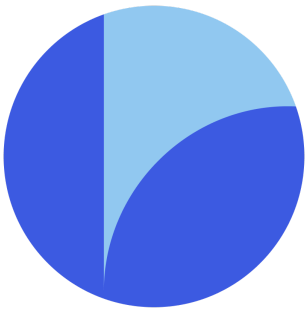
Liquity v2

Smart Contract Security Assessment

August 28, 2024



ABSTRACT

Dedaub was commissioned to perform a security audit of the Liquity v2 (BOLD) protocol, focusing on both the smart contract code and the overall logic of the system. The codebase was of high quality, with extensive comments and a detailed README that thoroughly describes all the aspects of the protocol. Additionally, the protocol is supported by an extensive test suite.

BACKGROUND

Liquity v2 is a decentralized lending protocol that mints a stablecoin called BOLD. The core mechanics are the same as in Liquity v1: users deposit collateral, and the protocol mints BOLD tokens. As long as a user's collateral ratio stays above a certain threshold, they remain safe from liquidation. However, if their collateral ratio drops below this threshold, anyone can trigger the liquidation process. The debt and collateral from a liquidated trove are absorbed by the stability pool, with any excess distributed proportionally to other troves based on their collateral. BOLD is redeemable at its face value of $1. It's important to note that while Liquity v2 shares similarities with v1, the two protocols are entirely independent.

Key Differences and New Features in Liquity v2:

1. **Multi-Collateral Support**: Liquity v2 introduces support for multiple collateral types, including wrapped ETH and various LSTs. Each collateral type forms its own branch with a separate set of troves and a dedicated stability pool.
2. **Interest Rate Mechanism**: Instead of paying a one-time fee when opening a trove, users in Liquity v2 select an annual interest rate, which they will pay over time. The interest is split between stability pool depositors and other liquidity providers, such as AMM LPs for pools containing BOLD. The percentage of interest allocated to the stability pool is fixed at deployment and cannot be changed.

3. **Redemption Process**: When redeeming BOLD tokens, users cannot choose which collateral they will receive. Instead, redemptions are distributed across the various collateral branches in proportion to their unbacked BOLD (i.e., total BOLD minted by the branch minus the amount of BOLD held in the branch's stability pool). Troves within each branch are ordered by their annual interest rate, with redemptions first targeting troves with lower interest rates. To avoid being targeted by redemptions, trove owners must adjust their interest rates. The protocol simplifies this process by allowing batch interest delegation. Any user can open an interest manager, charge a fee, and let trove owners join the manager to have their interest rates managed and updated.

SETTING & CAVEATS

This audit report covers the contracts of the at-the-time private repository https://github.com/liquity/bold, branch `dev`, of the Liquity v2 protocol at commit `2a859733eff540aae2996d13b06a9c5d334e7616`.

2 auditors worked on the codebase for 4 weeks on the following contracts:

```
src/
├── ActivePool.sol
├── AddressesRegistry.sol
├── BoldToken.sol
├── BorrowerOperations.sol
├── CollateralRegistry.sol
├── CollSurplusPool.sol
├── DefaultPool.sol
├── Dependencies/
│   ├── AddRemoveManagers.sol
│   ├── AggregatorV3Interface.sol
│   ├── Constants.sol
│   ├── IOsTokenVaultController.sol
│   ├── IRETHToken.sol
│   ├── IStaderOracle.sol
```

```
│   ├── LiquityBase.sol
│   ├── LiquityMath.sol
│   └── Ownable.sol
├── GasPool.sol
├── HintHelpers.sol
├── MultiTroveGetter.sol
├── PriceFeeds/
│   ├── CompositePriceFeed.sol
│   ├── ETHXPriceFeed.sol
│   ├── MainnetPriceFeedBase.sol
│   ├── OSETHPriceFeed.sol
│   ├── RETHPriceFeed.sol
│   ├── WETHPriceFeed.sol
│   └── WSTETHPriceFeed.sol
├── SortedTroves.sol
├── StabilityPool.sol
├── TroveManager.sol
├── TroveNFT.sol
├── Types/
│   ├── BatchId.sol
│   ├── LatestBatchData.sol
│   ├── LatestTroveData.sol
│   ├── TroveChange.sol
│   └── TroveId.sol
└── Zappers/
    ├── GasCompZapper.sol
    └── WETHZapper.sol
```

The audit's main target is security threats, i.e., what the community understanding would likely call "hacking", rather than the regular use of the protocol. Functional correctness (i.e. issues in "regular use") is a secondary consideration. Typically it can only be covered if we are provided with unambiguous (i.e. full-detail) specifications of what is the expected, correct behavior. In terms of functional correctness, we often trusted the code's calculations and interactions, in the absence of any other specification. Functional correctness relative to low-level calculations (including units, scaling and quantities returned from external protocols) is generally most effectively done through thorough testing rather than human auditing.

