



coinspect
You build, we defend.



Smart Contract Audit
Bold Core
December 2024



Bold Core Smart Contract Audit

Version: v241231

Prepared for: Liquity

December 2024

Security Assessment

1. Executive Summary
2. Summary of Findings
3. Scope
4. Changes from Liquity V1
5. Design and Risk Mitigation Mechanisms
6. Decentralization Analysis
7. Code quality and Testing
8. Fixes and Changes
9. Detailed Findings

BOLD-01 - Leveraged operations will not work as approvals are spontaneously reset

BOLD-02 - Attackers can claim minted yield from the Stability Pool without leaving their funds deposited

BOLD-03 - Adversaries can manipulate batch's debt/shares ratio by deflating the debt

BOLD-04 - Unused code

BOLD-05 - Single oracle failure stops redemptions across all branches triggering a depeg event

BOLD-06 - Wrong implementation of an LST oracle forces protocol re-deployment

BOLD-07 - Amount of parameters when opening troves facilitates deceiving users

BOLD-08 - Redemption decay rate is zero before expected

BOLD-09 - Missing bold amount check in urgent redemptions

10. Appendix: Protocol Overview

10.1 Protocol Architecture and Yield Generation

10.2 Asset and Position Management

10.3 Economic Mechanisms

10.4 Risk Management

10.5 Zappers

11. Disclaimer

1. Executive Summary

In **October 2024**, **Liquity** engaged **Coinspect** to perform a smart contract audit of Liquity Bold Core. The objective of the project was to evaluate the security of the smart contracts that implement the second version of the protocol.

Liquity V2 is a decentralized borrowing protocol that enables users to mint BOLD stablecoin tokens using multiple forms of collateral, such as WETH and Liquid Staking Tokens (LSTs). The system introduces several advancements to allow users to optimize their borrowing condition such as user-set interest rates. These improvements also include the representations of troves (i.e. debt positions) as NFTs. Users can also open troves through batches, which enables efficient adjustment of interest rates for multiple troves through batch managers.

Core stability mechanisms from Liquity V1, such as the redemption process and liquidations, are adapted to the multi-collateral framework and user-set interest rates. New features include delegation options for trove management, allowing borrowers to delegate control to third parties. The system is designed to achieve capital efficiency and stability, with mechanisms such as redemptions and batch management designed to provide solvency and maintain the stablecoin's value. For users, these enhancements translate to greater flexibility in managing debt positions and improved system stability.

Extensive documentation, rigorous prior audits by multiple firms, and comprehensive testing have contributed to a high-quality codebase. The protocol's advanced development stage is evident in the severity and scope of the issues identified during this audit.

Coinspect identified two significant findings during the security assessment: **BOLD-01** demonstrates that an approval reset renders Leveraged Zappers unusable, while **BOLD-05** highlights how a failure in a single oracle can halt redemptions across all branches, causing a depeg of the stablecoin.

While Liquity V2 incorporates innovative features, such as multi-collateral support and batch management, these enhancements add complexity, which introduces unique operational risks. For instance, decentralized oracles and trust on batch managers could influence the system's overall resilience.

However, these risks have been mitigated with fallback mechanisms and batch-level risk controls. Overall, Bold Core is a robust implementation of an innovative DeFi protocol.


Solved


Caution Advised


Resolution Pending

High	High	High
0	0	0
Medium	Medium	Medium
2	0	0
Low	Low	Low
0	0	0
No Risk	No Risk	No Risk
7	0	0
Total	Total	Total
9	0	0

2. Summary of Findings

This section provides a concise overview of all the findings in the report grouped by remediation status and sorted by estimated total risk.

Solved issues & Recommendations

These issues have been fully fixed or represent recommendations that could improve the long-term security posture of the project.

Id	Title	Risk
BOLD-01	Leveraged operations will not work as approvals are spontaneously reset	Medium
BOLD-05	Single oracle failure stops redemptions across all branches triggering a depeg event	Medium
BOLD-02	Attackers can claim minted yield from the Stability Pool without leaving their funds deposited	None
BOLD-03	Adversaries can manipulate batch's debt/shares ratio by deflating the debt	None
BOLD-04	Unused code	None
BOLD-06	Wrong implementation of an LST oracle forces protocol re-deployment	None
BOLD-07	Amount of parameters when opening troves facilitates deceiving users	None
BOLD-08	Redemption decay rate is zero before expected	None
BOLD-09	Missing bold amount check in urgent redemptions	None

3. Scope

The scope was set to be the repository at <https://github.com/liquity/bold/> at commit `aa8361269bb505de06afbe5a6646160cc9a935ef`.

For this security assessment, Liquity requested Coinspect to focus on the following features and components:

- Stability Pool
- Crypto Economics
- Zappers
- Batch Delegation

Coinspect reviewed these features after the project experienced changes and fixes for previously reported issues.

4. Changes from Liquity V1

This section highlights the key updates and enhancements introduced in Liquity V2, focusing on improved functionality, expanded features, and refined mechanisms that build upon the foundation of the original protocol.

- 1. Collateral Support:** The protocol migrated from a single-collateral, ETH, to a multi-collateral system in V2. The new version supports WETH and Liquid Staking Tokens (LSTs) including rETH and wstETH (but not ETH), enabling users to utilize multiple types of ETH derivatives as collateral.
- 2. User Set Interest Rates:** The system changed from having no interest rates on borrowed LUSD to implementing a dynamic, market-driven approach with user-set annual interest rates that borrowers can modify at any time. Interest now accrues continuously and compounds discretely when the trove is touched, enabling yield generation while allowing borrowers to optimize their positions based on market conditions.
- 3. Yield Distribution:** While V1 offered no yield from interest, V2 introduced a yield generation and distribution system. BOLD yields generated from trove interest are systematically split between the Stability Pool participants and LP incentives, creating additional value streams for protocol stakeholders.
- 4. Redemption Mechanics:** The redemption system underwent a redesign, changing from being based on trove's ICR ordering to a routing system. Redemptions are now proportionally distributed across branches based on their relative "unbackedness" and within each branch, redemptions target troves ordered by interest rate from lowest to highest, with the objective to create a more efficient and balanced redemption process.
- 5. Token Implementation:** The implementation changed from a straightforward LUSD stablecoin to BOLD stablecoin with integrated NFT functionality. Troves are now represented as NFTs, enabling improved transferability and management of positions while maintaining the core stability mechanisms of the protocol. They are meant to be transferred between different owner accounts (e.g. hot and cold wallets). Trading them in a marketplace poses additional risks. For example, trove delegations and management accounts are not cleared after a transfer, the trove owner is able to reduce the trove's collateral before a trade, among other adversarial scenarios that could be used by malicious sellers on marketplaces.
- 6. Recovery Mode:** The protocol replaced the Recovery Mode mechanism (which activated when system fell below 150% collateralization) with a Critical Collateral Ratio (CCR) restriction system. This new approach

provides more granular control over system stability seeking to keep the protection against undercollateralization.

7. **Unredeemable troves:** The protocol shifted from automatically closing troves during redemptions to keeping them open, introducing a new concept of "unredeemable troves" when debt falls below minimum. This change aims to provide more flexibility for borrowers while maintaining system stability through minimum debt requirements.
8. **Management System:** V2 introduced a delegation framework, featuring both individual and batch delegation systems. This allows more sophisticated management of interest rates and trove adjustments, with the goal of enabling more efficient position management and potential for management services.
9. **Liquidation Mechanics:** The liquidation system changed from simple full collateral seizure to a new mechanism with variable liquidation penalties. The new system allows for potential collateral surplus returns to borrowers.
10. **Gas Compensation:** The gas compensation mechanism developed from a simple ETH-based system to an approach combining WETH and collateral.
11. **Emergency Measures:** The protocol replaced the Recovery Mode with a collateral branch shutdown mechanism. This new system triggers upon either price collapse or oracle failure, providing branch-specific protection.
12. **Stability Pool:** The Stability Pool implementation received more flexibility in V2. Users can now choose to claim or stash LST gains and handle BOLD yield gains separately, providing more options for managing returns and positions.
13. **Oracle Implementation:** The oracle system was modified from primarily relying on Chainlink's ETH/USD oracle to supporting multiple oracle configurations for different collateral types. This includes composite oracle setups for LSTs.

