PUBLIC

Code Assessment

of the Bold

Smart Contracts

14 May, 2025

Produced for

Liquity

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Liquity team,

Thank you for trusting us to help you with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Bold according to Scope to support you in forming an opinion on their security risks.

Liquity implements Liquity V2, a decentralized stablecoin system with user set interest rates, iterating on Liquity V1.

The most critical subjects covered in our audit are functional correctness, rounding issues, and correctness of external integrations. The security regarding functional correctness is high, after issues in prior versions were resolved: Zappers can lose user funds. Security regarding rounding issues has been improved after the amount of share inflation was restricted, see Rounding in debt shares calculation can mint unbacked tokens. Security regarding external integrations is high, as issues with Balancer and Leverage Zapper have been resolved: BalancerFlashLoan missing access control and Leverage zappers do not return swap excess.

The general subjects covered are documentation, trustworthiness and code complexity. The project has very extensive documentation. The trustworthiness is high, as the system is designed to be immutable with limited trust assumptions. The system's contracts are very complex, which carries increased risk compared to simpler code.

In summary, we find that the core contracts provide a good level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	1
- Code Corrected	1
High-Severity Findings	3
- Code Corrected	3
Medium-Severity Findings	1
- Code Partially Corrected	1
Low-Severity Findings	16
- Code Corrected	12
- Specification Changed	3
- Acknowledged	1

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Bold repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	30 Aug 2024	7d9c8e68104cd4493f8b7da7e28e6951a2f84304	Initial Version
2	08 Nov 2024	d72dcb1bb2c8cbefd5ea37350d921d1bb7736da1	Version with fixes
3	09 Dec 2024	26ff5b24a83978801fa561af27dd9fcb228ed9af	Version with fixes
4	13 Jan 2025	ce0128a31922b4f7f5d79efb57ca6d1cfe6616d0	First Launch
5	03 Mar 2025	f87639de70369555073e00664bc8f7ea1fb9f000	Bug fixes
6	06 May 2025	2e901c785ba5c6d91a1fd55a358ead638961442a	Version with fixes
7	14 May 2025	abe7cbfbd465fba3812282c51773455766a70e96	Final Version

For the solidity smart contracts, the compiler version `0.8.24` was chosen.

This review assumes that the contracts will be deployed only to Ethereum mainnet. Prior to deploying to another chain, an additional in-depth assessment of the differences between that particular chain and Ethereum mainnet, and its effects on the contracts, must be done.

The following contracts in the folder `contracts/src` are in the scope of the review:

```
ActivePool.sol
AddressesRegistry.sol
BoldToken.sol
BorrowerOperations.sol
CollateralRegistry.sol
CollSurplusPool.sol
DefaultPool.sol
SortedTroves.sol
GasPool.sol
StabilityPool.sol
TroveManager.sol
TroveNFT.sol
Dependencies:
    AddRemoveManagers.sol
    AggregatorV3Interface.sol
    Constants.sol
    IRETHToken.sol
    LiquityBase.sol
    LiquityMath.sol
    Ownable.sol
Interfaces:
    IAddRemoveManagers.sol
    IAddressesRegistry.sol
    IBoldRewardsReceiver.sol
    IBoldToken.sol
    ICollateralRegistry.sol
    ICollSurplusPool.sol
    IBorrowerOperations.sol
    ICommunityIssuance.sol
    ICompositePriceFeed.sol
    IDefaultPool.sol
    IHintHelpers.sol
    IInterestRouter.sol
    ILiquityBase.sol
    ILQTYStaking.sol
    ILQTYToken.sol
    IMultiTroveGetter.sol
    IPriceFeed.sol
    ISortedTroves.sol
    IStabilityPool.sol
    IStabilityPoolEvents.sol
    ITroveEvents.sol
    ITroveManager.sol
    ITroveNFT.sol
    IWETHPriceFeed.sol
    IWETH.sol
    IWSTETH.sol
    IWSTETHPriceFeed.sol
PriceFeeds:
    CompositePriceFeed.sol
    MainnetPriceFeedBase.sol
    RETHPriceFeed.sol
    WETHPriceFeed.sol
    WSTETHPriceFeed.sol
Types:
    BatchId.sol
    LatestBatchData.sol
    LatestTroveData.sol
    TroveId.sol
    TroveChange.sol
Zappers:
    LeverageWETHZapper.sol
    WETHZapper.sol
    LeverageLSTZapper.sol
    GasCompZapper.sol
Interfaces:
    ILeverageZapper.sol
    IFlashLoanReceiver.sol
    IFlashLoanProvider.sol
    IExchange.sol
Modules/Exchanges:
    CurveExchange.sol
    UniV3Exchange.sol
Modules/FlashLoans:
    BalancerFlashLoan.sol
```

In Version 2, the scope was modified as follows:

- The files and contracts `BaseZapper`, `LeftoversSweep` and `HybridCurveUniV3Exchange` have been added.

2.1.1 Excluded from scope

Any contracts that are not explicitly listed above are out of the scope of this review. Third-party libraries, like openzeppelin libraries, are out of the scope of this review.

The soundness of the financial model was not evaluated.

The repository is a monorepo: only the smart contracts listed above were the scope of the review.

Any known issues at the time of the report, (such as those mentioned in the docs, GitHub issues on the public Bold repo, or on the Security Advisory page of the Liquity V1 repo) are considered out of scope and have generally not been duplicated in this report.

2.2 System Overview

This system overview describes the initially received version (**Version 1**) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Liquity V2 is a decentralized borrowing protocol, which issues the BOLD stablecoin. The codebase is a fork of the Liquity V1 codebase, using the same core architecture.

This system overview focuses on the parts that are new in Liquity V2 compared to V1. A detailed description of the entire system can be found in the ReadMe of the Bold repo (archived version here).

2.2.1 Major changes from Liquity V1

1. **Multi-collateral system**: The system includes a CollateralRegistry and multiple collateral branches, each with its own parameters and TroveManager, where liquidations and gains are handled within the same branch.
2. **Collateral choices**: The system supports collateral branches for ERC-20 tokens (WETH, rETH, and wstETH), but not native ETH.
3. **User-set interest rates**: Borrowers can set and change their annual interest rates, with interest accruing continuously and compounded discretely, and aggregate debt periodically minted as BOLD.
4. **Yield from interest paid to SP and LPs**: Interest from Troves is split between the Stability Pool and a router for DEX LP incentives, with each branch's interest paid to its own SP.
5. **Redemption routing**: BOLD redemptions are managed by the CollateralRegistry, aiming to restore the BOLD peg and reduce unbackedness in the most unbacked branches.
6. **Redemption ordering**: Redemptions now prioritize Troves with the lowest annual interest rates, ignoring collateral ratios.
7. **Unredeemable Troves**: Troves with very small BOLD debt after redemptions are tagged as unredeemable to prevent griefing attacks, becoming redeemable again when debt exceeds `MIN_DEBT`.
8. **Troves represented by NFTs**: Troves are transferable as NFTs, allowing multiple Troves per Ethereum address.
9. **Individual delegation**: Trove owners can appoint managers to set interest rates and control debt and collateral adjustments.

10. **Batch delegation**: Trove owners can appoint batch managers to adjust interest rates for multiple Troves within a predefined range.
11. **Collateral branch shutdown**: In extreme conditions, a collateral branch can be shut down, freezing operations and enabling urgent redemptions to clear debt quickly.
12. **Removal of Recovery Mode**: Recovery Mode is removed, with liquidations only occurring below the minimum collateral ratio. Borrowing restrictions still apply below the critical threshold.
13. **Liquidation penalties**: Liquidated borrowers below the minimum collateral ratio may now reclaim some collateral depending on the branch and liquidation type.
14. **Gas compensation**: Liquidators receive gas compensation in a mix of collateral and WETH, with a cap on variable compensation.
15. **More flexibility for SP reward claiming**: SP depositors can claim or stash LST gains and either claim BOLD yield gains or add them to their deposit.

2.2.2 CollateralRegistry

Liquity V2 supports multiple collateral tokens. The CollateralRegistry contract stores a list of all the valid collateral branches. Each collateral token has its own set of independent contracts, including a TroveManager, StabilityPool, BorrowerOperations, ActivePool, DefaultPool, SortedTroves, GasPool, and CollSurplusPool.

Collaterals can only be defined in the constructor. Once the system is deployed, it is impossible to add any new collateral.

The CollateralRegistry is now the entry point into redemptions (rather than the TroveManager as in Liquity V1). The `redeemCollateral` function will redeem BOLD from all active collateral branches simultaneously. The amount redeemed from each branch is weighted by their "unbackedness". The unbackedness is defined as the difference between the total BOLD debt of the branch, and the BOLD in the branch’s StabilityPool (SP).

Example: Two active collateral branches, branch 1 has 1000 debt and 500 BOLD in the SP. Branch 2 has 2000 debt and 500 BOLD in the SP. If a redemption of 400 BOLD is requested, 100 will be redeemed from branch 1 and 300 will be redeemed from branch 2. This is because the unbackedness of branch 2 is three times as large as that of branch 1.

Redemptions in each branch still work the same as in Liquity V1, except that troves are not closed if they are fully redeemed. Instead, they just stay open with no debt. Also, redemptions start with the lowest interest rate trove, not the lowest collateral ratio trove.

2.2.3 Trove interest rates

In Liquity V2, trove owners pay an interest rate on their borrowed BOLD. The interest rate is set when opening the position and can later be adjusted by the owner or can be delegated. The interest rate chosen must be in the valid range (currently between 0.5 - 100% APR). Troves are now redeemed in order of interest rate, starting with the lowest one. This incentivizes users to choose higher interest rates, as they will generally want to avoid redemption. However, users are also incentivized to choose a rate that is not too high, as otherwise, they will pay more interest than necessary. This should result in the market converging to a fair interest rate.

The contract stores the total interest rate paid by the system as the weighted sum of trove debt multiplied by trove interest rate:

```
SUM(trove.debt * trove.interestRate)
```

This expression is then used to mint the interest on every call modifying the debt. It is minted in a fixed split rate of `SP_YIELD_SPLIT` (currently set to 75%) to the Stability Pool (SP), and the rest to the Interest Router. The Stability Pool liquidity providers receive interest based on their share of the provided liquidity remaining in the pool (similar to how collateral gains are calculated). The yield is not automatically added to the user's BOLD deposit, it requires user action to compound.

The system uses the weighted sum to calculate the "approximate" average interest in the branch as:

```
SUM(trove.debt (exl. interest + fees) * trove.interestRate) /
SUM(trove.debt (incl. interest + fees))
```

Whenever a user borrows BOLD, they pay an upfront fee equivalent to 1 week of average interest on the amount borrowed. Further, any time they adjust their interest rate before `INTEREST_RATE_ADJ_COOLDOWN` (currently 3 days) has passed, they will be charged a fee equivalent to `UPFRONT_INTEREST_PERIOD` (currently 1 week) of the average interest rate. This is to discourage users from changing their interest rate often to avoid redemptions.

2.2.4 Individual delegation

Trove owners can delegate certain rights to other addresses (for example to facilitate a hot/cold wallet setup). There are four delegations that can be set:

1. addManager: This role can execute trove actions that improve a trove's collateralization, such as paying back debt or adding collateral. If this role is set to the zero address, anyone is allowed to.
2. removeManager: This role can execute trove actions that make a trove's collateralization worse, such as taking on debt or removing collateral.
3. receiver: This is the address that will receive minted tokens or collateral requested by the remove manager. If it is set to zero, the owner will receive the tokens. When the owner makes a withdrawal, the tokens are always sent to the owner.
4. interestIndividualDelegate: This role can adjust the interest rate of the trove within a certain range.

2.2.5 Batch delegation

Trove owners who do not want to manage their interest rate themselves can join a batch. A batch is a group of troves that all have the same interest rate, set by the batch manager. The manager can adjust the batch's interest rate at a specified maximum frequency and within a predefined range. In return, the manager can set a management fee, which is minted to the manager as new BOLD tokens, while the corresponding debt is added to the troves in the batch (pro rata).

The debts of all troves in a batch are accounted together, in a single `debt` variable. Each trove in turn receives `debtShares`, which represent their share of the debt. A trove's debt (ignoring pending debt redistributions) is calculated as:

```
batch.debt * trove.batchDebtShares / batch.totalDebtShares;
```

2.2.6 Collateral Branch shutdown

A branch can shut down under two conditions:

1. The TCR (Total collateral ratio) of the branch falls below the SCR (shutdown collateral ratio)
2. The Chainlink oracle fails by reverting, returning a non-positive price, or becoming stale

An oracle failure will trigger a shutdown when a user attempts to close a trove (either by closing or liquidating) or when the oracle is called directly to fetch the price. Other actions, such as opening a trove, will simply revert and not trigger a shutdown.

The condition *TCR* < *SCR* will trigger a shutdown when the *shutdown* function is called. A shutdown will perform the following steps:

1. Mint any pending interest to the Stability Pool (SP). No more interest will be minted after this point.
2. Set the branch's *isShutdown* flag in the *BorrowerOperations*.
3. Set the shutdown time in the active pool and trove manager.