In this audit, we identified several accounting bugs, most of which are related to the batch interest delegation feature. Additionally, the Liquity team, who conducted an internal review and testing of the code in parallel with this audit, discovered a few more bugs (although these bugs are not included in this report, we have reviewed the fixes). The relatively high number of accounting bugs and the complexity of the batch delegation feature suggest that further testing and potentially a second round of auditing are necessary before deployment.

PROTOCOL-LEVEL CONSIDERATIONS

ID	Description	STATUS
P1	Risk of bad debt and potential loss of peg in the case of branch shutdown	**INFO**

The new version of Liquity supports multiple collaterals, with each collateral associated with its own branch, with its own set of troves and stability pool. While certain risks are isolated within each branch—such as liquidations being contained to the branch's stability pool and redistributions affecting only its troves—not all risks are similarly contained.

We are particularly concerned about the potential impact on other branches if one is shut down due to low collateralization (`TCR < SCR`) caused by a significant decrease in the collateral asset's price. In such a scenario, it's possible that the total collateral of the affected branch may not cover its total debt, leading to bad debt.

Even if the total collateral value across all branches exceeds the total remaining BOLD, theoretically ensuring that BOLD is fully backed, in practice, not all BOLD might be redeemable. This could trigger BOLD holders to rush for redemptions, leading to a decrease in BOLD's market price and potentially causing it to lose its peg.

P2	Aggressive shutdown triggered by failed oracle calls	**INFO**

The protocol relies on Chainlink oracles to obtain the price for the collateral. If an oracle call fails, or returns an invalid (zero) value or stale price, the protocol immediately initiates the shutdown procedure. While we understand that it is challenging to determine on-chain whether an oracle failure is temporary or permanent, the current approach of instantly shutting down the branch is overly aggressive. A failed oracle call could be due to a temporary issue, such as an error during an update to the Chainlink contracts, which may be resolved shortly. Instead of an immediate shutdown, the protocol could monitor for multiple consecutive failed oracle calls over a set period and temporarily freeze the branch during this time. This would allow the system to handle temporary disruptions more gracefully.

VULNERABILITIES & FUNCTIONAL ISSUES

This section details issues affecting the functionality of the contract. Dedaub generally categorizes issues according to the following severities, but may also take other considerations into account such as impact or difficulty in exploitation:

Category	Description
CRITICAL	Can be profitably exploited by any knowledgeable third-party attacker to drain a portion of the system's or users' funds OR the contract does not function as intended and severe loss of funds may result.
HIGH	Third-party attackers or faulty functionality may block the system or cause the system or users to lose funds. Important system invariants can be violated.
MEDIUM	Examples: • User or system funds can be lost when third-party systems misbehave. • DoS, under specific conditions. • Part of the functionality becomes unusable due to a programming error.
LOW	Examples: • Breaking important system invariants but without apparent consequences. • Buggy functionality for trusted users where a workaround exists. • Security issues which may manifest when the system evolves.

Issue resolution includes "dismissed" or "acknowledged" but no action taken, by the client, or "resolved", per the auditors.

CRITICAL SEVERITY:

[No critical severity issues]

HIGH SEVERITY:

ID	Description	STATUS
H1	Lack of access control in CollateralRegistry::setTroveManager	**RESOLVED**

`CollateralRegistry::setTroveManager()` has no access control, which means anyone can set and change the TroveManager of any branch.

`CollateralRegistry::setTroveManager():69-72`

```
function setTroveManager(
  uint256 _branch,
  ITroveManager _troveManager
) external {
  require(_branch < totalCollaterals, "Branch too high");
  troveManagers[_branch] = _troveManager;
}
```

The `CollateralRegistry` contract is responsible for routing redemptions to the various collateral branches based on the unbacked portion of BOLD tokens. The unbacked portion is calculated as the total amount of BOLD tokens minted by the branch minus the amount deposited into the stability pool of that branch. To perform this calculation, the `CollateralRegistry` interacts with the `TroveManager` contract of each branch.

Without access control on the `setTroveManager()` function, a malicious actor could deploy a `TroveManager` contract with altered logic and set its address in the `CollateralRegistry`. This could result in incorrect routing of redemptions.

This issue is probably related to the TODO comment in the declaration of the troveManagers storage variable, which used to be immutable.