5. Design and Risk Mitigation Mechanisms

The Liquity V2 protocol includes several features designed to address potential risks and enhance the system's resilience. This section highlights the security-related mechanisms implemented in the protocol, focusing on collateral management, liquidations, yield distribution, position tracking, and emergency measures. The analysis reflects the functionality and intended impact of these mechanisms as observed during the audit.

Additional details about the Liquity V2 architecture and operations can be found in this report's Appendix, which provides an in-depth protocol overview. Further technical information is also available in Liquity's official [README](#), offering a detailed explanation of its design and implementation.

Collateral Support

The protocol's multi-collateral framework, supporting `WETH` and Liquid Staking Tokens (LSTs) like `rETH` and `wstETH`, works with independent branches with tailored risk collateral ratios, to manage collateral-specific risks effectively.

Interest Rate Mechanism

The protocol's interest rate mechanism enables borrowers to set individual rates, with interest accruing continuously and compounding upon trove modifications. Redemption prioritization is based on interest rates, targeting positions with the lowest rates first. This structure aligns redemption processes with borrower-defined rates, while batch interest calculations optimize gas usage for grouped troves.

Yield Distribution

Yield generated by interest payments is allocated between Stability Pool depositors and liquidity providers using a fixed ratio. The protocol maintains a global accounting mechanism combined with individual depositor snapshots to

facilitate proportional reward distribution. The sum-product algorithm supports the fair allocation of accumulated yield across participants.

Stability Pool Operations

The Stability Pool manages BOLD deposits contributed by users to support the system during liquidations. Depositors are rewarded with liquidated collateral and interest yield, tracked through snapshot-based accounting mechanisms. The pool allows users to withdraw collateral gains directly or reinvest them into the system, providing flexibility while supporting protocol stability.

Liquidation and Redemption Processes

The protocol incorporates a structured system to address undercollateralized positions and handle user redemptions. Liquidations apply variable penalties, with Stability Pool liquidations generally incurring lower penalties than redistributions to active troves. Redemption prioritization follows interest rate ordering, processing positions with the lowest rates first. The system updates variables at the system-wide level after each operation, supporting consistent state tracking across the protocol.

Unredeemable Troves

A new concept of unredeemable troves was introduced in V2. These troves remain open when debt falls below the minimum threshold, providing borrowers with more flexibility to restore positions. This change reduces forced closures while maintaining system integrity through minimum debt requirements.

Management System

The delegation framework in V2 enables both individual and batch management of troves. This allows more sophisticated control of interest rates and collateral adjustments, optimizing position management and supporting potential management services.

Gas Compensation

The protocol incorporates a mechanism to manage transaction costs for liquidators. During trove creation, a fixed amount of WETH is allocated to the Gas Pool. These funds are distributed to liquidators during successful operations, with unspent amounts returned to trove owners upon normal closure. The system balances the allocation of gas compensation with the preservation of user collateral.

Zappers

Zapper contracts operate as intermediaries for managing Liquity positions, facilitating token swaps, flash loans, and multi-step operations. These contracts incorporate sender authentication, and access control mechanisms to support efficient position management. The architecture consolidates multiple operations into single transactions, reducing operational complexity and costs.

Oracle Implementation

The V2 expanded its oracle framework beyond Chainlink's ETH/USD oracle to include multiple configurations tailored to the supported collateral types, such as composite oracles for LSTs. This diversification improves robustness in price data and supports the protocol's multi-collateral architecture.

Emergency Measures

Emergency mechanisms address critical scenarios, such as branch collateralization falling below thresholds or oracle failures. Branch shutdown processes include freezing borrower operations, halting interest accrual, and enabling urgent redemptions. In the case of oracle disruptions, the protocol employs a last-good-price mechanism to maintain functionality, allowing redemptions to continue even under adverse conditions.

6. Decentralization Analysis

Despite its decentralized design, Liquity V2 presents a few centralization risks. The protocol's reliance on Chainlink oracles for collateral valuation introduces a dependency on an external service. This reliance is critical for the system's operation and risk assessment.

The incorporation of Liquid Staking Tokens as collateral adds potential centralization risks associated with the governance and operation of these tokens' underlying protocols. The stability and decentralization of Liquity V2 partly depend on the characteristics of these external systems.

The batch management feature, allowing designated actors to control interest rates for multiple troves, presents a potential concentration of influence. If significant portions of the system's debt come under the management of a few large batches, it could lead to centralization of decision-making power within the protocol.

7. Code quality and Testing

The documentation for Liquity V2 is comprehensive, covering the core architecture, key mechanisms, and mathematical foundations of the protocol. It provides detailed explanations of changes from V1, known issues, and mitigation strategies. The documentation includes descriptions of the smart contract structure and interactions between different components of the system.

However, many smart contract functions lack NatSpec documentation. Adding NatSpec to function definitions would significantly improve code readability and maintainability. This addition would provide clear, standardized descriptions of function purposes, parameters, return values, and any important notes or warnings directly within the smart contract code.

While thorough, the documentation could benefit from more visual aids to enhance understanding of complex interactions within the system. Additionally, including more practical examples of user interactions and potential scenarios or expected flows would further clarify the protocol's operation for both users and developers.

Coinspect identified that the testing suite is comprehensive, thorough and easy to understand. This eased the process of testing different scenarios and confirming potential issues.

8. Fixes and Changes

On **November 4th 2024**, **Liquity** engaged **Coinspect** to perform a Smart Contract Audit of several fixes and modifications included to Bold. The objective of the project was to evaluate the security of the smart contracts' implementation.

- A new exchange used by Zappers that routes swaps between Curve and UniswapV3
- A fix for feedback errors in the Stability Pool
- The addition of a check for zero-debt in batched troves

8.1 Zappers Hybrid Exchange

The scope was set to be the `/contracts/src/Zappers/` directory the repository at <https://github.com/liquity/bold/> at commit `ec2272fa7887f323293c154c0ae42008577d8e6e`.

Coinspect reviewed the new implementation for Hybrid Exchanges used by Zappers. These exchanges allow complex and more efficient swap routes leveraging from Curve and Uniswap V3. Swaps of stablecoins (BOLD ↔ USDC) are made in Curve whereas those involving non-stable tokens (collateral) use UniswapV3. Its architecture allows performing more efficient swaps by reducing the global inefficiencies of using volatile pools for stablecoins.

Coinspect observed that intermediate swaps specify an infinite slippage, checking only the slippage on the last swap. Although the behavior protects end users, it can also increase the amount of reversals if the intermediate swap returns less tokens than those required to make the last swap. Users are still protected by the final slippage check enforced at the end of the swap route. However, this approach may lead to unnecessary gas consumption when intermediate swaps execute successfully but the transaction ultimately reverts at the final slippage check.

8.2 Stability Pool Fix

The scope was set to be the pull request [#552](#) the repository at <https://github.com/liquity/bold/> at commit `87198e94137d0b667d5a63bbb4a3ff3d7a3c8011`.

The introduced fix changes how feedback errors are accumulated and accounted. The issue was that errors were calculated using the absolute representation using the previous Bold deposit value. After multiple liquidations the Bold's balance decreases, amplifying the errors' value.