The *isShutdown* flag in the *BorrowerOperations* will prevent users from performing any operations with the contract except for closing troves. The shutdown time is used to calculate the pending interest owed by trove owners up until shutdown and to mint any pending management fee to the batch manager. After a shutdown, the collateral registry will not route any redemptions through the branch. Instead, redeemers are expected to call *urgentRedemption* on the TroveManager to redeem collateral from the shut-down branch. These urgent redemptions are allowed for all troves, regardless of their collateral ratio and interest rate. They generally behave like regular redemptions, except that there is no redemption fee, and the redeemer receives a bonus of `URGENT_REDEMPTION_BONUS` (e.g. 1%) on the collateral they redeem, paid by the trove owner. Urgent redemptions can lower the collateralization ratio of the system, prioritizing speed. The shutdown allows liquidations to continue, so the following actions are still allowed:

- `closeTrove()`
- `batchLiquidateTroves()`
- `claimCollateral()`
- `urgentRedemption()`

In case an urgently redeemed trove has an ICR under 101%, the redeemer will receive all the collateral, and the trove will be left with some debt. This debt can then be liquidated against the stability pool or redistributed to other troves. In case the TCR of the branch is below 101% and the stability pool is empty, the end-state will be a single remaining trove with no collateral and some bad debt. This could lead the BOLD token to depeg, which could in turn affect the other branches.

2.2.7 Gas compensation

For every trove opened, a gas compensation for a potential liquidation must be deposited. The fixed gas compensation (currently set to 0.0375 WETH) is always paid upfront, in WETH, regardless of the collateral token used.

The second part of the gas compensation is determined at the time of liquidation. It is taken as a percentage of the trove's collateral (currently set to 0.5%), with a maximum cap (currently set to 2 (2e18) of the collateral token).

This ensures that the gas compensation for large troves is higher than for small troves. This makes sense, as an unliquidated large trove is a bigger risk to the system's health than a small one.

2.2.8 PriceFeeds (Oracles)

Liquity V2 uses price feeds based on Chainlink oracles.

If a call to the Chainlink aggregator fails during a call to `fetchPrice()`, the `disablePriceFeedAndShutDown` function will be called, which initiates the Collateral Branch Shutdown. The branch will continue using the price feed's last valid price. This price may be significantly different from the real market price.

For some collateral tokens, a *CompositePriceFeed* is used. Here, the LST/USD price is calculated as the product of the LST/ETH and the ETH/USD price. The LST/ETH price is queried in two ways: once by calling the Chainlink oracle, and once using the LST's contract to get the "canonical rate" of how many ETH are backing each of the LST tokens. Out of these two, the minimum price is used. In **Version 1**, if the canonical rate call reverts, the oracle will not trigger a branch shutdown. (This changed in Changes In Version 2.)

2.2.9 Zappers

Liquity V2 introduces zappers, which can be used to wrap ETH into WETH (required for the gas compensation) and make a deposit to Liquity V2 in the same transaction. They can also be used to take on leverage using flashloans.

The zapper will set the addManager, removeManager, and receiver to itself in BorrowerOperations. This allows the zapper to adjust the trove on behalf of the user later. The zapper implements its own access control, which also supports delegating the same roles.

There are currently two supported zappers: GasCompZapper allows depositing ERC20 collateral and wrapping ETH for the gas compensation at the same time. WETHZapper allows wrapping ETH and using it for the gas compensation and as collateral simultaneously.

The zappers contain the following functions:

- `openTroveWithRawETH()`: Allows opening a Trove using raw ETH as the gas compensation (and raw ETH as collateral for WETHZapper).
- `addCollWithRawETH()`: Allows adding collateral to a trove. For WETHZapper, raw ETH is first wrapped. The user must be the trove owner or be set as add manager in the zapper.
- `withdrawCollToRawETH()`: Allows withdrawing collateral from a Trove. For WETHZapper, WETH is unwrapped to ETH before being sent to the user. The user must be the trove owner or be set as remove manager in the zapper.
- `adjustTroveWithRawETH()`: Allows adjusting a Trove's collateral and debt. For WETHZapper, raw ETH can be wrapped and used as collateral or unwrapped when withdrawing. The user must be the trove owner or have the required permissions for the adjustment in the zapper.
- `closeTroveWithRawETH()`: Allows closing a Trove. The gas compensation is unwrapped and sent to the user as raw ETH. For WETHZapper, withdrawn collateral is also unwrapped to ETH. The user must be the trove owner or be set as remove manager in the zapper.

The basic zappers are extended by LeverageLSTZapper and LeverageWETHZapper. These allow taking a flashloan to create or unwind a levered BOLD position. Currently, only Balancer is implemented as a flashloan source.

The leverage zappers contain the following functions:

- `openLeveragedTroveWithRawETH()`: Allows opening a Trove using raw ETH as the gas compensation (and raw ETH as collateral for WETHZapper). Some of the collateral is provided by the user, the rest is taken from a flashloan. After opening, the minted BOLD are sold on a DEX to repay the flashloan. This creates a levered long position on the collateral token.
- `leverUpTrove()`: Uses a flashloan to increase the leverage of a trove. The user must be the trove owner or be set as remove manager in the zapper and the zapper must be set as add manager, remove manager and receiver in the trove manager.
- `leverDownTrove()`: Uses a flashloan to decrease the leverage of a trove. The user must be the trove owner or be set as remove manager in the zapper and the zapper must be set as add manager, remove manager and receiver in the trove manager.

When swapping, the minimum swap price is defined implicitly. With the passed `params`, the user must define the amounts correctly. The swap will always happen such that the exact amount of collateral taken as flashloan is swapped (to). For example, if the user calls `leverUpTrove()` with a `flashLoanAmount` of 1 WETH and a `BOLDAmount` of 2000 BOLD, 2000 BOLD will be taken as additional debt, and at most this amount will be swapped for 1 WETH (any excess will not be swapped). If the price on the chosen market is worse than 2000 BOLD per WETH, the transaction will revert. This is the minimum swap price. If the user does not set the `params` correctly, they may incur significant slippage through MEV Sandwich attacks. Note that the removeManager set in the zapper has the power to perform these swap parameters, so they must be fully trusted.

2.2.10 Trust Model

The system's contracts are designed to be immutable, with limited trust assumptions. There are no admin roles (after deployment is complete) or upgradeability mechanisms (except for those in the oracles used).

The AddressesRegistry, BoldToken, and MainnetPriceFeedBase contracts assign an owner role. These owners are expected to correctly configure the system and relinquish their ownership afterwards.

The trust model for removeManager and receiver depends on their configuration. When both addresses are set, the receiver gets all funds withdrawn: removeManager and receiver are fully trusted to manage the user funds. If both addresses are compromised, they could withdraw all the funds from the Trove. If the receiver is set to address 0, the funds are transferred to the Trove owner instead. In this case, the removeManager is only trusted to maintain the collateralization ratio to avoid causing losses to the owner (e.g., by lowering the ICR close to the liquidation threshold). Additionally, the receiver configured in the Zapper must be trusted not to revert transactions. Otherwise, they could cause a denial of service (DoS) by exhausting all available gas when receiving ETH via raw call.

The interestIndividualDelegate and batchManager are trusted to adjust the interest rate in the best interest of the Trove owner. They are expected not to modify the interest rate more frequently than necessary, as the owner will incur upfront fees for each adjustment. Furthermore, they are trusted to set the interest rate at an optimal level, balancing the risk of redemptions with the cost of interest.

The Chainlink Oracle is trusted to provide price updates within the expected threshold and return prices in the expected format, not changing the return data size or the decimals used to report prices. Additionally, they are trusted not to revert any calls.

The addManager can only improve the collateralization ratio of a trove. The addManager is mostly untrusted, but can perform some special actions described in the Notes.

We have specifically investigated the use of WETH, wstETH and rETH as collateral tokens (i.e. their respective Ethereum mainnet deployments). Any token that does not use 18 decimals of precision or has other non-standard behavior is not supported by the protocol. https://web.archive.org/web/20240930034220/https://github.com/d-xo/weird-erc20

2.2.11 Changes in Version 2

Version 2 of the codebase introduces the following changes:

- The `closeTroveFromCollateral` function has been added to the GasCompZapper and WETHZapper contracts. This function allows the zapper to close a trove by selling its collateral, instead of repaying with BOLD from the user. It accomplishes this by taking a flashloan of collateral tokens, swapping them to BOLD on an exchange, then using the received BOLD to repay the trove debt. The collateral that was in the trove is used to repay the loan, with excess going to the user.
- Unredeemable troves have been renamed to zombie troves. When a trove becomes a zombie trove but still has some debt (through a partial redemption), a pointer to it is stored. When the next redemptions on the branch happen, that trove will be redeemed first (until it has no more debt). Usually, there should only be one zombie trove with more than zero debt at a time. However, if there are troves with collateral and no debt when a redistribution happens, they can receive debt. This debt will not be redeemable.
- Similar how batch manager can only update the interest rates in a maximum frequency, users can now set a minimum period between interest rate adjustments for their interestIndividualDelegate. This can lower the trust required in the delegate.
- Borrowing is now allowed if the system is below the critical threshold, as long as the new trove brings the system back above the critical threshold.

- The CompositePriceFeeds has been updated. They now continue to report prices if the LST Chainlink Oracle fails. The collateral branch is still shutdown, however the LST price will then be calculated using the ETH oracle price and the canonical exchange rate returned by the LST contract. The price is only allowed to be reduced, it can never increase when calculated this way. According to Liquity, this design incentivizes redeemers to quickly repay all outstanding debt on a shutdown branch by offering them the "lowest available price" on (urgent) redemptions.
- Redemption of LSTs uses the `fetchRedemptionPrice` function to get the collateral token price, taking the maximum of the LST price and the canonical price, as long as the prices are not at least 2% (for rETH) or 1% (for wstETH) apart. This reduces the risk of redemption arbitrage due to oracle price deviations. In previous versions, redeemers could exploit (downward) price deviations to redeem collateral at a lower price to cause losses to trove owner.
- In case the price difference exceeds the threshold, it is assumed to be a legitimate price difference, with the canonical exchange rate lagging behind (updated once per day). In these cases, redemptions (like all other price-relevant operations) return the minimum price for rETH, as the rETH/ETH oracle price is considered by Liquity to be more susceptible to upward manipulation that would otherwise make redemptions unprofitable. The stETH/USD oracle price feed is considered to be more resistant, so the redemption price for wstETH in case of a > 1% price difference is calculated with Chainlink's stETH price multiplied by the canonical exchange rate.
- The LST price feeds now trigger a shutdown if the canonical price call to the LST contract reverts.

2.2.12 Changes in Version 4

Version 4 of the codebase introduces the following changes:

- The `INTEREST_RATE_ADJ_COOLDOWN` has been increased from 3 to 7 days.
- The `URGENT_REDEMPTION_BONUS` changed from 1% to 2%.
- The `MAX_LIQUIDATION_PENALTY_REDISTRIBUTIONS` increased from 10% to 20%.

2.2.13 Changes in Version 5

Version 5 of the codebase introduces the following updates:

- Parameter Adjustments:
  - The Critical Collateral Ratio (CCR) is increased from 150% to 160% for staked ETH branches (sETH / rETH).
  - The Minimum Collateral Ratio (MCR) is increased from 110% to 120% for staked ETH branches (sETH / rETH).
  - The Liquidation Penalty Redistribution is increased from 10% to 20% for staked ETH branches (sETH / rETH).
  - The maximum annual interest rate is increased from 100% to 250%.
  - The minimum interest rate adjustment period is increased from 1 minute to 1 hour.
- A 10% buffer collateral ratio is introduced on top of the MCR for modifying troves in batches. For staked ETH branches, this results in an effective collateralization ratio of 130% (120% MCR + 10% buffer). The increased collateralization requirement applies only when opening a new trove, modifying an existing one, or joining/switching to a batch. Liquidations remain unaffected and continue to use the MCR of 120% for staked ETH branches.

- A minimum deposit of 1e18 BOLD tokens is now required in the Stability Pool. Only amounts exceeding this minimum can be used to offset debt during liquidations. Users cannot withdraw the last 1e18 BOLD from the Stability Pool, but they can initially deposit less than 1e18 BOLD.
- The Stability Pool can no longer be fully emptied via liquidation due to the minimum deposit requirement. As such the the pool no longer stores epochs to keep account of pool-emptying events. The precision of the running variable *P*, which tracks the remaining deposits in the pool, is increased from 1e18 to 1e36. Further, the error correction mechanisms for calculating running variables *P*, *S*, and *B* are removed as part of this change.

2.2.14 Changes in Version 6

Version 6 of the codebase introduced the following changes:

- Liquidators now receive only a portion of the trove collateral offset by the Stability Pool instead of the total collateral. If the total debt of the trove is redistributed, they only receive the WETH gas compensation.
- The owner of the last trove in a branch can now close the trove if the branch is shutdown.
- Allow anyone to remove a trove from a batch with inflated batch shares (`kick`). The trove from the batch is assigned the same interest rate as the batch it was previously in.
- The `RemoveManager` role now inherits `AddManager` permissions, allowing any `RemoveManager` to add collateral or repay debt for a trove.

2.2.15 Changes in Version 7

Version 7 of the codebase introduced the following changes:

- The MainnetPriceFeedBase contract no longer has an owner and is now configured during deployment.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Open Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation
- Trust: Violations to the least privilege principle

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	1

- Unredeemable Troves Can Pay Minimum Interest Rate Code Partially Corrected

Low-Severity Findings	1

- Discrepancy in swapFromBold Behavior Acknowledged

5.1 Unredeemable Troves Can Pay Minimum Interest Rate

Design Medium Version 1 Code Partially Corrected

CS-BOLD-005

A trove that has less than `MIN_DEBT` is marked as unredeemable (also referred to as a zombie trove). Unredeemable troves are not part of the sorted list of troves that can be redeemed, to prevent clogging of the list with tiny troves.

