in the constructor, or contracts deployed with nonsense values by accident can simply be abandoned). Thus checking the `bps` argument range is not strictly necessary.

**Recommendation:** Consider adding a range check for `maxDiscount` in the constructor, and removing the `require` statements in `addBps` to reduce the gas cost of swaps.

### 3.4.6 `_THUSD` argument of `_redeemCloseTrove` could be removed

**Severity:** Gas Optimization

**Context:** TroveManager.sol#L841

**Description:** The function `_redeemCloseTrove()` accepts an argument `_THUSD` which is always passed as the `THUSD_GAS_COMPENSATION` constant. This argument could be eliminated and the constant used directly in the body of the function, which reduces complexity and may also slightly reduce gas costs.

**Recommendation:** Consider removing the aforementioned function argument and employing the `THUSD_GAS_COMPENSATION` constant directly within the function body.

### 3.4.7 Loop optimization opportunities

**Severity:** Gas Optimization

**Context:** PCV.sol#L221-L226, PCV.sol#L234-L239, MultiTroveGetter.sol#L67, MultiTroveGetter.sol#L73, MultiTroveGetter.sol#L96, MultiTroveGetter.sol#L102, PriceFormula.sol#L12,PriceFormula.sol#L38, TellorCaller.sol#L71, TroveManager.sol#L527, TroveManager.sol#L584, TroveManager.sol#L667, TroveManager.sol#L722

**Description:** Under certain safe scenarios, `for` loops in Solidity can be optimized by taking advantage of the following properties:

- Un-initialized unsigned integers are set to `0` by default.
- Caching variables that are read multiple times reduces gas costs as `SLOAD`s are avoided this way.
- Overflow protection can be disabled if the variable to increment has a clearly defined upper bound.
- The prefix increment is the cheapest way to increament a counter.

So, for instance, one can optimize the loop at PCV.sol#L221-L226 this way:

```
  function addRecipientsToWhitelist(address[] calldata _recipients) external override onlyOwner {
+     uint256 length = _recipients.length;
-     require(_recipients.length > 0, "PCV: Recipients array must not be empty");
+     require(length > 0, "PCV: Recipients array must not be empty");
-     for(uint256 i = 0; i < _recipients.length; i++) {
+     for(uint256 i; i < length; ) {
      addRecipientToWhitelist(_recipients[i]);
+     unchecked{ ++i; }
  }
```

Besides the gas savings, another benefit from optimizing `for` loops is that out-of-gas errors will be harder to reach.

**Recommendation:** Where possible, consider implementing these `for` loops optimisations.

### 3.4.8 Redundant `_depositor` function argument in `_updateDepositAndSnapshots`

**Severity:** Gas Optimization

**Context:** StabilityPool.sol#L609-L631

**Description:** The current implementation of `_updateDepositAndSnapshots` includes the `address _depositor` argument, which is set as `msg.sender` in all call sites. This approach introduces unnecessary complexity as the `msg.sender` can be accessed from within the function body.

**Recommendation:** Consider removing the `address _depositor` argument and directly using `msg.sender` within the function body, as in `_sendCollateralGainToDepositor`, to simplify the codebase and reduce gas costs.

### 3.4.9 Unnecessary `struct` type declaration

**Severity:** Gas Optimization

**Context:** StabilityPool.sol#L153-L155, StabilityPool.sol#L164, StabilityPool.sol#L268, StabilityPool.sol#L293, StabilityPool.sol#L319, StabilityPool.sol#L489, StabilityPool.sol#L525, StabilityPool.sol#L610

**Description:** The `Deposit struct` contains only the `initialValue` field, and it is exclusively used in the definition of the `deposits mapping`. Whenever an element of `deposits` is accessed, it only retrieves the value stored in the single field of the `Deposit`. This design is redundant since the `deposits` could directly store the `initialValue` values without the need for an intermediary `struct`, resulting in unnecessary complexity and decreased code clarity. Depending on the compiler optimizer settings, this can also slightly increase contract size (and thus deployment costs) versus the recommendation below--in testing, this was only true when the optimizer was off, however (contract size was identical for the two versions with the optimizer on, at least for the `optimizer_runs` values used).