This new implementation changes an absolute error calculation for a relative one, considering the total Bold deposits step by step. This smooths the error feedback calculation, reducing the amplification effect of using older and higher total deposit values.

8.3 Check for Zero Debt in Batched troves

The scope was set to be the pull request [#560](https://github.com/liquity/bold/pull/560) the repository at <https://github.com/liquity/bold/> at commit [71026e626001e89174140081fb12272c0116c9b1](https://github.com/liquity/bold/commit/71026e626001e89174140081fb12272c0116c9b1).

This fix adds an invariant check to ensure that no dust is left after reducing the debt of a trove inside a batch. A variable tracking the new troves' debt is added to ensure that debt adjustments don't leave dust when the ending trove debt is zero after an operation.

8.4 Minor changes

After the main review, Liquity added the following minor changes to the protocol:

- Simplified zapper's interface of `closeTroveFromCollateral()` at <https://github.com/liquity/bold/pull/578>
- Added a check to prevent urgent redemptions with zero amount at <https://github.com/liquity/bold/pull/586/>
- Updated protocol's constants and tests at <https://github.com/liquity/bold/pull/562>
- Added a library to convert UniswapV3 prices at <https://github.com/liquity/bold/pull/571>
- Removed an unused dependency on a Zappers at <https://github.com/liquity/bold/pull/595/>
- Restricted delegation logic allowing other receivers when the caller is the manager at <https://github.com/liquity/bold/pull/610/files>
- Changed NFT font at <https://github.com/liquity/bold/pull/601>
- Changed route for `MockInterestRouter`, used for testing at <https://github.com/liquity/bold/commit/10d87177bb9bee4a68a12e45e55a3636t>
- Added `safeTransfer` for transfers made through Zappers at <https://github.com/liquity/bold/pull/634>
- Edited unused interface at price feeds at <https://github.com/liquity/bold/pull/633> and <https://github.com/liquity/bold/pull/640>



- Made two functions of price feeds abstract at <https://github.com/liquity/bold/commit/69d715c986789fe35aca7af06b5671d6e>
- Caches flashloan receiver address to comply with checks-effects-interactions if the code is reused at <https://github.com/liquity/bold/pull/666/>

Coinspect identified no issues related to these minor changes.

9. Detailed Findings

BOLD-01

Leveraged operations will not work as approvals are spontaneously reset

Status Solved	Risk Medium
	
Resolution Fixed	Impact Low Likelihood High

Location

```
bold/contracts/src/Zappers/LeverageWETHZapper.sol:80
bold/contracts/src/Zappers/LeverageWETHZapper.sol:157
bold/contracts/src/Zappers/LeverageLSTZapper.sol:85
bold/contracts/src/Zappers/LeverageLSTZapper.sol:160
```

Description

The ERC20 allowance is reset when making non leveraged operations through a Zapper. This prevents subsequent leveraged operations from functioning

correctly, as Zappers can't properly interact with BorrowOperations once a flashloan is received.

Upon deployment, leveraged Zappers grant infinite allowance for the collateral token to the BorrowOperations contract:

```
// Approve WETH to BorrowerOperations
WETH.approve(address(_borrowerOperations), type(uint256).max);
// Approve coll to BorrowerOperations
collToken.approve(address(_borrowerOperations),
type(uint256).max);
```

This allowance is required when a user wants to perform operations that send collateral to Liquity (e.g. opening leveraged troves, increasing a trove's leverage):

```
uint256 totalCollAmount = _params.collAmount +
_effectiveFlashLoanAmount;
// We compute boldAmount off-chain for efficiency

// Open trove
if (_params.batchManager == address(0)) {
    vars.troveId = vars.borrowerOperations.openTrove(
        _params.owner,
        _params.ownerIndex,
        totalCollAmount,
        _params.boldAmount,
        _params.upperHint,
        _params.lowerHint,
        _params.annualInterestRate,
        _params.maxUpfrontFee,
        // Add this contract as add/receive manager to be able
to fully adjust trove,
        // while keeping the same management functionality
        address(this), // add manager
        address(this), // remove manager
        address(this) // receiver for remove manager
    );
}
....
```

```
// Adjust trove
// With the received coll from flash loan, we increase both the
trove coll and debt
borrowerOperations.adjustTrove(
    _params.troveId,
    _effectiveFlashLoanAmount, // flash loan amount minus fee
    true, // _isCollIncrease
    _params.boldAmount,
    true, // _isDebtIncrease
    _params.maxUpfrontFee
);
```

However, leveraged Zappers inherit from base Zappers that allow users handling non-leveraged troves. This functionality grants borrows operations allowance for the collateral amount for that transaction, wiping the infinite allowance granted upon deployment:

```
function openTroveWithRawETH(OpenTroveParams calldata _params)
external payable returns (uint256)
{...}
// Pull and approve coll
vars.collToken.safeTransferFrom(msg.sender, address(this),
_params.collAmount);
vars.collToken.approve(address(vars.borrowerOperations),
_params.collAmount);
{...}
```

As a consequence, leveraged operations will not have the necessary allowance to interact with `BorrowOperations`, reverting the transaction.

Recommendation

Handle allowances to borrow operations inside each flashloan callback, before handling troves on Liquity.

Status

Fixed on commit `7f2058a02cffe6f9c8a21b605488a4faaa31b0c`.

Approvals are now only handled upon deployment and are not refreshed between intermediate steps.

Proof of Concept

The following scenario shows how opening a non-leveraged trove through a Zapper resets its collateral token allowance to borrow operations. As a result, trying to open a leveraged trove reverts since the transfer amount exceeds allowance.

```
[FAIL. Reason: revert: ERC20: transfer amount exceeds allowance]
testCoinspectBreaksLeveragedZapper()
```

```
function testCoinspectBreaksLeveragedZapper() external {
    ILeverageZapper _leverageZapper = leverageZapperCurveArray[1];
    uint256 _branch = 1; // index 1 means it is an LST
```

```

    address _batchManager = address(0);

    // A regular trove is opened (e.g. through openTroveWithRawETH)
    openTrove(_leverageZapper, B, 2.5 ether, 2000e18, true); //
    this call resets the collToken approval to collAmount

    TestVars memory vars;
    vars.collAmount = 10 ether;
    vars.newLeverageRatio = 2e18;
    vars.resultingCollateralRatio =
    _leverageZapper.leverageRatioToCollateralRatio(vars.newLeverageRatio);

    _setInitialBalances(_leverageZapper, _branch, vars);



    bool lst = _branch > 0;

    // Opening a trove reverts since there is not enough allowance
    from the Zapper to BorrowOps
    vars.troveId = openLeveragedTrove(
        _leverageZapper,
        vars.collAmount,
        vars.newLeverageRatio,
        contractsArray[_branch].priceFeed,
        lst,
        _batchManager
    );
}

```

BOLD-02

Attackers can claim minted yield from the Stability Pool without leaving their funds deposited

Status Solved	Risk None
	
Resolution Acknowledged	Impact Recommendation
	Likelihood -
Location <code>bold/contracts/src/StabilityPool.sol</code>	

Description

Opening and closing troves when the Stability Pool is bootstrapping (e.g. after an epoch change) allow adversaries to extract most of the minted interest by atomically depositing and withdrawing.