As unredeemable troves have no risk of being redeemed, they are not incentivized to pay more than the minimum interest rate. A trove can intentionally be made unredeemable by opening it with `MIN_DEBT`, then self-redeeming a small amount of debt to bring it below `MIN_DEBT`.

As such, it could be profitable to open many small troves and make them unredeemable, rather than creating a single large trove that must pay a higher interest rate to avoid redemption.

However, there are a few mitigating factors that make this strategy unattaractive:

1. For each trove opened, the `ETH_GAS_COMPENSATION` amount of WETH must be locked up. This is a fixed cost per trove. However, this amount is returned when closing the trove, so the cost is mainly the cost of capital/opportunity cost on this amount.
2. The Ethereum gas fee must be paid for the opening, redeeming, and closing of each trove. This is a fixed cost per trove, but changes over time depending on the gas price.
3. Unredeemable troves cannot be adjusted, except if they are made redeemable. This means any collateralization adjustment (e.g. to avoid liquidation) to the troves must be done in 2 steps: First adjust the trove, then self-redeem again to make it unredeemable. These steps must be repeated for every trove, so the cost of adjusting the trove is 2n times the cost of adjusting a single trove, where n is the number of troves.

4. Interest accrual will eventually increase the debt of the trove above `MIN_DEBT`, making it redeemable. This means the trove must start with a debt significantly below `MIN_DEBT`.

Ultimately, the strategy's profitability will depend on the balance between the additional costs (which heavily depend on gas price) and the interest rate savings.

Code partially corrected:

Unredeemable troves have been renamed to zombie troves. When a trove becomes a zombie trove but still has some debt (through a partial redemption), a pointer to it is stored. When the next redemptions on the branch happen, that trove will be redeemed first (until it has no more debt).

This resolves the attack described in the issue, as small troves can now be redeemed.

Note that there is still an edge case where the issue persists: Usually, there should only be one zombie trove with more than zero debt at a time. However, if there are troves with collateral and no debt when a liquidation through redistribution happens, they can receive debt. This debt will not be redeemable and those troves can still pay only the minimum interest rate, as long their debt stays below `MIN_DEBT`.

Intentionally creating troves like this is difficult, as it requires liquidations to take place when the stability pool is completely empty, which should only happen in extreme circumstances. It should be significantly easier to do this when the system is first deployed and there are no other users on the same branch yet, or when a branch's collateral token becomes very unpopular.

Note on the audit process: This issue was reported to ChainSecurity by Liquity while the audit was in progress. It had also been discovered internally already, though the severity had not been fully assessed yet.

The GitHub issue related to Liquity's report can be found here.

5.2 Discrepancy in swapFromBold Behavior

Design Low Version 1 Acknowledged

CS-BOLD-006

The UniV3Exchange and CurveExchange both implement the `swapFromBold` function. As the exchanges can be used interchangeably, it is expected that they behave the same.

However, the UniV3Exchange uses an `ExactOutput` swap, which swaps the exact amount of collateral tokens requested, and may leave excess BOLD tokens. CurveExchange on the other hand uses Curve's `exchange` function, which is equivalent to an `ExactInput` swap, and may leave excess collateral tokens.

Acknowledged:

Liquity acknowledged that `ExactOutput` is more natural but stated that they may add more exchanges in the future and cannot guarantee that every exchange implements the version they choose.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Open Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	1

- Rounding in Debt Shares Calculation Can Mint Unbacked Tokens Code Corrected

High-Severity Findings	3

- BalancerFlashLoan Missing Access Control Code Corrected
- Leverage Zappers Do Not Return Swap Excess Code Corrected
- Zappers Can Lose User Funds Code Corrected

Medium-Severity Findings	0

Low-Severity Findings	15

- Incorrect Scaling of P Code Corrected
- Pool Deposit Are Compared to Precision Code Corrected
- Total Deposit Are Compared to Wrong Constant Code Corrected
- Batches Can Be Used to Make Two Free Adjustments in a Row Code Corrected
- Delegation Specification Code Corrected
- Incorrect Code Comments Specification Changed
- Interest Delegates Are More Trusted Than Needed Code Corrected
- Missing Payable Modifier Code Corrected
- Missing Validation of Troves in Urgent Redemptions Code Corrected
- Opening Troves Can Be Blocked Code Corrected
- Out-of-gas May Lead to Shutdown Code Corrected
- Price Limit in UniV3Exchange Is Too Strict Specification Changed
- Shutdown Can Be Triggered Twice Code Corrected
- User-provided transferFrom Source Address Code Corrected
- Zapper Delegation Is Not Reset When a Trove Is Closed Specification Changed

Informational Findings	12

- Gas Optimizations in StabilityPool Code Corrected
- Receiver Address in Balancer Flashloan Is Reset Late Code Corrected
- Backed Tokens Can Be Redeemed Unproportionally Code Corrected
- CEI Pattern Violated in Adjust Trove Code Corrected
- Comments From Development Code Corrected
- Core Debt Invariant Incorrectly Documented Code Corrected
- Floating Pragma Code Corrected

- Inconsistent Input Validation by Zappers Code Corrected
- Indexed Parameters of Events Code Corrected
- Minting Unbacked Tokens via Redistributions Code Corrected
- Misleading Function Names in Zapper Code Corrected
- Missing Events Code Corrected

6.1 Rounding in Debt Shares Calculation Can Mint Unbacked Tokens

Design Critical Version 1 Code Corrected

CS-BOLD-001

The function `TroveManager._updateBatchShares()` rounds down the debt shares of the trove:

$batchDebtSharesDelta = currentBatchDebtShares * debtIncrease / batchDebt$

The rounding error increases with the amount of debt per debt share of the batch. Initially, one debt share is minted per unit of debt, but this ratio decreases over time as the batch debt grows due to rounding, interest, and management fees.

An adversary can exploit this by making the ratio very small, i.e.,

$batchDebt > 2000e18 * currentBatchDebtShares$

At a ratio of 2000e18, an attacker can mint Bold tokens for free. The amount of debt shares they receive for creating a trove of minimum debt is rounded to zero, allowing them to offload the debt to other users in the batch.

There are two ways to increase the ratio of debt to shares:

1. **High Management Fees and Frequent Compounding**: Charging a high management fee of 100% and compounding frequently can increase the debt-to-shares ratio to 2000e18 after approximately 50 years (ln(2000e18) / 100%).
2. **Donating Dust Amounts of Debt**: Donating small amounts of debt to the batch and increasing the impact of off-by-one rounding errors by creating a tiny trove.

The steps to exploit the second method are as follows:

1. Open Trove A with 2000e18 debt in the batch and 2000e18 shares.
2. Redeem Trove A down to 1 debt and 1 share.
3. Open another Trove B with 2000e18 debt in the same batch.
4. Wait to earn some interest, e.g., 1 wei.
    - Total shares = 2000e18 + 1, Total debt = 2000e18 + 2 (assuming zero fee for simplicity)
5. Donate 1 wei by calling *adjustTrove* with *increaseDebt = 1*. No shares will be minted since (2000e18 + 1) * 1 // (2000e18 + 2) = 0.
    - Total shares = 2000e18 + 1, Total debt = 2000e18 + 3
6. Close Trove B and pay back (2000e18 + 3) * 2000e18 / (2000e18 + 1) = 2000e18 + 1 debt.
    - Total shares = 1, Total debt = 2
7. Open another trove with 2000e18 debt in the same batch.

- Total shares = 2000e18 + 1, Total debt = 2000e18 + 2

An attacker can repeat steps 5. - 7. in a loop, increasing the amount of debt per share in each iteration. It can be shown that the ratio can be increased by a factor of at least 3/2 in each iteration. Thus, an attacker can inflate the ratio to 2000e18 in fewer than 120 iterations:

$$120 = log_{1.5}2000e18$$

Once the ratio is inflated, the attacker can mint Bold tokens for free by offloading the debt to Trove A. Multiple mints can be performed in one transaction, creating bad debt as Trove A cannot be liquidated in between. The bad mint will further inflate the batchDebtShares ratio, allowing the next mint to be twice the size.

In this way, the attacker can mint a large number of unbacked BOLD, at most until the TCR of the branch reaches the CCR.

There are also two other ways in which the manipulated batchDebtShares ratio can be exploited:

An attacker can exploit the redemption mechanism with an inflated exchange rate. During redemption, the redeemer specifies the amount of debt to be redeemed and then burns the corresponding number of shares:

$$batchDebtSharesDelta = currentBatchDebtShares * debtDecrease/_{b}atchDebt$$

An attacker can create a trove and join a redeemable batch. When paying back debt, the shares burned are rounded down. For example, if *currentDebtShares = 10* and *batchDebt = 20_000e18*, any redemption of less than 200e18 Bold will not burn any shares:

$$0 = 10 * 199e18//20.000e18$$

Thus, an attacker can repeatedly redeem a victim's trove in that batch, buying up all their collateral while paying back the batch's debt without burning any of the victim's debt shares. The attacker receives *attackerShares / totalShares * redeemBold* of Bold paid by the victim.

Finally, an attacker can also access dirty memory by creating a redeemable trove in a batch with no debt shares. This is done by creating a trove with the minimum debt of 2000e18 and no shares in a batch, exploiting the rounding issue discussed above. All other troves in the batch must be closed to make the number of total number of shares in the batch zero.

The *redeemCollateral* function overwrites the *singleRedemption* structure during each loop iteration. When the trove is in a batch, the function *_redeemCollateralFromTrove* computes the latest trove data from batch data. The function *_getLatestTroveDataFromBatch* does not write anything to *_latestTroveData* if *totalDebtShares* are zero. In this case, the attribute *singleRedemptionValues.batch* will contain values from the previous loop iteration.

```
if (totalDebtShares > 0) {
    _latestTroveData.recordedDebt = _latestBatchData.recordedDebt * batchDebtShares / totalDebtShares;
    ...
}
```

The attack will mint unbacked interest and management fee and lead to insolvency of the protocol.

Code corrected:

The function `_requireBelowMaxSharesRatio` has been added to the TroveManager to prevent the ratio of debt shares to debt to increase by too much. The function takes the boolean *_checkBatchSharesRatio* and reverts when the boolean is true and the ratio of debt to shares exceeds a threshold of `MAX_BATCH_SHARES_RATIO` (1E9).

```
function _requireBelowMaxSharesRatio(
    uint256 _currentBatchDebtShares,
    uint256 _batchDebt,
    bool _checkBatchSharesRatio
) internal pure {
    // debt / shares should be below MAX_BATCH_SHARES_RATIO
    if (_currentBatchDebtShares * MAX_BATCH_SHARES_RATIO < _batchDebt && _checkBatchSharesRatio) {
        revert BatchSharesRatioTooHigh();
    }
}
```

The boolean is set to true for all operations that call into `_updateBatchShares`, except redemptions. Hence, for troves in a batch with inflated shares, the following operations cannot be performed:

- opening troves
- adjusting troves (i.e. increasing debt, withdrawing collateral, etc...)
- applying pending debt

In addition to redemptions, the following operations that burn shares by calling into `_removeTroveSharesFromBatch` are still allowed:

- redeeming troves
- liquidating troves
- closing troves
- removing the trove from the batch

In case a user ends up being part of a batch with inflated shares for some reason, they should change to a different batch to unlock full functionality again.

The issue described 2 root causes: Inflating via high interest rate and inflation via donation attacks. Inflation with high interest rate is still possible, however inflation via donation attack is only possible until the ratio of debt shares to debt reaches `1e9`. In theory, an attacker can exploit the ratio up to `1e9` and then wait 25 years more to inflate the ratio to 100e18.

The issue described 3 attacks vectors:

1. Minting unbacked tokens by inflating the ratio of debt shares to debt
2. Redeeming a victim's trove without burning any of the victim's debt shares
3. Accessing dirty memory by creating a redeemable trove in a batch with no debt shares

The first and third attack vectors are prevented by restricting the opening of positions. The second attack vector is still theoretically possible but progresses very slowly. With 100% inflation compounded weekly, the debt shares to debt ratio would reach `100e18` after 25 years. The victim trove would need to stay in the batch for the entire time to be at-risk.

6.2 BalancerFlashLoan Missing Access Control

Correctness **High** **Version 1** **Code Corrected**

CS-BOLD-002

The leverage zapper contracts use Balancer flashloans to make trove adjustments. For example, the `receiveFlashLoanOnLeverUpTrove` function can only be called by the flashloanProvider. `flashLoanProvider.makeFlashLoan()` is in turn supposed to be called through the `leverUpTrove` function, which checks that the caller has rights to adjust the trove.

However, `flashLoanProvider.makeFlashLoan()` does not have any access control. It can be called by any address, and any `params` can be passed. In this way, the access control of the zapper can be circumvented. The flashLoanProvider will call back into the zapper and adjust any trove (that the Zapper has rights to remove from) passed in the `params`, even though the user does not have the required rights. Additionally, an attacker could call the Balancer vault directly, specifying the BalancerFlashLoan as callback recipient, without using the `makeFlashLoan` function at all. The `receiveFlashLoan` function would accept the callback, as it only checks that the call comes from the Balancer vault.

An attacker could use `leverUpTrove()` to bring a trove very close to liquidation, or `leverDownTrove()` to reduce the trove's leverage. The attacker could make a profit by sandwiching trades on the exchange used, which may cause slippage if the trove is large compared to the market's liquidity. The attacker can set the slippage limits that will be used in the trade (by choosing the parameters), so the execution price can become arbitrarily bad.

Code corrected:

In (Version 2), the BalancerFlashLoan contract no longer takes the zapper parameter as input. Instead, whenever the `makeFlashloan` function is called, the storage variable `receiver` will be set to the `msg.sender` of the call (expected to be a zapper). After the flashloan returns, the `receiver` is reset to the zero address.