**Recommendation:** Consider performing the following changes:

- Removing the declaration of the `Deposit struct` (at StabilityPool.sol#L153-L155) altogether.
- Declaring the mapping at StabilityPool.sol#L164 as:

```
mapping (address => uint256) public deposits
```

- Removing all the instances of `.initialValue` (at StabilityPool.sol#L268, StabilityPool.sol#L293, StabilityPool.sol#L319, StabilityPool.sol#L489, StabilityPool.sol#L525, StabilityPool.sol#L610).

**Threshold:** Updated in commit 58478e.

**Cantina:** Verified this is fixed in commit 58478e.

### 3.4.10 Can use immutables by pre-computing addresses

**Severity:** Gas Optimization

**Context:** ActivePool.sol#L38-L45, BorrowerOperations.sol#L86-L98, CollSurplusPool.sol#L31-L36, DefaultPool.sol#L32-L36, HintHelpers.sol#L25-L28, PCV.sol#L66-L71, PriceFeed.sol#L82-L85, StabilityPool.sol#L199-L207, TroveManager.sol#L199-L210, IBorrowerOperations.sol#L28-L41, ICollSurplusPool.sol#L20-L25, IPCV.sol#L32-L37, IStabilityPool.sol#L58-L66, ITroveManager.sol#L49-L60

**Description:** Each address is deterministic since it's the result of the deployer's address and nonce. This means that it is possible to write a simple script that:

1. Pre-computes all addresses in a sequential manner.
2. Attributes each contract a number.
3. Deploys the contracts in the appropriate order.

This approach may require removing some `isContract` checks. Nonetheless, after setup the system would work and it would be using `immutable` addresses, which implies a gas saving of at least 2.1k per transaction.

**Recommendation:** Consider whether the gas savings would be helpful and, if so, refactor the codebase to use `immutable` addresses for the system contracts.

### 3.4.11 Packing Opportunities in TroveManager

**Severity:** Gas Optimization

**Context:** TroveManager.sol

**Description:** `TroveManager` contains information around the Troves. Most tokens (including ETH) do not have a `totalSupply` that comes even remotely close to `uint128`, giving thus opportunities for packing. Below are listed a series of optimizations:

1. **Packing by using** `uint128`: In the `Trove` struct declared at TroveManager.sol#L60-L66, the `debt` and `coll` fields could be declared as `uint128` instead, saving this way one storage slot while still allowing them to store large values ($2^{128} - 1 \approx 10^{38}$):

```
struct Trove {
    uint128 debt;
    uint128 coll;
    uint256 stake;
    Status status;
    uint128 arrayIndex;
}
```

An additional storage slot can be reduced by applying a similar rationale to `stake` (declaring it as `uint128`) and `arrayIndex` (declaring it as `uint120`), while noting that `status` is an `enum` type:

```
struct Trove {
    // Slot 1
    uint128 debt;
    uint128 coll;
    // Slot 2
    uint128 stake;
    Status status;
    uint120 arrayIndex;
}
```

2. Abusing the above rationale again, `L_Collateral` and `L_THUSDDebt` can be packed into one storage slot:

```
uint128 public L_Collateral;
uint128 public L_THUSDDebt;
```

Which would still allow to track around $10^{11}$ billions of dollars, which is plenty at this time.

3. **Packing of time rate**. `baseRate` and `lastFeeOperationTime` could also be declared as smaller unsigned integer types (`uint32` and `uint64` respectively).

**Recommendation:** Consider applying the changes described above to save at least 2.1k to 5k+ gas for each operation.

### 3.4.12 Reuse the constant `BOOTSTRAP_LOAN` to save SLOADs

**Severity:** Gas Optimization

**Context:** PCV.sol#L106-L110

**Description:** The following logic, located at PCV.sol#L106-L110:

```
debtToPay = BOOTSTRAP_LOAN;
borrowerOperations.mintBootstrapLoanFromPCV(debtToPay);

isInitialized = true;
depositToBAMM(debtToPay);
```

can be changed in such a way that it uses `BOOTSTRAP_LOAN` multiple times, which will saved `SLOAD`s.