When a trove is opened or closed, aggregated interest is minted to the Stability Pool:

```
function _mintAggInterest(IBoldToken _boldToken, uint256
_upfrontFee) internal returns (uint256 mintedAmount) {
    mintedAmount = calcPendingAggInterest() + _upfrontFee;

    // Mint part of the BOLD interest to the SP and part to the
    router for LPs.
```

```

    if (mintedAmount > 0) {
        uint256 spYield = SP_YIELD_SPLIT * mintedAmount /
DECIMAL_PRECISION;
        uint256 remainderToLPs = mintedAmount - spYield;

        _boldToken.mint(address(interestRouter), remainderToLPs);

        if (spYield > 0) {
            _boldToken.mint(address(stabilityPool), spYield);
            stabilityPool.triggerBoldRewards(spYield);
        }
    }

    lastAggUpdateTime = block.timestamp;
}

```

When the Stability Pool is empty, the minted interest is accumulated and not accounted as there is an early return to the `_updateYieldRewardsSum` which skips updating `epochToScaleToB`:

```

function _updateYieldRewardsSum(uint256 _newYield) internal {
    uint256 accumulatedYieldGains = yieldGainsPending + _newYield;
    if (accumulatedYieldGains == 0) return;

    // When total deposits is very small, B is not updated. In this
    // case, the BOLD issued is hold
    // until the total deposits reach 1 BOLD (remains in the
    // balance of the SP).
    uint256 totalBoldDepositsCached = totalBoldDeposits; // cached
    to save an SLOAD
    if (totalBoldDepositsCached < DECIMAL_PRECISION) {
        yieldGainsPending = accumulatedYieldGains;
        return;
    }

    yieldGainsOwed += accumulatedYieldGains;
    yieldGainsPending = 0;

    /*
     * Calculate the BOLD-per-unit staked. Division uses a
     "feedback" error correction, to keep the
     * cumulative error low in the running total B:
     *
     * 1) Form a numerator which compensates for the floor division
     error that occurred the last time this
     * function was called.
     * 2) Calculate "per-unit-staked" ratio.
     * 3) Multiply the ratio back by its denominator, to reveal the
     current floor division error.
     * 4) Store this error for use in the next correction when this
     function is called.
     * 5) Note: static analysis tools complain about this "division
     before multiplication", however, it is intended.
     */
    uint256 yieldNumerator = accumulatedYieldGains *
DECIMAL_PRECISION + lastYieldError;

    uint256 yieldPerUnitStaked = yieldNumerator /

```

```

totalBoldDepositsCached;
    lastYieldError = yieldNumerator - yieldPerUnitStaked *
totalBoldDepositsCached;

    uint256 marginalYieldGain = yieldPerUnitStaked * (P - 1);
    epochToScaleToB[currentEpoch][currentScale] =
epochToScaleToB[currentEpoch][currentScale] + marginalYieldGain;

    emit B_Updated(epochToScaleToB[currentEpoch][currentScale],
currentEpoch, currentScale);
}

```

Then, all rewards are assigned to the first depositor when they provide more than `DECIMAL_PRECISION`.

In conclusion, attackers can track trove openings and closures when the Stability Pool is bootstrapping and maliciously extract minted interest leaving the Stability Pool empty.

Consider the following scenario:

- The liquidity across all opened troves is high, meaning that there is enough Bold in circulation (required to make deposits into the Stability Pool).
- At some point, the Stability Pool is emptied (e.g. due to massive liquidations because of a market crash).
- During this state, opening and closing troves mint interest to the empty Stability Pool.
- Attackers abuse this condition on every interest-minting transaction. They deposit and withdraw some Bold into the Stability Pool to maliciously extract almost all the recently minted interest.
- As long as the pool remains empty, all proceeds from interest-minting are vulnerable to this malicious extraction. Attackers are not required to leave their funds locked into the Stability Pool to extract this value, allowing them to quickly swap their profits elsewhere for a different token, progressively dumping Bold.

Recommendation

Document this scenario and how it aligns with the Stability Pool incentives.

Status

Acknowledged.

The Liquity Team stated that they don't expect the Stability Pool to be empty if there is significant borrowing activity. In such a case, the issue would be easy to counteract if someone deposits just 1 BOLD.

Proof of concept

The following test shows a scenario where an attacker opens a trove to get some Bold tokens. Then, waits until another user opens a trove, making a deposit just before. After the trove is opened, withdraws all their stakes from the Stability Pool extracting the fee and leaving the pool empty.

Output

```
Initial State...
  1. BOLD Balance in SP: 0

Opening initial Trove to B (generate Bold balance)...
  BoldYield minted: 69041095890410958904
  2. After opening TroveB: BOLD Balance in SP: 69041095890410958904

Before B Deposit on SP
  BOLD Balance in SP: 69041095890410958904
  getDepositorYieldGain(B): 0
  getYieldGainsPending(): 0
  getYieldGainsOwed(): 0
  Deposit amount: 1000000000000000000

Deposit successful...

After B Deposit on SP
  BOLD Balance in SP: 70041095890410958904
  getDepositorYieldGain(B): 69041095890410958834
  getYieldGainsPending(): 1000000000000000000
  getYieldGainsOwed(): 1000000000000000000

Opening victim Trove to C...
  BoldYield minted: 13808219178082191780
  2. After opening TroveB: BOLD Balance in SP: 83849315068493150684

Withdrawing position claiming collateral and yield...
  BOLD Balance in SP: 83849315068493150684
  getDepositorYieldGain(B): 82849315068493150601
  getYieldGainsPending(): 1000000000000000000
  getYieldGainsOwed(): 1000000000000000000

After Withdrawal
  BOLD Balance in SP: 83
  getDepositorYieldGain(B): 0
  getYieldGainsPending(): 0
  getYieldGainsOwed(): 0

Bold Profit of B: 82849315068493150601
```

Test

```

function test_Coinspect_StealsMintedYield() public {
    console.log("\nInitial State...");
    console.log("1. BOLD Balance in SP: %s",
boldToken.balanceOf(address(stabilityPool)));

    // Open Troves
    uint256 troveDebtRequest_B = 100_000e18; // Simulate that many
troves are opened
    uint256 interestRate = 5e16; // 5%
    uint256 price = 2000e18;
    priceFeed.setPrice(price);

    console.log("\nOpening initial Trove to B (generate Bold
balance)...");
    uint256 troveB = openTroveNoHints100pct(B, 150 ether,
troveDebtRequest_B, interestRate);
    console.log("2. After opening TroveB: BOLD Balance in SP: %s",
boldToken.balanceOf(address(stabilityPool)));

    // B deposits to SP
    console.log("\nBefore B Deposit on SP");
    console.log("BOLD Balance in SP: %s",
boldToken.balanceOf(address(stabilityPool)));
    console.log("getDepositorYieldGain(B): %s",
stabilityPool.getDepositorYieldGain(B));
    console.log("getYieldGainsPending(): %s",
stabilityPool.getTotalBoldDeposits());
    console.log("getYieldGainsOwed(): %s",
stabilityPool.getTotalBoldDeposits());

    uint256 depositAmount = 1e18;
    uint256 initialBoldBalance_B = boldToken.balanceOf(B);
    console.log("Deposit amount: %s", depositAmount);
    makeSPDepositAndClaim(B, depositAmount);
    console.log("\nDeposit successful...");

    console.log("\nAfter B Deposit on SP");
    console.log("BOLD Balance in SP: %s",
boldToken.balanceOf(address(stabilityPool)));
    console.log("getDepositorYieldGain(B): %s",
stabilityPool.getDepositorYieldGain(B));
    console.log("getYieldGainsPending(): %s",
stabilityPool.getTotalBoldDeposits());
    console.log("getYieldGainsOwed(): %s",
stabilityPool.getTotalBoldDeposits());

    console.log("\nOpening victim Trove to C...");
    uint256 troveDebtRequest_C = 20_000e18;
    uint256 troveC = openTroveNoHints100pct(C, 15 ether,
troveDebtRequest_C, interestRate);
    console.log("2. After opening TroveB: BOLD Balance in SP: %s",
boldToken.balanceOf(address(stabilityPool)));

    console.log("\nWithdrawing position claiming collateral and
yield...");
    console.log("BOLD Balance in SP: %s",
boldToken.balanceOf(address(stabilityPool)));
    console.log("getDepositorYieldGain(B): %s",
stabilityPool.getDepositorYieldGain(B));
    console.log("getYieldGainsPending(): %s",