MEDIUM SEVERITY:

ID	Description	STATUS
M1	Add Managers can grief debt repayments	**RESOLVED**

In `BorrowerOperations::_adjustTrove`, if the `_troveChange.debtDecrease > 0`, it is enforced that `vars.trove.entireDebt >= _troveChange.debtDecrease`. If the trove owner has allowed anyone (or a malicious party) to be able to decrease the trove's debt (by being an "add manager"), there exists the danger of someone frontrunning a debt repayment and blocking it by repaying a specific amount of debt, essentially DOSing debt repayments. In the general case, the amount of BOLD needed to deny a partial repayment would be `vars.trove.entireDebt - _troveChange.debtDecrease - MIN_DEBT + 1` (wei), where `MIN_DEBT` is the minimum amount of net Bold debt a trove must have. Setting the actual repayment amount to be the minimum of the `vars.trove.entireDebt - MIN_DEBT` and `_troveChange.debtDecrease` would render any such attack unsuccessful.

ID	Description	STATUS
M2	BorrowerOperations::applyPendingDebt does not check for batch inclusion and only reinserts normally	**RESOLVED**

In `BorrowerOperations::applyPendingDebt` it is not checked if the trove belongs to a batch when reinserting it back to the SortedTroves list in case it becomes redeemable. This omission breaks the SortedTroves list's invariant about batch troves and might affect other batch-related operations such as the `removeFromBatch` one.

`BorrowerOperations::applyPendingDebt:808-811`

```
if (
```

```
  _checkTroveIsUnredeemable(troveManagerCached, _troveId) &&
  trove.entireDebt >= MIN_DEBT
) {
  troveManagerCached.setTroveStatusToActive(_troveId);
  sortedTroves.insert(
    _troveId, trove.annualInterestRate, _upperHint, _lowerHint
  );
}
```

In contrast, the distinction between batched and non-batched troves is made by the `adjustUnredeemableTrove` function as can be seen in the code snippet below:

`BorrowerOperations::adjustUnredeemableTrove:520-530`

```
address batchManager = interestBatchManagerOf[_troveId];
if (batchManager == address(0)) {
  sortedTroves.insert(
    _troveId,
    troveManagerCached.getTroveAnnualInterestRate(_troveId),
    _upperHint,
    _lowerHint
  );
} else {
  LatestBatchData memory batch =
    troveManagerCached.getLatestBatchData(batchManager);
  sortedTroves.insertIntoBatch(
    _troveId,
    BatchId.wrap(batchManager),
    batch.annualInterestRate,
    _upperHint,
    _lowerHint
  );
}
```

M3	Opening a batched Trove is allowed during shutdown	**RESOLVED**

When a collateral branch has been shut down, several actions are restricted/disabled to prevent further deterioration of the protocol's economic state. The documentation thoroughly describes the set of restricted actions, including the prohibition on opening a new trove. While the `BorrowerOperations::openTrove` function implements the restriction by reverting if the branch has been shut down, the same check is missing from the `BorrowerOperations::openTroveAndJoinInterestBatchManager` function. This omission allows for the opening of batched troves even during a shutdown, which contradicts the intended restrictions and could lead to potential economic risks for the protocol.

M4	Missing approval in WETHZapper	**RESOLVED**

The `adjustTroveWithRawETH` and `adjustUnredeemableTroveWithRawETH` functions in the `WETHZapper` contract are designed to adjust a trove using raw ETH as collateral. These functions rely on the `_adjustTrovePre` function to handle pre-adjustment logic, which, in the case of collateral increase, deposits the ETH sent during the call and mints WETH for the `WETHZapper` contract.

`WETHZapper::_adjustTrovePre:184-187`

```
// ETH -> WETH
if (_isCollIncrease) {
  WETH.deposit{value: _collChange}();
  // Dedaub: raw ETH is deposited to get WETH, but BorrowerOperations
  // is not being approved to transfer this WETH amount
}
```

However, the `WETHZapper` contract does not approve the `BorrowerOperations` contract to transfer the newly minted WETH. As a result, when the `BorrowerOperations` functions (`adjustTrove`, `adjustUnredeemableTrove`) are invoked, they attempt to transfer the WETH collateral from the `WETHZapper` contract to the active pool but revert due to the lack of approval.

M5	Opening or adjusting a trove will succeed even if the transaction causes the shutdown of the branch	**RESOLVED**

A collateral branch can be shut down either if its total collateral ratio (TCR) falls below a certain threshold (SCR) or if the price oracle fails to return a valid price. After a shutdown, several actions, such as opening or adjusting a trove, are prohibited to prevent further risks to the protocol.

The shutdown can occur through two main mechanisms:

1. Manual Trigger via `BorrowerOperations::shutdown`: This function checks if the TCR is below the SCR, and if so, it initiates the shutdown.
2. Automatic Trigger via `fetchPrice`: If the price oracle fails to return a valid price, this function will raise the shutdown flag for the branch but will still return the last valid price (`lastGoodPrice`) without causing an immediate revert in the calling function.