**Recommendation:** Consider applying the aforementioned refactoring to save 200 gas.

**Threshold:** Updated in commit 58478e.

**Cantina:** Verified this is fixed by replacing the instances of `debtToPay` within `depositToBAMM()` for `BOOTSTRAP_LOAN` as suggested, in commit 58478e.

## 3.5 Informational

### 3.5.1 `vars.THUSDFee` does not have any effect and could be removed

**Severity:** Informational

**Context:** BorrowerOperations.sol#L185

**Description:** `vars.THUSDFee` does not have any impact and could be removed from `BorrowerOperations.sol`.

**Recommendation:** Consider removing `vars.THUSDFee` from the codebase.

### 3.5.2 Misleading comment for `TIMEOUT`

**Severity:** Informational

**Context:** PriceFeed.sol#L37

**Description:** The `TIMEOUT` constant is used for both Chainlink and Tellor, but the attached comment does not reflect that since it only names Chainlink.

**Recommendation:** Consider changing the comment so that it is clear that the variable is used for Teller too.

### 3.5.3 Unused storage variables `borrowerOperationsAddress` and `troveManagerAddress`

**Severity:** Informational

**Context:** PriceFeed.sol#L30-L31

**Description:** The defined storage variables `borrowerOperationsAddress` and `troveManagerAddress` are not being used.

**Recommendation:** Consider removing such variables.

### 3.5.4 `decimals` variable name collision in `BAMM.sol`

**Severity:** Informational

**Context:** BAMM.sol#L110

**Description:** `BAMM` gets `decimals` from the `priceAggregator` and stores its value in a local variable `decimals`. This name is already being used by a storage variable, and thus results in name collision.

**Recommendation:** Consider changing the variable name to `_chainlinkDecimals`:

```
try priceAggregator.decimals() returns (uint8 _chainlinkDecimals) {
    chainlinkDecimals =_chainlinkDecimals;
```

### 3.5.5 `_collateralERC20` should be renamed to `_collateral` since it could represent ether

**Severity:** Informational

**Context:** BAMM.sol#L49

**Description:** The `_collateralERC20` variable in the contract's constructor can both represent an ERC20 address or `address(0)` for Ether. The name can lead to confusion since it doesn't always represent a ERC20 token.

**Recommendation:** Consider changing the variable name in such a way that there is no relation with ERC20s or Ether, for example `_collateral`.

### 3.5.6 Switch tellor's integration using `UsingTellor.sol` such as the official documentation

**Severity:** Informational

**Context:** TellorCaller.sol#L6

**Description:** The implemented integration of Tellor's oracle is not the same than the one in the official documentation.

**Recommendation:** Consider switch the current integration for the one recommended in the official guide, i.e. `UsingTellor.sol`.

### 3.5.7 `_sliceUint()` could be removed and replaced with `abi.decode(_value, (uint256)`

**Severity:** Informational

**Context:** TellorCaller.sol#L66-L75

**Description:** The function `_sliceUint()` extracts a uint256 from a bytes object. That same functionality could be replicated using `abi.decode(_value, (uint256))` reducing the contract size and saving some gas in deployment time following the example in Tellor's documentation.

**Recommendation:** Consider removing `_sliceUint()` and using `abi.decode(_value, (uint256))` instead.

### 3.5.8 Unnecessary `CropJoinAdapter` constructor call in `BAMM` constructor

**Severity:** Informational

**Context:** BAMM.sol#L54

**Description:** `BAMM` includes a redundant constructor call to `CropJoinAdapter`, even though the latter does not have a constructor defined.

**Recommendation:** Consider removing the `CropJoinAdapter` constructor call:

```
  constructor(
      address _priceAggregator,
      address payable _SP,
      address _thusdToken,
      address _collateralERC20,
      uint256 _maxDiscount,
      address payable _feePool,
      address _bProtocolOwner
  )
-     CropJoinAdapter()
  {
  // ...
```

### 3.5.9 Deprecate `2**256 - 1` usage in favor of `type(uint256).max`

**Severity:** Informational

**Context:** LiquityMath.sol#L95, LiquityMath.sol#L107

**Description:** `LiquityMath.sol` makes use of the expression `2**256 - 1` to represent the maximum value for an unsigned integer of type `uint256`. By adopting `type(uint256).max` instead, the codebase becomes more expressive and easier to understand, minimizing the risk of interpretation errors.