```

```

stabilityPool.getTotalBoldDeposits());
    console.log("getYieldGainsOwed(): %s",
stabilityPool.getTotalBoldDeposits());

    vm.prank(B);
    stabilityPool.withdrawFromSP(type(uint256).max, true); //
withdraw all



    console.log("\nAfter Withdrawal");
    console.log("BOLD Balance in SP: %s",
boldToken.balanceOf(address(stabilityPool)));
    console.log("getDepositorYieldGain(B): %s",
stabilityPool.getDepositorYieldGain(B));
    console.log("getYieldGainsPending(): %s",
stabilityPool.getTotalBoldDeposits());
    console.log("getYieldGainsOwed(): %s",
stabilityPool.getTotalBoldDeposits());

    console.log("\nBold Profit of B: %s", boldToken.balanceOf(B) -
initialBoldBalance_B);
}

```

BOLD-03

Adversaries can manipulate batch's debt/shares ratio by deflating the debt

Status Solved	Risk None
	
Resolution Acknowledged	Impact Recommendation
	Likelihood -
Location bold/contracts/src/TroveManager.sol:1807	

Description

Trove owners inside a batch can manipulate the value of `batchDebtShares` by repaying small amounts of debt. As a consequence, the batch's debt is reduced leaving the amount of shares constant.

Shares of a batch are updated inside `_updateBatchShares` when a trove of the batch is touched. Upon debt decrease, shares are updated as follows:

```
// Subtract debt
batchDebtSharesDelta = currentBatchDebtShares * debtDecrease /
_batchDebt;

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

```
batches[_batchAddress].totalDebtShares = currentBatchDebtShares -  
batchDebtSharesDelta;
```

Making small debt decreases to a trove leads to `batchDebtSharesDelta` being zero for non-zero debt decreases (rounds down). Then, when a new trove is opened in the same batch more debt shares are assigned to that trove. This happened because the `batchDebt` decreased but the batch shares remained constant after the manipulation:

```
// To avoid rebasing issues, let's make sure the ratio debt /  
shares is not too high  
_requireBelowMaxSharesRatio(currentBatchDebtShares, _batchDebt,  
_checkBatchSharesRatio);  
  
batchDebtSharesDelta = currentBatchDebtShares * debtIncrease /  
_batchDebt;  
}  
  
Trove[_troveId].batchDebtShares += batchDebtSharesDelta;  
batches[_batchAddress].debt = _batchDebt + debtIncrease;  
batches[_batchAddress].totalDebtShares = currentBatchDebtShares +  
batchDebtSharesDelta;
```

This issue is considered to have no impact, since this path reduces the debt of a batch. It aims to flag a scenario where an attacker can still manipulate the tracking of variables, altering the representation of debt per share of a batch.

Recommendation

Set a minimum value for debt decreases, ensuring that this minimum leaves no dust.

Status

Acknowledged at <https://github.com/liquity/bold/issues/553#issuecomment-2519934916>.

Proof of Concept

The following scenario shows how an attacker is able to deflate the debt without altering the batch's shares.

```
batchDebtShares: 2036012054794520547945
debt: 2036012831469590917621
allBatchDebtSharesBefore: 2036012054794520547945
```

```
repay to force rounding
batchDebtShares: 2036012054794520547945
debt: 2036012831469590917421
allBatchDebtSharesAfter: 2036012054794520547945
```

```
deltas
batchDebtShares: 0
debt: 200
allBatchDebtSharesBefore: 0
```

```
function testCoinspectDeflateDebtLeavingSharesConstant() public {
    // == Generate Bold Balance on A == //
    priceFeed.setPrice(2000e18);
    openTroveNoHints100pct(C, 100 ether, 100e21,
MAX_ANNUAL_INTEREST_RATE);
    vm.startPrank(C);
    boldToken.transfer(A, boldToken.balanceOf(C));
    vm.stopPrank();

    uint256 BTroveId = openTroveAndJoinBatchManager(B, 100 ether,
MIN_DEBT - 2.3 ether, B, MAX_ANNUAL_INTEREST_RATE);

    if (WITH_INTEREST) {
        vm.warp(block.timestamp + 12);
    }
    _addOneDebtAndEnsureItDoesntMintShares(BTroveId, B);

    (uint256 debtBefore,,,,,, uint256 allBatchDebtSharesBefore) =
troveManager.getBatch(B);
    uint256 sharesBeforeRedeem = _getBatchDebtShares(BTroveId);
    console.log("batchDebtShares: %s", sharesBeforeRedeem);
    console.log("debt: %s", debtBefore);
    console.log("allBatchDebtSharesBefore: %s",
allBatchDebtSharesBefore);

    vm.startPrank(A);
    uint256 debtAfter;
    uint256 allBatchDebtSharesAfter;
    uint256 sharesAfterRedeem;

    console.log("\n repay to force rounding");
    uint256 x;
    while (x++ < 200) {
        borrowerOperations.repayBold(BTroveId, 1);
    }



    (debtAfter,,,,,, allBatchDebtSharesAfter) =
troveManager.getBatch(B);
    sharesAfterRedeem = _getBatchDebtShares(BTroveId);
    console.log("batchDebtShares: %s", sharesAfterRedeem);
    console.log("debt: %s", debtAfter);
    console.log("allBatchDebtSharesAfter: %s",
allBatchDebtSharesAfter);
```

```
vm.stopPrank();

console.log("\ndeltas");
console.log("batchDebtShares: %s", sharesBeforeRedeem -
sharesAfterRedeem);
console.log("debt: %s", debtBefore - debtAfter);
console.log("allBatchDebtSharesBefore: %s",
allBatchDebtSharesBefore - allBatchDebtSharesAfter);
}
```

BOLD-04

Unused code

Status Solved	Risk None
	
Resolution Fixed	Impact Recommendation
	Likelihood -

Location

bold/contracts/src/BorrowerOperations.sol:1263
bold/contracts/src/CollateralRegistry.sol

Description

The code includes an unused modifier in BorrowerOperations:

```
function _requireIsShutDown() internal view {  
    if (!hasBeenShutDown) {  
        revert NotShutDown();  
    }  
}
```

Also, the collateral registry initializes 10 immutable variables for tokens and trove managers, but the documentation specifies that the amount of collaterals (hence branches) to deploy is less than this value.