The `receiveFlashLoan` function calls the `receiver`, which is now read from storage rather than the input params. This way, the contract can only call a zapper if the zapper previously called `makeFlashLoan()`. The zapper's access control is enforced before making the call to `makeFlashloan()`.

If `receiveFlashLoan` is called while the `receiver` is set to zero, the function will revert. This ensures that the `receiveFlash` function cannot be entered by calling the Balancer vault directly.

In (Version 4), the receiver address is now reset at the beginning of the function `receiveFlashLoan`.

6.3 Leverage Zappers Do Not Return Swap Excess

Design High Version 1 Code Corrected

CS-BOLD-003

The leverage zappers use swaps and flashloans to create leveraged positions. However, they do not handle excess tokens returned from the swaps. Excess tokens should be expected to be present often, as market prices can change between submitting a transaction and its execution.

In `UniV3Exchange.swapFromBold()`, the zapper swaps BOLD for collateral tokens using an `ExactOutput` swap. Any excess BOLD tokens will be left in the UniV3Exchange contract. The tokens will not be returned to the user.

In `CurveExchange.swapFromBold()`, `UniV3Exchange.swapToBold()`, and `CurveExchange.swapToBold()` the zapper swaps tokens using an `ExactInput` swap. Any excess output tokens will be sent to the zapper but will not be returned to the user.

Tokens stuck in the zapper can be claimed by anyone (see Zappers can lose user funds).

Code corrected:

In (Version 2), the exchange contract `UniV3Exchange` returns excess tokens to the Zapper contract. Further, the zapper functions calling Curve or Uniswap (i.e. `openLeveragedTroveWithRawETH`, `leverUpTrove`, and `leverDownTrove`) have been updated to send excess tokens back to the caller.

For this, initial token balances in the Zapper and the Caller address are stored before the swap with `_setInitialBalancesAndReceiver`. After the swap, `_returnLeftovers` checks the Zapper's balance and returns the difference of initial balance and current balance to the caller.

```
uint256 currentCollBalance = _collToken.balanceOf(address(this));
if (currentCollBalance > _initialBalances.collBalance) {
    _collToken.transfer(_initialBalances.receiver, currentCollBalance - _initialBalances.collBalance);
}
uint256 currentBoldBalance = _boldToken.balanceOf(address(this));
if (currentBoldBalance > _initialBalances.boldBalance) {
    _boldToken.transfer(_initialBalances.receiver, currentBoldBalance - _initialBalances.boldBalance);
}
```

6.4 Zappers Can Lose User Funds

Correctness High Version 1 Code Corrected

CS-BOLD-004

The adjust, leverdown, and repay functions in the zappers can reduce the debt of a user's trove. To do this, they transfer the specified amount to repay from the user and then trigger the repayment in the BorrowerOperations.

However, if the specified debt repayment amount would bring a trove below `MIN_DEBT`, only the amount that would bring the trove to `MIN_DEBT` is repaid. The rest of the funds will remain stuck in the zapper.

This is due to the following code in `_adjustTrove()`:

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

Consider the following example:

1. A user has a trove with a debt of 3000 BOLD.
2. The user calls the adjust function in the zapper with a debt repayment amount of 2000 BOLD.
3. The zapper transfers 2000 BOLD from the user to the zapper.
4. The zapper triggers the repayment in the BorrowerOperations. This will repay 1000 BOLD to the trove, bringing the debt to 2000 BOLD (the `MIN_DEBT`).
5. The remaining 1000 BOLD will remain stuck in the zapper.

The worst case is the following: When there is a debt repayment to one of the lowest interest rate troves in a branch, the repayment can be frontrun by a redemption that brings the trove just above `MIN_DEBT`. This will result in the entire intended repayment amount being stuck in the zapper. A malicious batch manager could intentionally reduce the interest rate of the trove's batch to make the redemption possible.

The worst-case attack would look as follows:

1. A user has a trove with a debt of 1'000'000 BOLD.
2. The user calls the adjust function in the zapper with a debt repayment amount of 500'000 BOLD.
3. The trove's batch manager frontruns the call and changes the interest rate to make the trove low in the redemption order (or it already had a low interest rate).

4. The attacker redeems the trove until it has only 2001 BOLD debt left.

#. The zapper transfers 500'000 BOLD from the user to the zapper. 4. The zapper triggers the repayment in the BorrowerOperations. This will repay 1 BOLD to the trove, bringing the debt to 2000 BOLD (the `MIN_DEBT`). 5. The remaining 499'999 BOLD will remain stuck in the zapper. 6. The attacker can collect the stuck funds by backrunning using the method below

Anyone can collect the stuck funds from the zapper by creating a trove in the zapper, setting the `receiver` role in BorrowerOperations to an address other than the zapper, then creating additional BOLD tokens through the zapper. The newly created BOLD will go to the `receiver`, and the zapper will send the stuck funds to the caller.

Code corrected:

The Zappers now inherit from the new LeftoversSweep contract. It provides functionality to track the contract's balances at the start of a call, then return any extra amount that is added during the call using the `_returnLeftovers` function.

In all instances where there could be leftovers, they are now returned to the caller.

6.5 Incorrect Scaling of P

Correctness Low Version 5 Code Corrected

CS-BOLD-042

In the function `StabilityPool.offset`, P can be scaled up. The code assumes that P is in the value range of (1e27, 1e36]. However, this is not always true. Instead, P can get scaled to a higher value.

For example for the following values:

- P = 2e17 + 1
- totalBoldDeposits = 1e20
- _debtToOffset = 5e19

The while loop in `offset()` is entered, since in the calculation of newP the new value is rounded down to 1e27. But after scaling up again the final computed P is 10000000000000000000000000000500000000 which is bigger than 1e36.

```
uint256 numerator = P * (totalBoldDeposits - _debtToOffset);
uint256 newP = numerator / totalBoldDeposits;

...

while (newP <= P_PRECISION / SCALE_FACTOR) {
    numerator *= SCALE_FACTOR;
    newP = numerator / totalBoldDeposits;
    currentScale += 1;
    emit ScaleUpdated(currentScale);
}
```

As a result, multiple computations in the code, e.g. in `getDepositorCollGain`, will be incorrect. This is because for a scale period that begins with a P > 1e36, the P can drop by more than 1e9, even though the code assumes it drops by 1e9.

Code corrected:

In Version 6 the function `StabilityPool.offset` has been modified to ensure that the value of P is always in the range [1e27, 1e36].

```
while (newP < P_PRECISION / SCALE_FACTOR) {
    numerator *= SCALE_FACTOR;
    newP = numerator / totalBoldDeposits;
    currentScale += 1;
    emit ScaleUpdated(currentScale);
}
```

6.6 Pool Deposit Are Compared to Precision

Correctness Low Version 5 Code Corrected

CS-BOLD-039

The function `_updateYieldRewardsSum()` compares the total deposits to `DECIMAL_PRECISION` when verifying that the minimum threshold for yield distribution is reached. However, other parts of the code use `MIN_BOLD_IN_SP` to check that the stability pool has enough funds. Note that both constants are set to `10**18` so the generated bytecode is the same.

Code corrected:

The function has been updated to compare the Stability Pool deposits against `MIN_BOLD_IN_SP` in Version 6.

6.7 Total Deposit Are Compared to Wrong Constant

Correctness Low Version 5 Code Corrected

CS-BOLD-040

The function `_updateYieldRewardsSum()` compares the total deposits to `DECIMAL_PRECISION` when verifying that the minimum threshold for yield distribution is reached. However, other parts of the code use `MIN_BOLD_IN_SP` to check that the total deposits are large enough. Both constants have the same value 1e18 so the bug has no impact.

Code corrected:

The function has been updated to compare the total deposits to `MIN_BOLD_IN_SP` in Version 6.

6.8 Batches Can Be Used to Make Two Free Adjustments in a Row

Design Low Version 1 Code Corrected

CS-BOLD-021

Whenever a trove's interest rate is adjusted, it is checked when the trove's last adjustment took place. If it is less than `INTEREST_RATE_ADJ_COOLDOWN` ago, the upfrontfee for an early adjustment is charged.

However, the `setBatchManagerAnnualInterestRate` function only takes into account the last adjustment of the batch, not of the individual troves.

This means a user can do the following repeatedly:

1. Wait until their trove's last adjustment was more than `INTEREST_RATE_ADJ_COOLDOWN` ago.
2. Join a batch controlled by them with a different interest rate, that was adjusted more than `INTEREST_RATE_ADJ_COOLDOWN` ago.
3. Adjust the interest rate of the new batch.
4. Wait until the batch adjustment was more than `INTEREST_RATE_ADJ_COOLDOWN` ago.
5. Leave the batch.

In this way, the user is able to adjust their interest rate twice in a row (steps 2./3.) without paying the upfront fee. Leaving the batch in step 5 does not trigger any upfront fee either as long as the interest rate of the trove is equal to the interest rate of the batch.

In summary, a user can perform two adjustments in a row every `2 * INTEREST_RATE_ADJ_COOLDOWN` (or 6 days). One way to exploit is by starting with a market interest rate of 5% in step 1 and then join a batch with a lower interest, such as 0.5%, in step 2.

Without the free adjustment, maintaining a low interest rate would be risky, since the user would risk redemptions and changing the interest later would require them to pay 7 days' worth of interest. However, since the second update is free, the user can wait until they face the risk of redemptions and only then update the interest rate back to 5% in step 3.

The profitability of this strategy depends on the frequency of redemptions and, consequently, the duration for which the user can continue paying the low interest rate.

Code corrected:

The function `BorrowerOperations.setInterestBatchManager` has been updated to charge an upfront fee whenever a user joins a batch. Previously, the fee was only charged if the interest rate was adjusted less than `INTEREST_RATE_ADJ_COOLDOWN` ago. Now, users pay the upfront fee when joining the batch (in step 2) and cannot profit from a free adjustment in step 3.

6.9 Delegation Specification

Correctness Low Version 1 Code Corrected

CS-BOLD-034

The documentation states that a `receiver` address can be chosen, who receives the collateral drawn by the Remove Manager. It does not mention that the `receiver` also receives the tokens when the owner makes a withdrawal.

In Version 1 of the code, the `receiver` receives the tokens no matter who initiates a withdrawal, whenever it is set.

Code corrected:

The code has been corrected to match the documentation. The `receiver` only receives the collateral when a withdrawal is initiated by the Remove Manager. When the owner makes a withdrawal, the tokens are sent to the owner.

6.10 Incorrect Code Comments

Correctness Low Version 1 Specification Changed

CS-BOLD-007

1. The code comments of the function `ActivePool._mintBatchManagementFeeAndAccountForChange()` state that the arithmetic is done in two steps to avoid overflow. However, the arithmetic could only underflow from the decrease.
2. The natspec in the function `BorrowerOperations._openTrove()` mentions a so-called composite debt that includes bold gas compensation. The concept of composite debt was used in Liquity V1, but is not present in Liquity V2, where the gas compensation is charged in WETH instead.
3. Similar to 2., code comments in `BorrowerOperations._applyUpfrontFee()` mention the same deprecated concept of composite debt.
4. The code comments in `TroveManager.redeemCollateral()` write the word "proportinally" instead of "proportionally".
5. The code comments in `TroveManager.redeemCollateral()` state that troves are redeemed based on their collateral ratio. However, redemption order is now actually based on the interest rate.
6. The natspec above the function `LiquityMath._decPow()` mentions two functions `TroveManager._calcDecayedBaseRate` and `CommunityIssuance._getCumulativeIssuanceFraction` that are not part of the repository.
7. The natspec above the public variable in the `StabilityPool.sortedTroves` mentions that the state variable is used for liquidations, but it is not used anywhere.
8. The code comments above the struct `TroveManager.Batch` state that the collateral is shared between troves in a batch, but collateral is kept separately.
9. The code comments above the state variable `MainnetPriceFeedBase.priceFeedDisabled` state that it should be removed after shutdown logic is implemented, but the code relies on it to return the fallback price.

#. The comments in `TroveManager._urgentRedeemCollateralFromTrove()` state that collateral has to be capped as the CR can be below 100% for urgent redemptions. However, due to the bonus multiplier, CR < 101% already requires capping the reward.

Additionally, there is a typo in the code:

- In BorrowerOperations.adjustUnredeemableTrove, batchAnnualInterestRate is written as batchAnnualInteresRate.

Specification changed:

The code comments have been updated.

6.11 Interest Delegates Are More Trusted Than Needed

Trust Low Version 1 Code Corrected

CS-BOLD-022

Interest individual delegates have some restrictions on the interest rate they can set, indicating they are not fully trusted.

However, they are not restricted in how frequently they can update the interest rate, allowing them to grief users by adjusting the interest rate multiple times and incurring the upfront fee each time.

In contrast, users are able to set a minimum interest rate adjustment period for batch managers. This allows the user to limit how often a batch manager may incur the upfront fee (if at all).

Code corrected:

The parameter `minInterestRateChangePeriod` has been added for interest rate delegatees. Delegatees must now wait for this period to pass after the last update before they can adjust the interest rate.

6.12 Missing Payable Modifier

Design Low Version 1 Code Corrected

CS-BOLD-037

The function `WETHZapper.adjustZombieTroveWithRawETH` calls the internal function `_adjustTrovePre`, which expects an ether amount to be sent for conversion to WETH. However, `adjustZombieTroveWithRawETH` lacks the payable modifier, causing any transaction that sends ether to revert.