**Recommendation:** Consider replacing all instances of `2**256 - 1` by `type(uint256).max`.

### 3.5.10 Redundant/unnecessary named/explicit returns in functions

**Severity:** Informational

**Context:** LiquityBase.sol#L59-L64, LiquityBase.sol#L66-L71, LiquityBase.sol#L73-L79, PriceFeed.sol#L499-L502, PriceFeed.sol#L531-L534, PriceFeed.sol#L562-L565

**Description:** Although named returns allow functions to explicitly declare return variables, enhancing readability and self-documentation, several functions within `LiquityBase.sol` employ them without a genuine need, adding thus unnecessary complexity and verbodity, leading to potential confusion. The opposite occurs in several functions within `PriceFeed.sol`, where named returns are preferred and explicit while explicit returns are redundant.

**Recommendation:** Consider applying the following changes to the affected sites:

- **LiquityBase.sol#L59-L64**: remove the named return entirely.
- **LiquityBase.sol#L66-L71**: remove the named return entirely.
- **PriceFeed.sol#L499-L502**: remove the explicit returns entirely
- **PriceFeed.sol#L531-534**: remove the explicit returns entirely.
- **PriceFeed.sol#L562-565**: remove the explicit returns entirely.
- **LiquityBase.sol#L73-L79**: choose one of the following implementations

1. Remove explicit return

```
function _getTCR(uint256 _price) internal view returns (uint256 TCR) {
    uint256 entireSystemColl = getEntireSystemColl();
    uint256 entireSystemDebt = getEntireSystemDebt();

    TCR = LiquityMath._computeCR(entireSystemColl, entireSystemDebt, _price);
}
```

2. Remove named return

```
function _getTCR(uint256 _price) internal view returns (uint256) {
    uint256 entireSystemColl = getEntireSystemColl();
    uint256 entireSystemDebt = getEntireSystemDebt();

    return LiquityMath._computeCR(entireSystemColl, entireSystemDebt, _price);
}
```

### 3.5.11 `require` vs. `assert` in debuggability

**Severity:** Informational

**Context:** StabilityPool.sol#L391, StabilityPool.sol#L416, StabilityPool.sol#L456, THUSDToken.sol#L345-L346, THUSDToken.sol#L355, THUSDToken.sol#L363, THUSDToken.sol#L372-L373, TroveManager.sol#L368, TroveManager.sol#L915, TroveManager.sol#L1154, TroveManager.sol#L1202, TroveManager.sol#L1267, TroveManager.sol#L1273, TroveManager.sol#L1327, TroveManager.sol#L1402, BorrowerWrappersScript.sol#L66

**Description:** The codebase employs `assert` statements to perform certain checks. Because `assert` statements do not emit an error code or message, this makes debugging issues more difficult (relative to using `require` statements). Also, some or all appear to be unnecessary in a strict sense, and could potentially be removed for deployment with strong enough invariant testing; this would reduce the gas costs of deployment and execution.

**Recommendation:** Consider replacing `assert` statements with `require` statements, since the latter provides better error messages making it quicker to root-cause issues. Further, consider evaluating which `assert` statements are strictly necessary and which ones could be removed for deployment if there is sufficient invariant testing in place (note that it may be beneficial to add them back in if further work is done on the codebase that could cause one of them to be violated).

### 3.5.12 `ecrecover` signature malleability risk

**Severity:** Informational

**Context:** THUSDToken.sol#L302

**Description:** Signature malleability refers to the ability to generate an equally valid signature on some data given one already valid signature on said data, and it is a typical security risk when using the `ecrecover` function. Although, `permit` implementations don't rely on signature uniqueness as `nonce`s ensure they are single-use, it is important to be aware of this risk.

**Recommendation:** It is good practice to check that the value `s` belongs either to he top half of the elliptic curve or the bottom half.