```
IERC20Metadata internal immutable token0;  
IERC20Metadata internal immutable token1;
```



```
IERC20Metadata internal immutable token2;  
IERC20Metadata internal immutable token3;  
IERC20Metadata internal immutable token4;  
IERC20Metadata internal immutable token5;  
IERC20Metadata internal immutable token6;  
IERC20Metadata internal immutable token7;  
IERC20Metadata internal immutable token8;  
IERC20Metadata internal immutable token9;  
  
ITroveManager internal immutable troveManager0;  
ITroveManager internal immutable troveManager1;  
ITroveManager internal immutable troveManager2;  
ITroveManager internal immutable troveManager3;  
ITroveManager internal immutable troveManager4;  
ITroveManager internal immutable troveManager5;  
ITroveManager internal immutable troveManager6;  
ITroveManager internal immutable troveManager7;  
ITroveManager internal immutable troveManager8;  
ITroveManager internal immutable troveManager9;
```

Recommendation



Remove unused code.

Status

Fixed on commit [4aa8b3473529eb1b181625d23e7bf403777111b8](#).

BOLD-05

Single oracle failure stops redemptions across all branches triggering a depeg event

Status Solved	Risk Medium
	
Resolution Fixed	Impact High Likelihood Low

Location

`bold/contracts/src/PriceFeeds/CompositePriceFeed.sol:31`

Description

If an oracle fails to retrieve the canonical price of an LST token, redemptions across all branches will be halted, leading to a depeg event. Additionally, all user deposits in that branch will be locked and irrecoverable.

Price feeds for liquid staking tokens compare canonical prices with those reported by Chainlink, using the minimum price to prevent upward manipulation:

```
function _fetchPrice() internal override returns (uint256, bool) {
    (uint256 ethUsdPrice, bool ethUsdOracleDown) =
_getOracleAnswer(ethUsdOracle);
    (uint256 lstEthPrice, bool lstEthOracleDown) =
```

```

_getOracleAnswer(1stEthOracle);

    // If one of Chainlink's responses was invalid in this
    transaction, disable this PriceFeed and
    // return the last good LST-USD price calculated
    if (ethUsdOracleDown) return
(_disableFeedAndShutDown(address(ethUsdOracle.aggregator)), true);
    if (1stEthOracleDown) return
(_disableFeedAndShutDown(address(1stEthOracle.aggregator)), true);

    // Calculate the market LST-USD price: USD_per_LST =
    USD_per_ETH * ETH_per_LST
    uint256 1stUsdMarketPrice = ethUsdPrice * 1stEthPrice / 1e18;

    // Get the ETH_per_LST canonical rate directly from the LST
    contract
    // TODO: Should we also shutdown if the call to the canonical
    rate reverts, or returns 0?
    uint256 1stEthRate = _getCanonicalRate();

    // Calculate the canonical LST-USD price: USD_per_LST =
    USD_per_ETH * ETH_per_LST
    uint256 1stUsdCanonicalPrice = ethUsdPrice * 1stEthRate / 1e18;

    // Take the minimum of (market, canonical) in order to mitigate
    against upward market price manipulation
    uint256 1stUsdPrice = LiquidityMath._min(1stUsdMarketPrice,
    1stUsdCanonicalPrice);

    lastGoodPrice = 1stUsdPrice;

    return (1stUsdPrice, false);
}

```

The canonical price for each collateral token is calculated by making an external call to the smart contract of each LST. For example:

```

function _getCanonicalRate() internal view override returns
(uint256) {
    // RETHToken returns exchange rate with 18 digit decimal
    precision
    return IREHToken(rateProviderAddress).getExchangeRate();
}

```

```

function _getCanonicalRate() internal view override returns
(uint256) {
    // OsTokenVaultController returns rate with 18 digit decimal
    precision
    return
    IOsTokenVaultController(rateProviderAddress).convertToAssets(1e18);
}

```

```

function _getCanonicalRate() internal view override returns
(uint256) {
    // StaderOracle returns ETH balance and ETHX supply each with

```

18 digit decimal precision

```
(
    , // uint256 reportingBlockNumber
    uint256 ethBalance,
    uint256 ethXSupply
) = IStaderOracle(rateProviderAddress).exchangeRate();

return ethBalance * 1e18 / ethXSupply;
}
```

If any of the LST contracts revert, it bubbles up and reverts the call to `fetchPrice` without leading to a branch shutdown. This scenario has two catastrophic consequences:

1. **Locked User Deposits:** User deposits on that collateral's branch will remain locked inside the protocol.
2. **Reverted Redemptions:** Redemptions will be halted, even if the protocol has all other branches operative.

Both consequences are related to the fact that it is impossible to calculate collateral ratios since this operation calls `fetchPrice`.

The second consequence arises because the `CollateralRegistry` smart contract loops over every branch calculating the unbacked debt:

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

Because of this, when a single oracle reverts after being queried the canonical price, the whole `redeemCollateral` call reverts. This prevents anyone from redeeming BOLD, causing the stablecoin to depeg.

Since only a reversal from a single branch is needed, if the TVL of any of the LSTs is lower than Liquity's, attackers have incentive to manipulate and/or drain any of the underlying LST protocol in order to profit from Bold's depeg. For example, canonical price for OSETH reverts when there are no assets:

```
function _convertToShares(
    uint256 assets,
    uint256 totalShares_,
    uint256 totalAssets_,
    Math.Rounding rounding
) internal pure returns (uint256 shares) {
    // Will revert if assets > 0, totalShares > 0 and totalAssets = 0.
```

```
// That corresponds to a case where any asset would represent an
infinite amount of shares.
return
  (assets == 0 || totalShares_ == 0)
  ? assets
  : Math.mulDiv(assets, totalShares_, totalAssets_, rounding);
}
```

Additionally, canonical feeds for `wstETH` (Lido) and `ETHX` (Stader) are upgradeable contracts that could change in the future.

Coinspect considers a revert from LST contracts is unlikely. However, there are scenarios where this could happen, for example, the V1 of the protocol required a custom fix to its fallback price oracle to [prevent an exploit](#) during ETH2 migration. Because of that, the likelihood of this issue has been determined to be low.

The impact of this issue is high since it will trigger a depeg event, preventing redemptions and locking all users' funds inside the broken branch. Also, because it impacts redemptions for the whole protocol, oracles are considered single points of failure for the system.

Recommendation

Use try-catch logic when making external calls to each LST contract. In case of revert, evaluate the best alternative between triggering a branch shutdown or keeping operating only with the price retrieved by Chainlink.

Status

Fixed on commit `bf2a5c167784a24ad170c6939cfb8a56dff49704`.


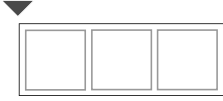
Liquity modified the price feeds by adding try-catch logic when retrieving canonical exchange rates. This feature triggers a shutdown when canonical rate feeds are not responding, preventing reversals during the redemption process as mentioned on this issue.

Also, Liquity added safeguards to prevent triggering a shutdown in case of insufficient gas forwarding when making the external call.

Lastly, Liquity removed `osETH` and `ETHX` as collateral types at the fix commit.

BOLD-06

Wrong implementation of an LST oracle forces protocol re-deployment

Status Solved	Risk None
	
Resolution Fixed	Impact Recommendation
	Likelihood -

Location

`bold/contracts/src/PriceFeeds/CompositePriceFeed.sol:75`

Description

Each composite oracle is meant to override the `_getCanonicalRate()` function to work as expected. However, if an oracle is deployed directly as a `CompositePriceFeed` instead of using it as a parent with an overridden version of `_getCanonicalRate()`, the branch depending on that oracle will not work.

```
// Returns the ETH_per_LST as from the LST smart contract.  
Implementation depends on the specific LST.  
function _getCanonicalRate() internal view virtual returns  
(uint256) {}
```

When this function is not implemented, `_fetchPrice` returns zero and the underlying branch will not be able to calculate collateral ratios.

Also the `CollateralRegistry` sets each branch upon deployment:

```
ITroveManager internal immutable troveManager0;  
ITroveManager internal immutable troveManager1;  
ITroveManager internal immutable troveManager2;  
ITroveManager internal immutable troveManager3;  
ITroveManager internal immutable troveManager4;  
ITroveManager internal immutable troveManager5;  
ITroveManager internal immutable troveManager6;  
ITroveManager internal immutable troveManager7;  
ITroveManager internal immutable troveManager8;  
ITroveManager internal immutable troveManager9;
```

If this issue is detected after deploying the `CollateralRegistry`, the whole protocol will have to be re-deployed in order to re-include that branch. This happens because each branch sets the `CollateralRegistry` address upon deployment.

Recommendation

Make `CompositePriceFeed` an abstract contract leaving `_getCanonicalRate()` unimplemented.

Status

Partially fixed on `bf2a5c167784a24ad170c6939cfb8a56dff49704`. Fully fixed at `69d715c986789fe35aca7af06b5671d6e25972d0`.

The `CompositePriceFeed` is now an abstract contract. Also, price feeds that inherit from it are now enforced to override `_getCanonicalRate()` and `_fetchPricePrimary()` since these functions are abstract.

BOLD-07

Amount of parameters when opening troves facilitates deceiving users

Status

Solved

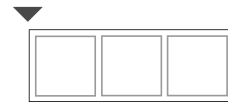


Resolution

Acknowledged

Risk

None



Impact

Recommendation

Likelihood

-

Location

`bold/contracts/src/BorrowerOperations.sol:196`

Description

When opening a trove, users must input 11 parameters. This amount could facilitate adversarial scenarios in case of a front-end attack. For example, openings require specifying the delegation accounts for the trove (managers and receiver):