The function is used to adjust zombie troves that were previously redeemed below the minimum debt of 2000e18 to adjust debt and collateral so that the trove is no longer considered a zombie. Zombie troves are the first in line for redemptions taking place.

If a trove's current collateral is insufficient to cover the minimum debt required to exit zombie status, the trove cannot be adjusted and will face full redemption in the next call.

Trove owners can still adjust their troves manually by calling `BorrowerOperations` directly. However, calls to the zapper function will continue to fail until it is redeployed with the necessary payable modifier.

Code corrected:

In Version 2, a payable modifier has been added to the function `adjustZombieTroveWithRawETH` to allow the function to receive ether.

6.13 Missing Validation of Troves in Urgent Redemptions

Correctness Low Version 1 Code Corrected

CS-BOLD-023

The function `TroveManager.urgentRedemption` does not validate if a trove ID exists or if the trove has any debt to redeem. This can lead to unnecessary gas consumption when a trove has no debt or has been previously redeemed.

Further it accepts Troves that have been closed or never existed, causing writes to the following storage locations by `_applySingleRedemption`:

- *rewardSnapshots[_troveId].coll = L_coll*

- *rewardSnapshots[_troveId].boldDebt = L_boldDebt*
- *Troves[_troveId].lastDebtUpdateTime = uint64(block.timestamp)`*

The dirty memory locations cannot cause invalid state modifications within the protocol, but they are returned by getter functions and can be read by external applications.

Code corrected:

Troves are now skipped if their status is not `active` or `zombie`, or if they have no debt.

6.14 Opening Troves Can Be Blocked

Design Low Version 1 Code Corrected

CS-BOLD-008

The function `BorrowerOperations._openTrove` computes the trove ID from the owner address and a trove index chosen by the caller. Anyone can open a trove on behalf of another `_owner` by providing the necessary collateral. Note that there cannot be two troves with the same trove ID:

```
vars.troveId = uint256(keccak256(abi.encode(_owner, _ownerIndex)));
_requireTroveIsNotOpen(vars.troveManager, vars.troveId);
```

An attacker can grief other users by frontrunning them and opening a trove with the same ID on behalf of the same account, setting themselves as the manager and receiver of the collateral. They could then backrun the failing transaction by withdrawing the collateral from the trove. This attack is not free, as the attacker must pay the upfront fee for the trove. However, the fee can be relatively small. At an average interest rate of 5%, they would pay 0.05 * 2000 / 52 = 1.92 BOLD, plus the gas costs for the operation. The owner of the trove can remove the attacker as manager and claim their collateral if they are able to make a transaction before the attacker.

This griefing is most problematic for multisigs or governance proposals that are executed after a time lock. Here, an attacker can potentially permanently DOS a contract from opening troves.

Code corrected:

In Version 6, the caller of the function `BorrowerOperations._openTrove` is hashed into the trove ID:

```
vars.troveId = uint256(keccak256(abi.encode(msg.sender, _owner, _ownerIndex)));
_requireTroveDoesNotExists(vars.troveManager, vars.troveId);
```

This ensures that trove IDs are unique to the caller, preventing an attacker from frontrunning troves created by calling `BorrowerOperations` directly.

All troves opened via a Zapper contract share the same `msg.sender` (the Zapper). As such the Zapper hashes the `msg.sender` into the `ownerIndex`, which is then later hashed into the trove ID:

```
function _getTroveIndex(uint256 _ownerIndex) internal view returns (uint256) {
    return _getTroveIndex(msg.sender, _ownerIndex);
}
```

This mechanism ensures that troves created through a Zapper contract are also protected from frontrunning attacks.

6.15 Out-of-gas May Lead to Shutdown

Design Low Version 1 Code Corrected

CS-BOLD-030

The MainnetPriceFeedBase shuts down the collateral branch when the static call to `latestRoundData()` reverts. The fallback uses a try-catch block as follows in `_getCurrentChainlinkResponse`:

```
// Secondly, try to get latest price data:
try _aggregator.latestRoundData() returns (
    uint80 roundId, int256 answer, uint256, /* startedAt */ uint256 updatedAt, uint80 /* answeredInRound */
) {
    // If call to Chainlink succeeds, return the response and success = true
    chainlinkResponse.roundId = roundId;
    chainlinkResponse.answer = answer;
    chainlinkResponse.timestamp = updatedAt;
    chainlinkResponse.success = true;

    return chainlinkResponse;
} catch {
    // If call to Chainlink aggregator reverts, return a zero response with success = false
    return chainlinkResponse;
}
```

There are two cases in which the external call to Chainlink reverts: Either Chainlink explicitly reverts, or the call runs out of gas. As such, the catch statement (which triggers branch shutdown) can be executed if the call to Chainlink runs out of gas. However, this will only have an effect if there is enough gas left to execute the shutdown logic. The call to Chainlink will receive 63/64 of all available gas. After reverting due to out of gas, 1/64 will be left. This means that if the shutdown logic consumes 64 times less gas than the Chainlink's `latestRoundData()`, the branch can unintentionally be shutdown even though the Chainlink oracle has not failed. The code has no access-control, so anyone can call `fetchPrice()` with any amount of gas.

At the time of writing, a call to *latestRoundData* appears to use approximately `11000` gas units. `1/64 * 11000 = 172`, which is by far not enough to execute the branch shutdown. Given these conditions, the attack is currently not feasible.

Note that an attacker can prewarm storage slots and addresses to reduce the cost of the remainder of the execution. Further, note that the gas consumption of Chainlink might increase in the future (as the contracts are upgradeable). Additionally, the gas cost of Ethereum opcodes could change in the future.

In Version 2, the function `_getCanonicalRate` was added, that retrieves the internal accounting rate of the LST tokens (rETH / ETH or stETH / ETH) and uses a similar try-catch block to handle reverting calls. Further, the oracle has been changed to return the canonical price (eth price x LST rate) when the Chainlink LST oracle has failed. This code change would allow the attack on the call to `wstETH.stEthPerToken()` even with the current gas cost of opcodes. The function wstETH.stEthPerToken() consumes approximately 37000 gas. To continue the execution, `1/64 * 37000 = 578` would be needed.

In case the system is already shutdown, an attacker could call `fetchPrice`, which falls back to `_fetchPriceETHUSDxCanonical`. Here, the call to retrieve the canonical rate could be provided with e.g. 35000 gas units to make it run out of gas. Note that afterwards the only "expensive" operations are writing `PriceSource.lastGoodPrice` to storage and reading and returning `lastGoodPrice`. The attacker could prewarm the storage slots and addresses to reduce the cost of the remainder of the execution.

```
// Get the underlying_per_LST canonical rate directly from the LST contract
(uint256 lstRate, bool exchangeRateIsDown) = _getCanonicalRate();
```

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

Code corrected:

The code has been updated to check the gas left before and after the call in `WSTETHPriceFeed._getCanonicalRate` and `MainnetPriceFeedBase._getCurrentChainlinkResponse`.

```
function _getCurrentChainlinkResponse(AggregatorV3Interface _aggregator)
    internal
    view
    returns (ChainlinkResponse memory chainlinkResponse)
{
    uint256 gasBefore = gasleft();

    // Try to get latest price data:
    try _aggregator.latestRoundData() returns (
        uint80 roundId, int256 answer, uint256, /* startedAt */ uint256 updatedAt, uint80 /* answeredInRound */
    ) {
        // If call to Chainlink succeeds, return the response and success = true
        chainlinkResponse.roundId = roundId;
        chainlinkResponse.answer = answer;
        chainlinkResponse.timestamp = updatedAt;
        chainlinkResponse.success = true;

        return chainlinkResponse;
    } catch {
        // Require that enough gas was provided to prevent an OOG revert in the call to Chainlink
        // causing a shutdown. Instead, just revert. Slightly conservative, as it includes gas used
        // in the check itself.
        if (gasleft() <= gasBefore / 64) revert InsufficientGasForExternalCall();

        // If call to Chainlink aggregator reverts, return a zero response with success = false
        return chainlinkResponse;
    }
}
```

Note that the check is conservative, as it does not include the gas used in the check itself and the gas cost for setting up the call. In practice that can lead to some calls reverting that should not, but the user can just provide a higher gas limit in that case, i.e. right before the call we only have gasBeforeCall = gasleft() - gasUsedUntilExternalCAll and after the call we only have gasAfterCall = gasleft() - gasUsedAfterExternalCall. Note that now the following condiition holds here:

$$\mathit{gasAfter} > \mathit{gasAfterCall} > \mathit{gasBeforeCall} > \mathit{gasBefore}$$

6.16 Price Limit in UniV3Exchange Is Too Strict

Design Low Version 1 Specification Changed

CS-BOLD-009

The function `UniV3Exchange.getBoldAmountToSwap` calculates the amount of BOLD to swap for a given amount of collateral tokens. It uses the user provided-specified `_maxBoldAmount` and `_minCollAmount` to set a price limit, which is then passed to the Quoter.

```
function getBoldAmountToSwap(uint256, /*_boldAmount*/ uint256 _maxBoldAmount, uint256 _minCollAmount)
    external /* view */
    returns (uint256)
{
    // See: https://github.com/Uniswap/v3-core/blob/d8b1c635c275d2a9450bd6a78f3fa2484fef73eb/contracts/UniswapV3Pool.sol#L608
    //uint160 sqrtPriceLimitX96 = _zeroForOne(boldToken, collToken) ? MIN_SQRT_RATIO + 1: MAX_SQRT_RATIO - 1;
    uint256 maxPrice = _maxBoldAmount * DECIMAL_PRECISION / _minCollAmount;
    uint160 sqrtPriceLimitX96 = priceToSqrtPrice(boldToken, collToken, maxPrice);
    IQuoterV2.QuoteExactOutputSingleParams memory params = IQuoterV2.QuoteExactOutputSingleParams({
        tokenIn: address(boldToken),
        tokenOut: address(collToken),
        amount: _minCollAmount,
        fee: fee,
        sqrtPriceLimitX96: sqrtPriceLimitX96
    });
    (uint256 amountIn,,,) = uniV3Quoter.quoteExactOutputSingle(params);
    return amountIn;
}
```

The Quoter reverts if the price limit is exceeded during the swap. However, this does not imply that swapping *_maxBoldAmount* BOLD for *_minCollAmount* of collateral is unachievable —just that the post-trade price exceeds the limit ratio (*_maxBoldAmount / _minCollAmount*). What matters for the user is the average price of the trade.

This overly strict price limit may prompt users to choose looser values for *_maxBoldAmount* and *_minCollAmount* than needed and allow MEV bots to extract more value during large swaps.

Specification changed:

The function `UniV3Exchange.getBoldAmountToSwap` has been removed in Version 2.

6.17 Shutdown Can Be Triggered Twice

Correctness Low Version 1 Code Corrected

CS-BOLD-010

The `shutdown` function in BorrowerOperations is used to shut down the branch when the TCR falls below the SCR. If the branch is already shut down, the function will revert.

However, the function calls `priceFeed.fetchPrice()`, which can also shut down the branch if the price feed is failing.

This means that if the oracle fails at the same time that the branch's TCR falls below the SCR, the branch can be shut down twice. This will emit both the `ShutDown` and `ShutDownFromOracleFailure` events, which should be mutually exclusive and may not be handled correctly by off-chain infrastructure. There don't seem to be any other side effects, as the `_applyShutdown` will have no additional effect if it is called twice in the same block.

Code corrected:

In Version 2, the `shutdown` function returns early if the oracle failure causes a shutdown. The event `Shutdown` is not emitted.

```
function shutdown() external {
    if (hasBeenShutDown) revert IsShutDown();

    uint256 totalColl = getEntireSystemColl();
    uint256 totalDebt = getEntireSystemDebt();
    (uint256 price, bool newOracleFailureDetected) = priceFeed.fetchPrice();
    // If the oracle failed, the above call to PriceFeed will have shut this branch down
    if (newOracleFailureDetected) return;

    // Otherwise, proceed with the TCR check:
    uint256 TCR = LiquityMath._computeCR(totalColl, totalDebt, price);
    if (TCR >= SCR) revert TCRNotBelowSCR();

    _applyShutdown();

    emit ShutDown(TCR);
}
```

6.18 User-provided transferFrom Source Address

Design Low Version 1 Code Corrected

CS-BOLD-011

In UniV3Exchange and CurveExchange, the `swapFromBold` and `SwapToBold` function take a `_zapper` argument. This is used in a transferFrom call.

```
boldTokenCached.transferFrom(_zapper, address(this), _boldAmount);
```

As the `_zapper` address is provided by the user, this can be used to pull tokens from any address that has given approval to the Exchange contract. For example, the zappers give unlimited approval. If a misinformed user gives approval to the Exchange contract accidentally, they could also be drained.

The zappers are not intended to ever have a balance, so the impact is limited. However, it is considered bad practice to use `transferFrom` with a user-provided address.

The `msg.sender` address could be used instead.

Code corrected:

The code has been updated to use `msg.sender` as the `from` argument in `transferFrom`.

6.19 Zapper Delegation Is Not Reset When a Trove Is Closed

Design Low Version 1 Specification Changed

CS-BOLD-012

When a trove is created through a zapper, the zapper will be set as the `addManager`, `removeManager` and `receiver` of the trove in the core system. However, a user can also create a trove without using a zapper, then set these roles to the zapper later to use its functionality.

This must be done with extreme care, as there might be delegations set in the zapper that the user is not aware of. Either the user could have had a trove with the same `troveId` earlier, that they had set delegation for in the zapper and then closed (closing a trove does not reset zapper delegation). Or the user could have gotten frontrun on trove creation, with an attacker creating a trove with the same troveId in the zapper, then immediately closing it but leaving the delegation in place on the zapper.