### 3.5.13 Deprecate `SECONDS_IN_A_MINUTE` constant in favor of `1 minute`

**Severity:** Informational

**Context:** TroveManager.sol#L39

**Description:** `TroveManager.sol` defines the `SECONDS_IN_A_MINUTE` constant to represent the number of seconds in a minute. However, it would be more favorable to directly use the `minute` keyword, as it automatically converts the value into seconds.

**Recommendation:** Consider replacing all instances of `SECONDS_IN_A_MINUTE` by `1 minute` in `TroveManager.sol` and its child contracts.

### 3.5.14 Employ scientific notation in constants

**Severity:** Informational

**Context:** PCV.sol#L19, LiquityBase.sol#L18-L30

**Description:** PCV.sol#L19 defines the `BOOTSTRAP_LOAN` constant using numerical exponentiation, and LiquityBase.sol defines several large numerical constants with an excessive amount of zeros. In both cases, it is better to resort to scientific notation for such constants, as it offers better readability and provides code clarity.

In addition, doing so makes it consistent with other numerical constant declarations within the codebase (for example at BaseMath.sol#L6).

**Recommendation:** Consider adopting scientific notation throughout the codebase when defining numerical constants.

- PCV.sol#L19

```
- uint256 constant public BOOTSTRAP_LOAN = 10**26
+ uint256 constant public BOOTSTRAP_LOAN = 1e26
```

- LiquityBase.sol#L18-L30

```
- uint256 constant public _100pct = 1000000000000000000; // 1e18 == 100%
+ uint256 constant public _100pct = 1e18; // 1e18 == 100%

 // Minimum collateral ratio for individual troves
- uint256 constant public MCR = 1100000000000000000; // 110%
+ uint256 constant public MCR = 1.1e19; // 110%

 // Critical system collateral ratio. If the system's total collateral ratio (TCR) falls below the CCR,
↪  Recovery Mode is triggered.
- uint256 constant public CCR = 1500000000000000000; // 150%
+ uint256 constant public CCR = 1.5e18; // 150%

 // Amount of THUSD to be locked in gas pool on opening troves
- uint256 constant public THUSD_GAS_COMPENSATION = 200e18;
+ uint256 constant public THUSD_GAS_COMPENSATION = 2e20;

 // Minimum amount of net THUSD debt a trove must have
- uint256 constant public MIN_NET_DEBT = 1800e18;
+ uint256 constant public MIN_NET_DEBT = 1.8e21;
```

### 3.5.15 Deprecate `_chainID` function in favor of `block.chainid`

**Severity:** Informational

**Context:** THUSDToken.sol#L89, THUSDToken.sol#L277, THUSDToken.sol#L313-L317, THUSDToken.sol#L320

**Description:** `THUSDToken.sol` defines an internal function, `_chainID()`, that retrieves the chain ID through an `assembly` call to the `CHAIND` EVM opcode. However, since Solidity compiler version 0.8.0, it is possible to retrieve the chain ID through `block.chainid`.

**Recommendation:** Consider updating `THUSDToken.sol` to utilize `block.chainid` for retrieving the chain ID instead and removing the `_chainID` function, improving code readability and simplifying the implementation.

### 3.5.16 Inclusion of Debug Dependencies in Codebase

**Severity:** Informational

**Context:** BorrowerOperations.sol#L14, CollSurplusPool.sol#L11, DefaultPool.sol#L10, StabilityPool.sol#L15, THUSDToken.sol#L8, SortedTroves.sol#L10, TroveManager.sol#L14, PriceFeed.sol#L12, LiquityMath.sol#L5, BorrowerWrappersScript.sol#L16

**Description:** The codebase currently incorporates debug dependencies, specifically `console.sol`, which was most likely imported solely for testing purposes as it allows simple logging. Including this in the production codebase is considered undesirable.