```
function openTrove(  
    address _owner,  
    uint256 _ownerIndex,  
    uint256 _collAmount,  
    uint256 _boldAmount,  
    uint256 _upperHint,  
    uint256 _lowerHint,  
    uint256 _annualInterestRate,  
    uint256 _maxUpfrontFee,  
    address _addManager,
```



```
        address _removeManager,  
        address _receiver  
    )
```

Users using a hardware wallet to interact with the dApp could require enabling blind-signing in order to approve the trove opening transaction. Most hardware wallets are yet exploring alternatives to prevent this (e.g. [Ledger Clear Signing](#)), but it requires users to get new devices.

A compromised or malicious dApp might show users they are setting themselves or `address(0)` as the delegation accounts but instead, set a malicious account as the manager and receiver. Unsuspecting users might still believe they are signing for a normal trove opening operation since the wallet will show only a payload.

Moreover, users interacting with a browser wallet capable of parsing the calldata into its parameters, will have to scroll all the way down to display the delegation accounts information. This happens since these parameters are at the end of the function's interface.

Users opening troves delegating management to malicious addresses could be unaware of this unless they verify it at some point. Since delegations are not cleared after transferring (per design decision), attackers could accumulate a considerable amount of "owned" troves from unsuspecting users and then proceed to drain them all at once.

Tests indicate that the most probable flow for users opening a trove for themselves, involves the following set of parameters:

```
troveId = borrowerOperations.openTrove(  
    _account,  
    _index,  
    _coll,  
    _boldAmount,  
    0, // _upperHint  
    0, // _lowerHint  
    _annualInterestRate,  
    upfrontFee,  
    address(0),  
    address(0),  
    address(0)  
);
```

It can be seen that for a common use-case, trove openings specify 6 out of the 11 parameters. This leaves room for the before-mentioned attack by tuning the other 5 parameters.

Recommendation

Add a function to BorrowOperations's interface (e.g. `openTroveSimplifiedToSelf()`) with less parameters to reduce the attack surface when signing for trove openings.


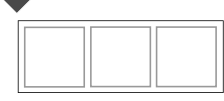
Status

Acknowledged.

The Liquity team stated that they accept the risks derived from this issue as they are close to maximum contract size.

BOLD-08

Redemption decay rate is zero before expected

Status Solved	Risk None
	
Resolution Deferred	Impact Recommendation
	Likelihood -

Location

```
bold/contracts/src/Dependencies/LiquidityMath.sol:51
```

Description

The calculation of the `decayRate` when computing the redemption rate specifies that it should become zero after almost 1000 years:

```
the decayed base rate will be 0 for 1000 years or > 1000 years
```

However, Coinspect identified that this rate is zero nearly after 35 days. This is because the value of `REDEMPTION_MINUTE_DECAY_FACTOR`.

This quicker decay implies that fee rates decrease more rapidly, reducing the collateral fee charged upon redemption. As a consequence, redemptions would become more frequent and BOLD price might experience considerable fluctuations.

Recommendation

Document this behavior and analyze the impact it could have in Bold price fluctuations.

Status

Deferred.

The Liquity team stated:



The comment justifies why capping at 1,000 years is safe for `_decPow`. It doesn't say that it's expected to decay in 1k years. So, as long as the rate is zero after 1k years, the function is safe. As the real value seems to be ~35 years, the function is therefore super safe.

Proof of Concept

```
function testDecayAfterSomeTime() external {
  uint256 _n = (34 days + 23 hours) / ONE_MINUTE;
  uint256 _base = REDEMPTION_MINUTE_DECAY_FACTOR;
  assertEq(LiquityMath._decPow(_base, _n), 0);
}
```

BOLD-09

Missing bold amount check in urgent redemptions

Status Solved	Risk None
	
Resolution Fixed	Impact Recommendation
	Likelihood -
Location <code>bold/contracts/src/TroveManager.sol:850</code>	

Description

The `CollateralRegistry` smart contract checks for calls redeeming zero bold amounts:

```
function redeemCollateral(uint256 _boldAmount, uint256
_maxIterationsPerCollateral, uint256 _maxFeePercentage)
    external
{
    _requireValidMaxFeePercentage(_maxFeePercentage);
    _requireAmountGreaterThanZero(_boldAmount);
    ...
}
```

However, this is not the case for `urgentRedemption()`, allowing to trigger debt redistributions without providing any BOLD.

Recommendation

Ensure that the **BOLD** to redeem is greater than zero.

Status

Fixed on commit `5c767f18625ccbc3b7979c46176551f1aba8869d`.

10. Appendix: Protocol Overview

10.1 Protocol Architecture and Yield Generation

Liquity V2 includes a multi-collateral lending system that supports various tokens through independent branches, each with specific risk parameters. The protocol implements a market-driven interest rate mechanism where borrowers set their own rates, and generates yield through interest payments that are distributed between stability providers and liquidity providers according to fixed ratios.

10.1.1 Collateral Support

The V2 implements a multi-collateral lending system where users can deposit `WETH` and Liquid Staking Tokens (LSTs) - `rETH` and `wstETH` - as collateral. Each collateral type operates in its independent *branch* with specific parameters and risk metrics. Each collateral branch maintains its own set of parameters including Minimum Collateral Ratio (MCR), Critical Collateral Ratio (CCR), and Shutdown Collateral Ratio (SCR).

The protocol tracks collateral positions through three dedicated pools. The `ActivePool` holds collateral for active troves, while the `DefaultPool` holds collateral from liquidated positions. Any excess collateral from liquidations is stored in the `CollSurplusPool`.

10.1.2 Interest Rate Mechanism

Borrowers set their own annual interest rate when opening a trove. The interest accrues continuously based on the chosen rate and compounds when the trove is modified (*touched*) through any operation. The chosen interest rate affects the trove's position in redemption ordering, where lower rates are redeemed first. This rate can be modified by the borrower or a delegated manager at any time.

The protocol calculates interest through a combination of mechanisms. Interest accumulates as simple interest between trove operations, while it tracks aggregate interest at a system level. For grouped troves, the protocol performs

batch interest calculations to optimize gas usage and maintain accounting of all positions.

10.1.3 Yield Distribution

Interest generated from troves creates system revenue in **BOLD** tokens. The protocol distributes this yield through a fixed split, with 72% directed to Stability Pool depositors and 28% allocated to LP incentives (note these values were updated to 75% and 25% after the audit). This distribution occurs when interest is minted from aggregate system debt and on most system operations that touch troves, with allocation happening proportionally to depositor stake in the Stability Pool.

The yield system maintains accounting through several mechanisms. The protocol tracks yield at a global level while maintaining individual depositor snapshots. Accumulated yield calculations utilize a sum-product algorithm to make the distribution of rewards. This creates a system where interest payments from borrowers flow to stability providers and liquidity providers through a distribution mechanism.

10.2 Asset and Position Management

Liquity V2 manages lending positions through NFT-based troves and the **BOLD** stablecoin, handles special states like unredeemable positions, and provides flexible management options through both individual and batch delegation systems. This architecture enables efficient position tracking, specialized handling of edge cases, and scalable management of multiple positions.

10.2.1 Token Implementation

The **BOLD** token works as the protocol's stablecoin, while troves exist as NFTs. Each trove NFT represents a debt position against deposited collateral, containing data about the debt, collateral amount, and interest rate. The NFT implementation enables trove transfers between addresses, though transfers maintain existing delegations and management settings. The system mints a new NFT when a trove opens and burns it upon closure.

10.2.2 Unredeemable Troves

Troves enter an unredeemable state when redemptions reduce their debt below the minimum threshold. In this state, the last trove marked as zombie is used as the first one in the next redemption sequence. Unredeemable troves maintain their collateral and continue to accrue interest, though they cannot participate in interest rate adjustments. The owner must either bring the trove's debt back above minimum or close the position to exit this state.

10.2.3 Management System

The protocol implements delegation through individual and batch management systems. Individual delegation allows trove owners to appoint managers for specific operations, including collateral management, debt adjustments, and interest rate modifications. These managers operate within owner-defined parameters and permissions.

Batch management enables multiple troves to be managed as a group. Batch managers control the interest rates for all troves in their batch, operating within predefined minimum and maximum rate boundaries. The batch system includes management fees, which accrue to the batch manager based on the total debt under management. Batch operations process multiple troves in a single transaction, reducing gas costs for interest rate adjustments and other management functions.

10.2.4 Stability Pool

The Stability Pool holds **BOLD** tokens deposited by users who provide stability to the system. Depositors receive rewards in the form of liquidated collateral and interest yield from troves. The pool tracks deposits and rewards through a scalable system using snapshots and mathematical formulas to calculate accumulated gains. Users can choose to claim their collateral gains directly or stash them for future withdrawal, while yield gains can be added to their deposit or claimed as **BOLD** tokens. The pool processes liquidations by using deposited **BOLD** to repay liquidated trove debt in exchange for collateral.

10.3 Economic Mechanisms

The protocol implements mechanisms to manage collateral branch health, process position redemptions, and handle branch shutdowns. The system determines redemption allocation through branch debt metrics and position interest rates, with distinct processes for normal operations and shutdown scenarios.

10.3.1 Redemption Mechanics

The protocol's redemption system distributes redemptions across collateral branches based on the *unbackedness* of each branch. Unbackedness represents the difference between a branch's total **BOLD** debt and its Stability Pool deposits. When users redeem **BOLD** tokens, the system calculates redemption amounts for each branch proportionally to their unbackedness, ensuring branches with higher unbackedness receive more redemptions.

Within each branch, redemptions target troves based on their interest rates. The system processes redemptions starting from troves with the lowest interest rates, moving upward. This creates a redemption queue where troves with lower rates face redemptions before those with higher rates. For each redemption, users receive collateral at face value minus the redemption fee. The redemption fee increases with redemption volume and decays over time without redemptions.

During branch shutdown, the system enables urgent redemptions with no fee and a collateral bonus. These redemptions bypass the interest rate ordering, allowing users to specify which troves to redeem against. The shutdown redemption mechanism aims to reduce the branch's debt by incentivizing redemptions through the collateral bonus.

10.3.2 Gas Compensation

The system compensates liquidators for transaction costs through a combination of **WETH** and collateral tokens. Each trove sets aside a fixed amount of **WETH** (0.0375) upon opening, stored in the Gas Pool.

The compensation mechanism releases these funds to liquidators upon successful liquidation of a trove. If a trove closes normally, the **WETH** returns to the owner. This dual-token compensation structure balances the need to cover gas costs while limiting the impact on the trove's collateral. When a liquidation occurs, the trove Manager sends this compensation to the liquidator.

10.4 Risk Management

The protocol implements a multi-layered approach to risk management through collateralization thresholds, branch shutdown conditions, and liquidation processes. These mechanisms work together to maintain system solvency by enforcing borrowing limits, handling oracle failures, and processing undercollateralized positions through either the Stability Pool or redistribution to other troves.

10.4.1 Critical Collateral Ratio (CCR)

The protocol enforces borrowing restrictions through the Critical Collateral Ratio mechanism. When a branch's Total Collateral Ratio (TCR) falls below CCR, the system limits borrower operations. Users can only open new troves with ICR above CCR, while existing troves face restrictions on debt increases and collateral withdrawals. The CCR restrictions remain until the branch's TCR returns above the threshold. The system calculates TCR continuously using oracle prices and total collateral values.

10.4.2 Emergency Measures

The branch shutdown mechanism activates in two scenarios: when TCR falls below Shutdown Collateral Ratio (SCR), or when oracle failure occurs. Upon shutdown, the branch freezes borrower operations except for trove closures, stops interest accrual, and enables urgent redemptions. The system records the shutdown timestamp to manage the transition state. The shutdown remains permanent for that branch, preventing any reactivation.

The oracle failure path triggers when price feeds return invalid data, stale prices, or revert. The system uses a last-good-price mechanism for urgent redemptions after shutdown, allowing users to redeem their **BOLD** even when fresh prices become unavailable. Each collateral type maintains its own set of oracle parameters and failure conditions based on the underlying asset.

10.4.3 Liquidation Mechanics

The liquidation process incorporates variable penalties based on liquidation type. Stability Pool liquidations use a lower penalty than redistributions to other troves. The system calculates collateral seizure based on the liquidation penalty and returns any surplus to the borrower through the `CollSurplusPool`. The liquidation mechanism processes troves below Minimum Collateral Ratio (MCR).

The protocol distributes liquidated debt and collateral through two channels. The Stability Pool receives the first liquidation using deposited **BOLD** to repay debt. Any remaining liquidation redistributes to active troves proportional to their collateral. The redistribution mechanism tracks gains through a system of snapshots and applies them when troves undergo operations. Each liquidation updates system-wide variables to maintain accurate accounting of redistributed assets.

10.5 Zappers

Zappers are peripheral smart contracts that work as intermediaries for managing Liquity positions. These contracts implement position management through single entry points, integrate token swaps and flash loans, support native token operations, and execute multi-step position adjustments. They combine multiple protocol operations into single transactions and handle the implementation of swap and flash loan logic. These contracts, convert between native and wrapped token versions required by the protocol and execute position leverage modifications.

The threat model for Zapper contracts addresses several technical risk areas. Sender authentication requires `msg.sender` validation across execution contexts, access control mechanisms, and unauthorized call prevention. Token approvals require quantity management for protocol interactions, delegation security, and recipient validation. External interactions require trusted contract address verification, external call target integrity, and return data validation.

These contracts operate as intermediary layers between users and the core protocol, handling the orchestration of swaps, flash loans, and position management operations. This architecture allows direct position and leverage management through Zapper contracts rather than requiring separate implementation of the underlying swap and flash loan mechanisms.

11. Disclaimer

The contents of this report are provided "as is" without warranty of any kind. Coinspect is not responsible for any consequences of using the information contained herein.

This report represents a point-in-time and time-boxed evaluation conducted within a specific timeframe and scope agreed upon with the client. The assessment's findings and recommendations are based on the information, source code, and systems access provided by the client during the review period.

The assessment's findings should not be considered an exhaustive list of all potential security issues. This report does not cover out-of-scope components that may interact with the analyzed system, nor does it assess the operational security of the organization that developed and deployed the system.

This report does not imply ongoing security monitoring or guaranteeing the current security status of the assessed system. Due to the dynamic nature of information security threats, new vulnerabilities may emerge after the assessment period.

This report should not be considered an endorsement or disapproval of any project or team. It does not provide investment advice and should not be used to make investment decisions.