Zapper roles are always written to when a trove is created in the zapper, so the issue can only appear when a user opens a trove without using a zapper, then sets the zapper as the manager.

Specification changed:

In Version 2 of the protocol, each trove must have a unique troveId, preventing the reuse of delegations from closed troves.

6.20 Gas Optimizations in StabilityPool

Informational Version 5 Code Corrected

CS-BOLD-041

The code in the StabilityPool could be made more gas efficient. The following optimizations were identified:

1. In getDepositorYieldGainWithPending(), the condition totalBoldDeposits >= DECIMAL_PRECISION should be checked at the very start of the function, since no yield is distributed when totalBoldDeposits is less than 10**18.
2. In the same function, yieldGainsPending is always zero when added to newYieldGainsOwed, so the addition has no effect.
3. The withdrawFromSP() function attempts to move pending yield to depositors by calling _updateYieldRewardsSum(0) in case the sum of pending yield and deposit exceed the threshold. However, since yield is only distributed when total deposits exceed the threshold (and withdrawals cannot increase the deposit amount) this call has no effect.

Code corrected:

All aforementioned optimizations have been implemented in Version 6.

6.21 Receiver Address in Balancer Flashloan Is Reset Late

Informational Version 2 Code Corrected

CS-BOLD-043

The function BalancerFlashLoan.makeFlashLoan fixed the access control issue by setting the receiver address to the caller's address and then resetting it to the zero address after the flashloan is executed (see BalancerFlashLoan missing access control).

```
// This will be used by the callback below no
receiver = IFlashLoanReceiver(msg.sender);

vault.flashLoan(this, tokens, amounts, userData);

// Reset receiver
receiver = IFlashLoanReceiver(address(0));
```

The receiveFlashLoan function only checks the receiver address and the function does not reset the receiver address. In the current contract design, this is not a problem. However, to lower the risk of mistakes in future versions or forks of the code, the receiver could be reset in the receiveFlashLoan function.

Code corrected:

In Version 4, the code has been updated to reset the receiver at the beginning of the `receiveFlashloan`.

6.22 Backed Tokens Can Be Redeemed Unproportionally

Informational Version 1 Code Corrected

CS-BOLD-013

In CollateralRegistry, the `redeemCollateral` function has a special case for when all active branches are fully "backed". In this case the function will redeem from all active branches proportionally to the branch debt.

However, an edge case that is not explicitly handled is when there is a branch with more than zero unbacked debt, but less than the requested `_boldAmount` (and the other branches are fully backed). In this case, the function will redeem only from this branch, even though the branch will also become fully backed during the redemption process. This means that "fully backed" branches can be redeemed disproportionately.

It is not clearly specified if this is expected behavior or not.

Code corrected:

In Version 6, the redeemable amount gets capped by the total amount of unbacked debt if not all branches are fully backed.

```
// Don't allow redeeming more than the total unbacked in one go, as that would result in a disproportionate
// redemption (see CS-BOLD-013). Instead, truncate the redemption to total unbacked. If this happens, the
// redeemer can call `redeemCollateral()` a second time to redeem the remainder of their BOLD.
if (_boldAmount > totals.unbacked) {
    _boldAmount = totals.unbacked;
}
```

As a result, an attacker cannot redeem disproportionately from a backed branch, as they will only redeem until the branch becomes fully backed. The redeemer is expected to invoke the function again to redeem the remaining amount. That second redemption is proportionally between all backed branches.

6.23 CEI Pattern Violated in Adjust Trove

Informational Version 1 Code Corrected

CS-BOLD-024

The function `WETHZapper._adjustTrovePost` sends tokens after adjusting a trove. It first sends ETH via a call to the receiver and then sends BOLD tokens.

```
function _adjustTrovePost(
    ...
) internal {
    // WETH -> ETH
    if (!_isCollIncrease) {
        WETH.withdraw(_collChange);
        (bool success,) = _receiver.call{value: _collChange}("");
        require(success, "WZ: Sending ETH failed");
    }
    // Send Bold
    if (_isDebtIncrease) {
        boldToken.transfer(_receiver, _boldChange);
    }
}
```

The BOLD token is not reentrant, but the recipient could reenter the contract if it receives Ether.

Therefore, it is considered best practice to first send the bold tokens, and call the recipient last, following the Checks-effects-interactions (CEI) pattern.

Code corrected:

In Version 2, the ETH call has been moved to the end of the function.

6.24 Comments From Development

Informational Version 1 Code Corrected

CS-BOLD-025

We have identified the following remaining code comments that should be removed before deployment:

1. Several functions in the code have open TODOs.
2. Code for batch redistributions in function `TroveManager._getLatestTroveDataFromBatch()` is commented out.
3. The TroveManager contract has imports for testing imports `// import "forge-std/console2.sol";`

Code corrected:

The remaining TODOs have been removed in Version 7.

6.25 Core Debt Invariant Incorrectly Documented

Informational Version 1 Code Corrected

CS-BOLD-015

The core debt invariant in the docs is described as:

`SUM_i=1_n(trove.entireDebt) = ActivePool.aggRecordedDebt + ActivePool.calcPendingAggInterest()`

for all n troves in the branch.

However, this is incorrect as pending debt from redistributions is not included in `aggRecordedDebt`.

The correct invariant must include the `defaultPool.BoldDebt`:

`SUM_i=1_n(trove.entireDebt) = ActivePool.aggRecordedDebt + ActivePool.calcPendingAggInterest() + defaultPool.BoldDebt`

Code corrected:

The core debt invariant in the docs has been correct in Version 2.

6.26 Floating Pragma

Informational Version 1 Code Corrected

CS-BOLD-026

Liquity uses a floating pragma in some of the contract files (i.e. AddressesRegistry.sol). It is considered best practice to lock the Solidity version, to ensure the contracts are tested with the same compiler version as they are deployed.

Code corrected:

The floating pragmas have been removed. Additionally, the solidity version was upgraded from 0.8.18 to 0.8.24.

6.27 Inconsistent Input Validation by Zappers

Informational Version 1 Code Corrected

CS-BOLD-028

Liquity uses separate Zapper contracts for WETH collateral and other LST collateral tokens. The contracts are inconsistent.

The WETHZapper enforces, in _adjustTrovePre(), that debt must decrease by a positive amount:

```
require(!_isDebtIncrease || _boldChange > 0, "WZ: Increase bold amount should not be zero");
```

However, GasCompZapper._adjustTrovePre() is missing that check, allowing calls with debtIncrease set to true and boldChange equal to zero:

```
if (_isCollIncrease || (!_isDebtIncrease && _boldChange > 0)) {
    _requireSenderIsOwnerOrAddManager(_troveId, owner);
}
```

If no add manager is assigned (set to address 0), anyone can call this function, but it has no effect on BorrowerOperations.adjustTrove(), as _adjustTrove() debt increases only have an effect if they are larger than 0.

Code corrected:

All zappers now inherit from the new BaseZapper contract, which contains the _checkAdjustTroveManagers function.

This function enforces the following checks:

```
if ((!_isCollIncrease && _collChange > 0) || _isDebtIncrease) {
    receiver = _requireSenderIsOwnerOrRemoveManagerAndGetReceiver(_troveId, owner);
}

if (_isCollIncrease || (!_isDebtIncrease && _boldChange > 0)) {
    _requireSenderIsOwnerOrAddManager(_troveId, owner);
}
```

6.28 Indexed Parameters of Events

Informational Version 1 Code Corrected

CS-BOLD-035

The event `CollSent` used by the CollSurplusPool could denote the parameter `_to` with the keyword `indexed`, to make it easier for external applications to query pending collateral withdrawals.

Code corrected:

The event `CollSent` in CollSurplusPool has been updated to index the parameter `_to`.

6.29 Minting Unbacked Tokens via Redistributions

Informational Version 1 Code Corrected

CS-BOLD-036

Version 1 of the protocol had a critical rounding issue that allowed an attacker to mint unbacked tokens by manipulating the debt share exchange rate of batches via donations. In Version 2, the issue was fixed by banning all batch operations that increase debt, once a certain exchange rate limit is reached. The exception is redemptions, as explained in Rounding in debt shares calculation can mint unbacked tokens.

Under normal circumstances, redemptions decrease the debt of the user, hence one cannot mint unbacked tokens even with the exchange rate being manipulated. However, an exception to this are redemptions that trigger redistribution of debt to Troves where the amount of debt redeemed is smaller than the amount of debt redistributed. In this case, the debt of the user will go up with the redemption, and they can mint unbacked tokens.

From experience with the previous iterations of the protocol, the probability of redistributions is considered low, as liquidations typically use the StabilityPool first, falling back to redistributions only if the StabilityPool is empty. As liquidations are profitable for the StabilityPool (they are given up to a 10% discount) the StabilityPool should attract deposits when liquidations take place.

However, if one branch has no users (i.e. as the LST chosen is not very popular), or if the contract has just been deployed, then an attacker could be the only user of the protocol and could create redistributions to themselves to inflate the shares price. For this they could run a slightly modified version of the attack as described previously:

1. Manipulate the exchange rate of a Batch up to 1e8 (or any other value below 1e9) as described in Rounding in debt shares calculation can mint unbacked tokens.
2. Open Trove A ("Debt Trove") with minimum debt & collateral (= stake) AND Trove B ("Collateral Trove") with minimum debt and large amount of collateral (= stake).
3. Fully redeem Troves A and (partially) redeem B. Trove A maintains some shares in the Batch (due to rounding), but is not in the sorted troves anymore, and hence unredeemable. Trove B is *lastZombieTroveId*.

Example: Trove A - 1 debt shares, Trove B - 1 debt share, Batch A - 2 debt shares, 4e9 debt.

Note that both Trove A and Trove B have debt shares. So, any debt given to the batch (without minting shares) will be allocated by 1/2 to Trove A and 1/2 to Trove B. To exploits this, an attacker creates many redistributions: One reliable way to create a redistribution is to open a trove in a batch with a collateralization ratio just a little above the MCR. Troves in a batch can be be reliably lowered below by triggering an upfront fee update (i.e. by changing the interest rate). The attacker can then liquidate the trove in the same transaction.

4. Trigger a liquidation with redistribution, i.e. redistribution allocates 1e18 new debt to Trove B.
5. Redeem Trove B (*lastZombieTroveId*) with *boldAmount*, so that the debt is increased by a small amount, i.e. with 1e18 pending redistributions you can redeem 1e18 - 1e9 - 1 debt. In that case, no

debt shares are minted as the expression rounds down: (1e18 - (1e18 - 1e9 - 1) // 1e9 = 0.

Now repeat Step 4. - 5. until hitting an economically relevant size (i.e. 1 debt share = 2000e18 debt).

An attacker could then trigger a redistribution that transfers 1500e18 Bold and 0.5 WETH to Trove B. Trove B receives all the collateral, but all debt is given to the Batch as 1500e18 debt will be rounded to 0 debt shares. As a result, 1/2 of the debt will be owed by Trove A instead.

A sophisticated attacker could create multiple of these redemptions in a row and redeem trove A, until trove B gets liquidated. The impact is twofold:

1. The attacker can inflate the share price despite borrowing restrictions.
2. The attacker can exploit the inflated shares by creating troves and liquidating them via redistribution. One trove with a large stake receives the majority of the collateral, but the debt is given to the Batch they are in.

If the Trove has few shares in the Batch, redistributions become highly profitable as it receives collateral but no debt. This is very costly for the trove holding the majority of the debt shares. Repeatedly doing this creates "bad debt" (unbacked tokens) as the trove with the majority of the debt shares cannot be liquidated quickly enough.

This attack requires the StabilityPool of the branch to be empty and is only profitable if there is very little collateral on the branch that does not belong to the attacker. This is most likely to happen in 2 cases:

1. The system is freshly deployed and one of the branches does not have any troves yet.
2. One branch's collateral token becomes so unpopular that all users close their positions and withdraw their collateral from the branch.

In case 1., there is likely not much collateral in the system yet, so the impact is limited. In case 2., there may be another branch with high value that could lose a large amount of money through unbacked minting.

Note that the attacker cannot close the position if the total collateralization ratio is not above the CCR after their Trove is closed. This can only be achieved if other users add more collateral to the system, making the probability of this attack being exploited very low.

Code corrected:

In Version 2, the issue was resolved by burning all shares from a trove when all its debt is getting repaid. This prevents the attacker from redeeming the "Debt Trove" while retaining some shares in step 2.

6.30 Misleading Function Names in Zapper

Informational Version 1 Code Corrected

CS-BOLD-017

In GasCompZapper and LeverageLSTZapper, the same function names are used as in their counterparts WETHZapper and LeverageWETHZapper.

For example, `adjustTroveWithRawETH()`: In WETHZapper the function name is accurate, as raw ETH can be wrapped or unwrapped to be used in the function. However, in GasCompZapper the function never operates with raw ETH.

Code corrected:

The relevant function names in the GasCompZapper have been adjusted to no longer reference raw ETH.

6.31 Missing Events

Informational Version 1 Code Corrected

CS-BOLD-029

The functions `AddRemoveManager._setAddManager()` and `AddRemoveManager._setRemoveManager()` do not emit events when the manager is set. This can make it difficult to track changes to the manager.

Since Trove NFTs can be sent to other addresses, the missing events make it hard for the owner to track the current add and remove manager addresses and could lead to users receiving Trove NFTs without knowing the current manager addresses.