**Recommendation:** Consider removing all instances of `import "./Dependencies/console.sol";`.

### 3.5.17 Insufficient event information in `increaseTHUSDDebt` and `decreaseTHUSDDebt` functions

**Severity:** Informational

**Context:** ActivePool.sol#L125-L135, DefaultPool.sol#L94-L104, IDefaultPool.sol#L11, IActivePool.sol#L12

**Description:** `increaseTHUSDDebt` and `decreaseTHUSDDebt` functions emit an event whenever THUSD Debt is updated. However, the emitted event does not provide any information regarding whether the trigger was an increase or a decrease (i.e. the events emitted by `increaseTHUSDDebt` and `decreaseTHUSDDebt` are indistinguishable). This lack of transparency makes it difficult to track and understand the changes in THUSD Debt, potentially leading to confusion and hindered troubleshooting efforts.

**Recommendation:** Consider either defining a specific event for each action:

```
event DefaultPoolTHUSDDebtIncreased(uint256 _THUSDDebt);
event DefaultPoolTHUSDDebtDecreased(uint256 _THUSDDebt);
```

or modifying the current event so that it also informs about the function where the event was emitted from:

```
event DefaultPoolTHUSDDebtUpdated(bytes4 _functionSelector, uint256 _THUSDDebt);
```

This way, it is possible to distinguish a THUSD Debt update caused by an increase from one caused by a decrease. The homologous mitigations should be applied to `ActivePool.sol` and its `ActivePoolTHUSDDebtUpdated` event.

### 3.5.18 Reminder to use USD feeds and not USDC to avoid Depeg Risk

**Severity:** Informational

**Context:** BAMM.sol#L222

**Description:** While the comment found at BAMM.sol#L222 is most likely a typo, it is worth re-iterating that feeds denominated in USDC can be prone to fluctuations that wouldn't happen to USD-denominated feeds.

From backtesting the USDC depeg, the worst difference was 13%, which can create strong incentives for Value Extraction as well as potentially create an incentive to lever up to the max, and swap the token while letting the position become under-collateralized.

**Recommendation:** Make sure all feeds are denominated in USD and not USDC.

### 3.5.19 Storage name collision

**Severity:** Informational

**Context:** BAMM.sol#L110

**Description:** The local variable `decimals` is shadowing the storage variable `decimals`. It's recommended to change the variable name of either one to avoid collisions.

**Recommendation:** Consider changing the variable name to `_chainlinkDecimals` for example

```
try priceAggregator.decimals() returns (uint8 _chainlinkDecimals) {
    chainlinkDecimals =_chainlinkDecimals;
```

### 3.5.20 Inconsistent usage of `delete` and manually zeroing variables

**Severity:** Informational

**Context:** StabilityPool.sol#L613, THUSDToken.sol#L192, PCV.sol#L192

**Description:** While functionally equivalent, inconsistent usage of the `delete` keyword and manually zeroing a variable can lead to confusion and decreased code readability.

**Recommendation:** Consider using a single method throughout the codebase for the sake of consistency.

### 3.5.21 Lack of consistency in parameter ordering among internal and exposed functions

**Severity:** Informational

**Context:** StabilityPool.sol#L355, StabilityPool.sol#L370-374

**Description:** The external function `offset` calls the internal functions `_computeRewardsPerUnitStaked` and passes its inputs in an appropriate order. Declaring the input parameters of the exposed function in an order that is different from the internal function calls is an inconsistent design choice.

**Recommendation:** Consider conserving the order of parameters among internal and exposed functions for the sake of consistency.

### 3.5.22 Unnecessary `uint` casting

**Severity:** Informational

**Context:** StabilityPool.sol#L421

**Description:** Explicit type casting of `DECIMAL_PRECISION` (a `uint256` variable, as inherited from BaseMath.sol) to `uint` (which is essentially an alias for `uint256`) is redundant and serves no practical purpose, adding unnecessary complexity and potentially confusing developers who might mistakenly assume there is a difference between `uint` and `uint256`.

**Recommendation:** Consider removing `uint` cast to improve code readability.

### 3.5.23 Incorrect or Misleading Information in Error String

**Severity:** Informational

**Context:** ActivePool.sol#L143, ActivePool.sol#L164, ActivePool.sol#L174, CollSurplusPool.sol#L127, CollSurplusPool.sol#L135, DefaultPool.sol#L119, DefaultPool.sol#L128

**Description:** Error strings in `require` statements play a crucial role in providing feedback to users and developers as they aid in debugging and help addressing errors effectively. When such error strings contain mistakes or misleading information, both developers and users may not discern the origin of the error effectively.