Code corrected:

In Version 2, the events `AddManagerUpdated` and `RemoveManagerAndReceiverUpdated` are emitted whenever managers are set, changed, or removed.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Unimplemented Function Can Be Called

Informational Version 2

CS-BOLD-038

The contracts GasCompZapper and WETHZapper are not marked as abstract and can be used as base contracts.

They define the functions `receiveFlashLoanOnOpenLeveragedTrove`, `receiveFlashLoanOnLeverUpTrove` and `receiveFlashLoanOnLeverDownTrove` with empty bodies, performing no operations. This means these functions can be called, but they will not perform any actions. If these functions are not intended to be called, a caller might expect them revert on the call to prevent misuse.

7.2 Bypassing Collateral Adjustment Check

Informational Version 1 Acknowledged

CS-BOLD-014

The check `BorrowerOperations._requireValidAdjustmentInCurrentMode` prevents reducing collateral in undercollateralized troves (ICR < MCR).

It can be bypassed through self-redemptions, by first adjusting the interest rate of the trove to be the lowest in the branch, then redeeming from it.

Acknowledged:

Liquity responded:

```
That workaround would:

Reduce the size of an undercollateralized trove
Improve the ICR of an undercollateralized trove
So it’s not bad for the system.
That condition prevents the final state to be undercollateralized.
So the only way to use that workaround and still end up with ICR < MCR
is the trove was already undercollateralized.
It doesn’t allow to convert a healthy trove into an undercollateralized one.

Therefore, we accept it for simplicity of the code.
```

7.3 Gas Optimizations

Informational Version 1 Code Partially Corrected

1. The function `ActivePool.mintBatchManagementFeeAndAccountForChange()` is protected by the `_requireCallerIsBOorTroveM()` modifier but is only called by the TroveManager contract and never the BorrowOperations.
2. The interest calculation in `ActivePool.calcPendingAggInterest()` could short-escape and return `0` if the last update has been at the current timestamp. Similarly, the function `ActivePool.calcPendingAggBatchManagementFee()` could short-escape early.
3. The function `TroveManager._getLatestTroveDataFromBatch()` reads the entire structure `batch` from storage, but only uses one value from it.
4. The modifier `CollateralRegistry._requireBoldBalanceCoversRedemption()` asserts that the user's balance is smaller than the total supply, but this invariant is already enforced by OZ's token implementation.
5. The function `TroveManager._redistributeDebtAndColl` can skip the call `sendCollToDefaultPool` when the collateral to redistribute is zero.
6. The modifier `UniV3Exchange._requireCallerIsUniV3Router()` is never used.
7. The function `UniV3Exchange.uniswapV3SwapCallback()` is never used.
8. The `BorrowerOperations.adjustTroveInterestRate` function only charges an upfront fee when the new interest rate is not equal to the previous rate, but this is not possible due to the check in *_requireAnnualInterestRateIsNew*.
9. The immutable `StabilityPool.sortedTroves` is never used.
10. The modifier `BorrowerOperations._requireIsShutDown()` is never used.
11. The function `WSTETHPriceFeed._fetchPrice()` calls `wstETH.stEthPerToken()`, which in turn calls `stETH.getPooledEthByShares`. It would be cheaper to call stETH directly.
12. The TroveManager contract could define state variables only set once in the constructor as immutable: collateralRegistry, sortedTroves, boldToken, collSurplusPool, gasPoolAddress, stabilityPool, borrowerOperations, troveNFT.
13. The BorrowerOperations contract could define state variables only set once in the constructor as immutable: troveManager, gasPoolAddress, collSurplusPool, sortedTroves, boldToken.
14. The ActivePool contract could define state variables only set once in the constructor as immutable: interestRouter, boldToken.

Version 2:

15. The function `priceToSqrtPrice` and `_zeroForOne` in `UniV3Exchange` are not used anywhere.
16. The function `TroveManager._requireBelowMaxSharesRatio` could first check the condition `_checkBatchSharesRatio` before computing the product `_currentBatchDebtShares * MAX_BATCH_SHARES_RATIO` to short-circuit the evaluation.

Code partially corrected:

The optimization points 6, 7 and 14 have been implemented in Version 2.

7.4 Insufficient Gas Compensation Could Mint Bad Debt

Informational Version 1 Risk Accepted

CS-BOLD-016

The system relies on the gas compensation to be sufficient to cover the gas costs of the liquidation. If the gas cost of a liquidation is higher than the gas compensation, the system can end up with bad debt. This can happen due to falling collateral prices, in which case no new debt can be directly minted to insufficiently collateralized troves. However, bad debt can still be newly minted through interest and batch fee accrual, if the gas is expensive for extended periods of time.

An attacker could join their own batch and set the batch management fee to the maximum amount (currently 100%) to mint themselves tokens. If liquidations stay expensive for months, this could be profitable. A user could also create a batch with ICR = 1.1 and the maximum interest rate (currently 100% APR) and join the stability pool to which the interest is minted. If it is unprofitable to liquidate the trove, after about 4 days unbacked tokens will be minted to the stability pool (as the interest will surpass the 10% overcollateralization).

However, both of these attacks can easily be punished by anyone willing to make an unprofitable liquidation, so they incur a high risk for the attacker. As timely liquidations are profitable for the stability pool, anyone who is part of the stability pool also has an additional incentive to liquidate. These factors make the described attacks unlikely to be profitable.

Note that there is a variable gas compensation dependent on a trove's collateral amount, so liquidations of larger troves are more likely to be profitable than those of smaller troves.

In conclusion, these attacks are likely not of concern unless the gas prices on Ethereum significantly and permanently increase in the years to come. However, they illustrate how the debt of the system can increase (and be minted to an attacker), even when troves fall below the minimum collateralization ratio.

Risk accepted:

Liquity is aware of this issue, but has decided to keep the code unchanged, providing the following reasoning:

```
Indeed, we are aware that the liquidation mechanism is not perfect.
We’ve had long discussions about it internally.
As mentioned in the issue, big troves are prioritized,
so it’s very unlikely that this becomes a problem for the system health.
Based also on Liquity v1 experience, we are confident that the mechanism will be robust enough.
```

7.5 Price Deviation in Composite Price Feed

Informational Version 1 Risk Accepted

CS-BOLD-031

The `RETHPriceFeed` calculates RETH/USD price as the product of the RETH/ETH and ETH/USD price fees in `CompositePriceFeed._fetchPrice()`.

The Chainlink RETH/ETH price feed has a deviation threshold of 2%, and the ETH/USD price feed has a deviation threshold of 0.5%. So, the combined price can deviate up to approximately 2.5% from the actual price before it gets updated.

A branch can shut down for two reasons:

1. Oracle failure.

2. Total collateralization ratio (TCR) drops below the Shutdown Collateralization Ratio (SCR).

The second event can occur when the Liquid Staking Token (LST) price de-pegs due to slashing events or market volatility. In such cases, the protocol provides a 1% bonus on the oracle price in `TroveManager.urgentRedemption()`.

However, this bonus may not be sufficient, as the oracle price can deviate up to 2.5% from the actual price before it gets updated.

Additionally, the price feed queries the canonical rate from Rocket Pool that updates every 24 hours and then takes the minimum of Chainlink Price and canonical rate. In the worst case the urgent redemptions could be delayed by 24 hours, as it may happen that neither Chainlink nor the canonical rate update within this period.

Risk accepted:

Liquity is aware of the issue and provided the following response:

```
This a known issue and accepted risk. We already accept the urgent redemption bonus may be insufficient in branch shutdown, especially if the branch is using lastGoodPrice.
Besides we have checked the historical deviation of RETH/ETH and it seem in practice was way more accurate, so that 2% seems theoretical and unlikely to be hit.
Of course no warranty that it cannot happen at all, but, again, we accept the risk.
```

7.6 Small Redemptions Do Not Increase Base Rate

Informational **Version 1** **Acknowledged**

CS-BOLD-018

The function `CollateralRegistry._getUpdatedBaseRateFromRedemption` calculates the base rate from the share of bold tokens that are redeemed.

```
// get the fraction of total supply that was redeemed
uint256 redeemedBoldFraction = _redeemAmount * DECIMAL_PRECISION / _totalBoldSupply;
```

As the redeemedBoldFraction is rounded down, splitting a redemption into multiple smaller ones can reduce the fee paid. The most extreme case are redemptions with `redeemAmount < totalBoldSupply / 1e18`, which will have their fraction rounded to zero, meaning they will not increase the base rate at all. However, multiple redemptions will incur higher gas costs.

Acknowledged:

The Liquity team is aware of this behavior and has provided the following description:

```
For “the most extreme case”, assuming a total supply of 10B (1e10 * 1e18), the redeem amount would be 1e10 wei, i.e. 1e-8 BOLD.
It doesn’t make sense to redeem such amount, as the amount paid in gas would be much higher.
```

7.7 Upfront Fee Is Zero for Small Borrows

Informational **Version 1** **Acknowledged**

CS-BOLD-033

The function `BorrowerOperations._calcUpfrontFee` calculates the upfront fee as

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

Note that the period is currently 1 week, and the weighted debt is equal to the trove debt times the average interest rate in the system.

Hence, the upfront fee is rounded to zero when

$debt * averageinterestrate / 52e18 < 0$

When the average the interest rate is 0.5% (5e15), then anyone borrowing less than 10400 (52e18 / 5e15) wei pays no fees.

Similarily, the interest rate charged on small troves can be rounded to zero. For a trove with 1e8 debt and an interest rate of 5e15, updating the interest rate every 12 seconds would not result in any interest owed:

$1e8 * 5e15 * 12 < 31536000 * 1e18$

Acknowledged:

Liquity acknowledged the issue and has decided not to make a change, giving the following response:

```
This is only possible for zombie troves.
10k wei (1e-14 BOLD) seems a negligible amount, both in at user and system wide level.
```

7.8 Upfrontfee Can Bring Troves Below MCR

Informational **Version 1** **Code Partially Corrected**

CS-BOLD-019

The `_applyUpfrontFee` function in BorrowerOperations contains the `_requireICRisAboveMCR()` check. This ensures that an upfrontfee from an adjustment cannot bring the trove below the MCR.

However, the `setBatchManagerAnnualInterestRate` function, which is used to adjust the interest rate of a batch, does not contain such a check (as troves in a batch can have different ICRs). As a result, it is possible for an upfrontfee of a batch interest adjustment to bring a trove below the MCR.

The batch interest rate can only be adjusted at most once per second, so if a trove is pushed below the MCR, it should be liquidated before the next adjustment can happen. If this limitation was not in place, batch interest could be adjusted many times in a block and potentially create bad debt when the ICR falls below 100%.

Other ways of bringing a Trove's ICR below MCR are:

- Batch management fee is charged
- Interest is charged
- Collateral price falls

Code partially corrected:

In **Version 5**, a buffer collateralization ratio of 10% has been introduced. This ensures that whenever a Trove owner adjusts their trove that is part of a batch or joins a batch, their collateralization ratio remains sufficiently above the Minimum Collateralization Ratio (MCR).

Previously, a user could create a liquidation within a single transaction by following these steps on the WETH collateral branch (staked ETH branches have a similar exploit):

1. Create a Trove with a collateralization ratio of exactly 110% in Batch A.
2. Charge an upfront fee in Batch A, which immediately lowers the collateralization ratio of the Trove below 110%, making the Trove liquidatable.
3. Liquidate the Trove.

With the buffer collateralization ratio, it is no longer possible to execute all three steps in a single transaction. The upfront fee is equivalent to one weeks' worth of interest, with a maximum annual interest rate of 250%. This caps the maximum upfront fee at:

`250`

Since the enforced 10% collateral buffer exceeds this maximum upfront fee, a single transaction can no longer immediately push a Trove into liquidation.

A user can still achieve the same outcome over time:

1. Create a Trove with a 120% collateralization ratio in Batch A.
2. Wait until the Trove's collateralization ratio naturally declines (due to interest accrual, price changes, or management fees) until it approaches 110%.
3. Charge an upfront fee in Batch A, reducing the collateralization ratio to below 110%, making the Trove liquidatable.
4. Liquidate the Trove.

Unlike the previous exploit, steps 3 and 4 can still be performed in the same transaction, but some time must pass between step 1 (creating the Trove) and step 3 (applying the upfront fee).

However, unlike a trove that is not in a batch, steps 3 and 4 can be carried out in the same transaction so that the owner of the batch can perform the liquidation with certainty. For a trove that is not in a batch, the owner of the trove will have to wait for the collateralization rate to fall naturally and generally has no guarantee that they will able to liquidate their trove. This opportunity could instead be taken by another user, such as an MEV bot.

7.9 rETH Address Might Change

Informational **Version 1** **Acknowledged**

CS-BOLD-020

The rocket pool protocol stores addresses in a storage contract. The protocol's codebase indicates that addresses should not be used directly but be retrieved on-chain. However, the collateral token address is set once in the `AddressesRegistry` and cannot be updated.

In practice, it seems unlikely that the rETH token's address will change due to other integrations. Additionally, it is unspecified how integrators should properly handle such a migration.

Acknowledged:

Liquity has reached out to the Rocketpool team to confirm that, while technically possible, the rETH address is not expected to ever change.

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 Add Manager Can Increase Stake

Note Version 1

The `addManager` is generally not trusted, as it can only improve the collateralization ratio of a trove. If the manager is set to address 0, anyone can add collateral to a trove.

However, increasing the collateral of a trove also increases its stake. In extreme scenarios where redistributions give more debt to active positions than collateral, an attacker could send collateral to another trove owner to increase their share in the bad redistributions.