**Recommendation:** Consider rewriting the affected error strings.

- ActivePool.sol#L143: consider replacing the current error string for

```
"ActivePool: Caller is neither BorrowerOperations nor Default Pool"
```

as the current error string assumes developers and users know what `BO` stands for.

- ActivePool.sol#L164: consider replacing the current error string for:

```
"ActivePool: collateral must be ERC20"`
```

- ActivePool.sol#L174: consider replacing the current error string for:

```
"ActivePool: collateral must be ERC20"`
```

- CollSurplusPool.sol#L127: consider replacing the current error string for:

```
"CollSurplusPool: collateral must be ERC20 token"
```

- CollSurplusPool.sol#L135: consider replacing the current error string for:
- DefaultPool.sol#L119: consider replacing the current error string for:

```
"DefaultPool: collateral must be ERC20 token"
```

- DefaultPool.sol#L128: consider replacing the current error string for:

```
"DefaultPool: collateral must be ETH"
```

**Threshold:** Error strings have been modified in commit 00b730.

**Cantina:** Verified that the recommendation has been applied in commit 00b730.

### 3.5.24 Refactoring chained `if` statements improves readability

**Severity:** Informational

**Context:** ActivePool.sol#L110-L122, PriceFeed.sol#L345-L356

**Description:** While useful, chained `if` statements give rise to redundant evaluations, inefficient code flow and contribute towards a decrease in code readability. It is recommended to refactor the affected logic in such a way that chained `if` statements are avoided, or at least reduced.

**Recommendation:** The affected lines could be re-written by taking advantage of the following:

- Solidity performs short-circuiting when dealing with logical expressions
- The function signature invoked on all the affected interfaces is exactly the same

Consider applying the following modifications:

- ActivePool.sol#L110-L122:

First consider defining the following interface:

```
// Common interface for Pools
interface ICollateralBalanceUpdater {
    function updateCollateralBalance(uint256 _amount) external;
}
```

Then, substitute the chained `if` statements for the following single `if` statement

```
if (
    (collateralAddress != address(0))
     && (
        (_account == defaultPoolAddress) ||
        (_account == collSurplusPoolAddress) ||
        (_account == stabilityPoolAddress)
    )
) {
    ICollateralBalanceUpdater(_account).updateCollateralBalance(_amount);
}
```

- PriceFeed.sol#L346-L353:

Consider defining the `_badChainlinkResponse()` function as follows:

```
function _badChainlinkResponse(ChainlinkResponse memory _response) internal view returns (bool) {
    return(
        // Check for response call reverted
        (!_response.success) ||
        // Check for an invalid roundId that is 0
        (_response.roundId == 0) ||
        // Check for an invalid timeStamp that is 0, or in the future
        (_response.timestamp == 0 || _response.timestamp > block.timestamp) ||
        // Check for non-positive price
        (_response.answer <= 0)
    );
}
```

### 3.5.25 Could check for the amount of debt to burn being <= to remaining debt

**Severity:** Informational

**Context:** PCV.sol#L156-L158

**Description:** Most functions have checks with helpful error messages, it may be best to add a check to ensure that the `_thusdToBurn` is less than `debtToPay` as otherwise the function would revert with a `Arithmetic Overflow` error message:

```
function payDebt(uint256 _thusdToBurn) external override onlyOwnerOrCouncilOrTreasury {
    require(debtToPay > 0, "PCV: debt has already paid");
    require(_thusdToBurn <= thusdToken.balanceOf(address(this)), "PCV: not enough tokens");
```

**Recommendation:** Consider adding a `require` check for `_thusdToBurn <= debtToPay`.

### 3.5.26 `feeTHUSDAmount` can be computed in a simpler way

**Severity:** Informational

**Context:** BAMM.sol#L252

**Description:** The expression

```
feeTHUSDAmount = addBps(thusdQty, int(fee)) - thusdQty;
```

can be simplified to:

```
thusdQty * fee / MAX_BPS
```

**Recommendation:** Consider if you need the helper function `addBps` and refactor to remove it.