Users who want to avoid this can set the `addManager` to the same address as the `owner`, which will disallow others from increasing their collateral.

8.2 CCR and SCR Considerations

Note Version 1

Note that the CCR and SCR must be set in a way that they are not too close to each other to prevent intentional triggering of branch shutdown.

In particular, the CCR should be higher than the SCR by an amount that is greater than the largest expected price change in a single oracle update. Otherwise, an attacker could intentionally trigger shutdown by frontrunning the oracle update.

This would be similar to the known attacks for triggering recovery mode in Liquity V1.

The currently proposed values of `CCR = 1.5` and `SCR = 1.1` appear far enough apart that these attacks are not feasible. Any future deployments that use closer values should consider this attack vector.

8.3 Delegations Are Deleted on Liquidation

Note Version 1

Liquidations return surplus collateral to the trove owner. However, only the owner of the trove can reclaim the surplus. Delegated accounts cannot. Note that liquidations trigger the *BorrowerOperations.onLiquidateTrove* hook, so all delegations for that trove ID are deleted on liquidation.

If the trove owner is a smart contract, it must implement the function to claim collateral. It cannot rely on another contract via delegation. Not having the ability to call `claimCollateral()` risks losing access to the surplus collateral in case of liquidation.

8.4 Frontrunning Considerations for Off-Chain Infrastructure

Note Version 1

Due to the risk of frontrunning, integrators should make sure to consider the following points when writing off-chain infrastructure:

1. The function `CollateralRegistry.redeemCollateral` calculates the redemption fee based on the user-provided maximum redeemed amount, rather than the actual amount. If the user sets a `_maxIterationsPerCollateral`, they may not receive the maximum amount. This will result in receiving less collateral than expected, while still paying the same percentage fee. Similarily, if all but one branch is backed, a redemption could redeem all collateral on a branch and then escape.
2. The user can also set a `_maxFeePercentage`, which will revert the transaction if the actual fee percentage is higher than expected. If the user is frontrun, the prior redemption will increase the basefee and reduce the total BOLD supply, so the fee will increase. This can cause their redemption to revert, as the user's specified `_boldAmount` would push the fee above the specified maximum. The user may want to write a wrapper contract that calculates the `_boldAmount` that can be redeemed without going above the `_maxFeePercentage`.
3. The execution order of liquidations can influence subsequent liquidations through redistributions. A profitable redistribution could restore a previously liquidatable trove, while an unprofitable redistribution might cause a previously healthy trove to become liquidatable.

8.5 Fully Backed Branches Can Have Low Interest Rates

Note Version 1

Redemptions are distributed between branches according to their "unbackedness". If a branch has more BOLD in the stability pool than it has outstanding debt, it is considered "fully backed" and no redemptions are routed to it.

Usually, users are encouraged to pay a high interest rate to avoid redemption. However, in the case of a fully backed branch, no redemptions can happen in that branch, so there is no longer an incentive to pay more than the minimum interest rate. As a result, it should be expected that the interest rate on fully backed branches will be the minimum interest rate.

The troves at the minimum interest rate risk that the branch will become unbacked, as this will re-enable redemptions. This can happen through withdrawals from the stability pool, liquidations, or through more debt creation on the branch. However, if this happens gradually, there will still only be a small percentage of redemptions executed through that branch (affecting the latest troves to adopt the strategy), as its unbackedness will still be quite low. As soon as the branch becomes unbacked, the trove owners can react by increasing the interest rate again (or depositing more BOLD to the stability pool). If the owners need to adjust, they may need to pay an upfrontfee for doing so.

This problem could be self-correcting if stability pool depositors withdraw when the average interest rate of the branch drops, as this will result in a lower yield paid to the stability pool. It is unclear how this would play out in practice.

8.6 Inconsistent Use of Receivers in Zappers

Note Version 1

The Zappers have a `receiver` per trove. However, it is not used consistently:

- In `adjustTroveWithRawETH` the Zapper sends the WETH and BOLD to the `receiver` defined in the Zapper itself and any "extra" BOLD to the `msg.sender`.
- In `closeTroveFromCollateral` the Zapper sends all funds to the `receiver` defined in the Zapper itself.
- In `openTroveWithRawETH` the Zapper sends all BOLD to the `msg.sender`.

Integrators and users should be aware of the when the `receiver` is used and when it is not used.

8.7 Integration Notes for Smart Contract Devs

Note **Version 1**

The following function behaviors may be unexpected and must be taken into account when writing contracts that integrate with Liquity V2:

The `repayBold` and `adjustTrove` functions in BorrowerOperations can be used to repay debt. If the amount specified to repay would result in the trove falling below `MIN_DEBT`, then only the amount required to reach `MIN_DEBT` will be repaid. The remaining amount will be left with the caller. Integrators must not expect a revert if the amount specified is impossible to repay exactly and must return leftover amounts in the calling contract to the user.

The `withdrawFromSP` function in StabilityPool takes an `_amount` parameter. If the user has a BOLD deposit that is smaller than `_amount`, the user's full balance will be withdrawn. Integrators must not expect a revert if `_amount` is impossible to withdraw and must account for the actual withdrawal amount.

Any operation on a trove within a batch can influence other troves in the same batch. For instance, if we retrieve the data for Trove A by calling `getLatestTroveData`, then close another trove within the same batch, and subsequently call `getLatestTroveData` for Trove A again, the debt of trove A may have increased due to rounding errors. Integrators must expect that the state of a trove can change between calls, even if the trove itself is not directly modified.

8.8 Interest Rate Adjustments Below CCR

Note **Version 1**

When the system's TCR falls below the CCR, adjustments to troves that create debt are not allowed. As a result, paying the upfrontfee for a premature interest rate adjustment is also not allowed. This means that interest rate adjustments are only allowed after the `INTEREST_RATE_ADJ_COOLDOWN` period has passed since the last adjustment.

Users should be aware of this and take caution when the TCR gets close to the CCR, as they may be unable to increase their interest rate once the threshold is passed.

8.9 Manipulating Bold Supply With Flashloans

Note **Version 1**

When opening a trove, a user pays an upfront fee equal to the average interest for 7 days. The interest rate can be as low as 0.5%, thus the upfront fee can be as low as 0.5%/52 = 0.0096%. The interest rate calculation also takes the interest of the newly opened Trove into account. Hence, a large trove can have a significant impact on the average interest rate and by extension the upfront fee of opening the trove.

A user could be incentivized to inflate the BOLD supply using a flashloan, since the redemption fee depends on the percentage of the BOLD supply being redeemed.

This can be particularly profitable when large amounts of BOLD are getting redeemed. For example, when a user attempts to redeem 50'000 BOLD of a total supply of 1 million BOLD, they would pay a fee of 5% of the redeemed amount. If the supply is doubled to 2 million BOLD, the fee would be halved to 2.5%.

Assuming that the average interest rate is 1% after the big trove is opened, this behavior is profitable:

$$0.05 * 50000 < 0.025 * 50000 + 1e9 * 0.01/52$$

Note that the flashloan can be taken out from the smallest branch to have a larger impact on the average interest rate in that branch, which will make the upfront fee cheaper. If the user wants to avoid self-redemption, they can add the minted BOLD to the stability pool to reduce the percentage of redemptions that are routed through that branch.

8.10 No Interest Paid on Pending Redistributions

Note Version 1

When a liquidation causes a redistribution, the redistributed debt is accounted for in the DefaultPool. Whenever a trove is touched, its pending redistributions are moved from the DefaultPool to the trove.

Note that no interest is paid on pending redistributed debt. Interest will only start accruing once the debt is moved to the trove.

Also note that the approximate interest rate calculated for the upfrontfee ignores the pending redistributed debt. As the pending debt has an interest rate of 0%, the average interest rate can therefore be higher than the effective interest rate paid on all outstanding BOLD.

8.11 Possible Attacks Against Zapper Users

Note Version 6

For Zapper operations that involve a swap, users provide a part of the input data, e.g. for the `closeTroveFromCollateral` function of the `WETHZapper` the users provide `_flashLoanAmount` which is (after fee subtraction) swapped against the entire debt of the trove. As the trove state might change, after the user has signed and sent the transaction, certain attacks are possible.

Extracting Redistribution Profits:

Preconditions: A user calls a function like `closeTroveFromCollateral` of the `WETHZapper`, then a significant redistribution happens and the user's trove is redeemable.

Possible Attack: By performing a redemption so that the `minExpectedCollateral` just passes, the attacker can extract the redistribution profits the trove just received, as they are now part of the trading slippage. Any original trading slippage can additionally be extracted.

Extracting Redemption Fees:

Preconditions: A user calls a function like `closeTroveFromCollateral` of the `WETHZapper`, then a significant redemption with a high fee happens and the user's trove does not have an `AddManager` in `BorrowOperations`.

Possible Attack: The attacker donates collateral, so that the `minExpectedCollateral` just passes. Then, the attacker can extract the redemption fee, that was just received, as it is now part of the trading slippage. Any original trading slippage can additionally be extracted.

8.12 Sending NFTs Does Not Reset Delegation

Note Version 1

Troves in Liquity V2 are transferable NFTs. This allows the user to transfer a trove, which means it could be sold on a marketplace. Users purchasing a trove should be aware that the delegation of the trove is not reset when the trove is transferred. This means the seller could set themselves as the remove manager and receiver before the transfer is executed (potentially setting it at the last second, frontrunning the purchase), then remove collateral from the trove after the transfer is completed.

As a result, purchasers of troves should be cautious and ensure that the purchase transaction includes a reset of the delegation.

Also note that delegation can be set in the zappers, so users must check this before delegating their trove to a zapper.

8.13 Trove Shares Exchange Rate Invariant

Note Version 1

The function `TroveManager._updateBatchShares()` rounds down the shares removed from the batch when debt is decreased. Hence, any operations that decrease the debt level (i.e. redemptions, lowering the debt level of troves) reduce the amount of debt per debt share due to rounding.

We have not identified any strategy that could reduce the exchange rate below the initial exchange rate of one debt token per debt share.

If it was possible to bring the value of a debt share below one debt, it may be possible for further rounding issues to make debt shares very cheap (i.e., totalDebtShares >> totalBatchDebt). This could result in an overflow of the debt share calculation. For example, when the `currentBatchDebtShares` is `1e60` and the `debtDecrease` is `1e18`, then the calculation to get the trove of a debt *recordedDebt* would overflow.

```
function _getLatestTroveDataFromBatch() internal view {
    Trove memory trove = Troves[_troveId];
    Batch memory batch = batches[_batchAddress];
    uint256 batchDebtShares = trove.batchDebtShares;
    uint256 totalDebtShares = batch.totalDebtShares;

    ...

    if (totalDebtShares > 0) {
        _latestTroveData.recordedDebt = _latestBatchData.recordedDebt * batchDebtShares / totalDebtShares;
```

This could prohibit certain positions from getting liquidated and be exploited to mint unbacked tokens to managers.

Additionally, full redemptions can leave some debt in the trove. For example, when total batch shares are 4000e18, debt is 4000e18 + 5, and the user has 2000e18 shares, the user debt is calculated as:

*user.debt = batch.debt * user.shares / batch.shares = (4000e18 + 5) * 2000e18 / 4000e18 = 2000e18 + 2*

However, redeeming that user debt burns:

*user.shares = batch.shares * user.debt / batch.debt = 4000e18 * (2000e18 + 2) / (4000e18 + 5) = 2000e18 - 1*

Thus, the user would end up with *shares = 1* left after the redemption.

In conclusion, the rounding can potentially lead to unexpected behavior. If there is a way to trigger rounding that causes the ratio of `debt:debtshares` to go below `1`, this could have catastrophic consequences. However, we have not been able to identify a transaction sequence that causes this.

It should be seen as a core protocol invariant that the `debt:debtshare` ratio is never allowed to go below 1.

In Version 2 the function TroveManager._updateBatchShares does not round down when fully closing a position:

```
// Subtract debt
// We make sure that if final trove debt is zero, shares are too (avoiding rounding issues)
// This can only happen from redemptions, as otherwise we would be using _removeTroveSharesFromBatch
// In redemptions we don’t do that because we don’t want to kick the trove out of the batch (it’d be bad UX)
if (_newTroveDebt == 0) {
    batches[_batchAddress].debt = _batchDebt - debtDecrease;
    batches[_batchAddress].totalDebtShares = currentBatchDebtShares - Troves[_troveId].batchDebtShares;
    Troves[_troveId].batchDebtShares = 0;
} else {
    batchDebtSharesDelta = currentBatchDebtShares * debtDecrease / _batchDebt;

    Troves[_troveId].batchDebtShares -= batchDebtSharesDelta;
    batches[_batchAddress].debt = _batchDebt - debtDecrease;
    batches[_batchAddress].totalDebtShares = currentBatchDebtShares - batchDebtSharesDelta;
}
```

Note on the audit process: At the beginning of the audit, it was already known to Liquity that fully redeemed troves can end up with a non-zero amount of debt shares. The GitHub issue related can be found here.

8.14 Zapper Remove Manager Requires Increased Trust

Note Version 1

The zappers implement the same delegation scheme as the core system. There is a removeManager and a receiver, that can be set to different addresses. In the core system, the removeManager cannot directly profit or cause losses by making malicious changes, because the receiver will receive the funds.

In the zappers, the removeManager can directly profit and cause losses by making malicious changes, because they can decide the parameters of the swaps made in the leverage functions. Swapping at bad exchange rates (and sandwiching those swaps) can cause losses to the owner of the trove.

As a result, the trust required in a removeManager that is not also the receiver is higher in the zapper than in the core system.