The `BorrowerOperations::openTrove` function first checks if the branch has already shut down. If the branch is not shut down, it proceeds to call `_openTrove`, which internally calls `fetchPrice`. If `fetchPrice` triggers a shutdown due to an invalid oracle price, the shutdown will be executed, but the `openTrove` action will still proceed. Consequently:

- The new trove will be opened even though the branch has been shut down.
- The approximate average rate used to calculate the upfront fee will be zero due to the `hasBeenShutDown` flag being raised in the `ActivePool`.
- Since interest is not applied after shutdown, this new trove will also avoid interest payments.

`BorrowerOperations::_openTrove:345-361`

```
vars.price = priceFeed.fetchPrice();
// Dedaub: fetchPrice can trigger a branch shutdown, without
```

```
// reverting the action

// --- Checks ---

_requireNotBelowCriticalThreshold(vars.price);

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

_troveChange.collIncrease = _collAmount;
_troveChange.debtIncrease = _boldAmount;

// For simplicity, we ignore the fee when calculating the approx.
// interest rate
_troveChange.newWeightedRecordedDebt = _troveChange.debtIncrease *
  _annualInterestRate;

// Dedaub: if fetchPrice has already shut down the branch, the
// getNewApproxiAvgInterestRateFromTroveChange will return 0, therefore
// the upfront fee for opening the trove will be 0.
vars.avgInterestRate = vars.activePool.
  getNewApproxAvgInterestRateFromTroveChange(_troveChange);
_troveChange.oldWeightedRecordedDebt = vars.batch.weightedRecordedDebt
_troveChange.upfrontFee = _calcUpfrontFee(
  _troveChange.debtIncrease, vars.avgInterestRate
);
```

The functions calling `_adjustTrove` have the same issues, as `_adjustTrove` calls `fetchPrice`.

M6	BorrowerOperations::_adjustTrove does not apply the redistribution debt gain of a batched trove to the weighted recorded debt	**RESOLVED**

The `BorrowerOperations::_adjustTrove()` function does not add the `redistBoldDebtGain` of a batched trove to the `batchFutureDebt` that is used in the

`_troveChange.newWeightedRecordedDebt` and `_troveChange.newWeightedRecordedBatchManagementFee` calculations.

`BorrowerOperations::_adjustTrove():628-649`

```
vars.newColl = vars.trove.entireColl +
  _troveChange.collIncrease - _troveChange.collDecrease;
// Dedaub: the new debt includes the trove's entire debt, which includes
//         the trove's redistribution debt gain
vars.newDebt = vars.trove.entireDebt +
  _troveChange.debtIncrease - _troveChange.debtDecrease;

address batchManager = interestBatchManagerOf[_troveId];
bool isTroveInBatch = batchManager != address(0);
LatestBatchData memory batch;
uint256 batchFutureDebt;
if (isTroveInBatch) {
  batch = _troveManager.getLatestBatchData(batchManager);
  // Dedaub: the future debt of the batch should include the entire batch
  // debt without the batch troves' redistribution debt except from the
  // redistribution debt of the trove that is currently adjusted, thus
  // vars.trove.redistBoldDebtGain should be added to batchFutureDebt
  batchFutureDebt = batch.entireDebtWithoutRedistribution +
    _troveChange.debtIncrease - _troveChange.debtDecrease;

  // Dedaub: code omitted for brevity
}
```

As a result, in `ActivePool::mintAggInterestAndAccountForTroveChange`, this redistributed debt is not added to the `aggWeightedDebtSum` and does not accumulate interest. However, in the call to `TroveManager::onAdjustTroveInsideBatch`, the same applied redistributed debt is recorded as it is passed via the `_troveChange` struct variable and it is used by `_updateBatchShares()` to calculate the new total debt of the batch.

M7	BorrowerOperations::applyPendingDebt does not apply the redistribution debt gain of a batched trove to the weighted recorded debt	**RESOLVED**

The `BorrowerOperations::applyPendingDebt()` function does not add the `redistBoldDebtGain` of a batched trove to the `batch.entireDebtWithoutRedistribution` (similarly to what is described in issue M6) when calculating the `_troveChange.newWeightedRecordedDebt` and `_troveChange.newWeightedRecordedBatchManagementFee`.

M8	Opening a batched Trove ignores the batch entire debt when updating the new batch weighted recorded debt	**RESOLVED**

The computation of the `avgInterestRate` in `BorrowerOperations::_openTrove` appears to be incorrect when the caller of the `_openTrove` function is `openTroveAndJoinInterestBatchManager`. The `newWeightedRecordedDebt` is set to `_troveChange.debtIncrease * _annualInterestRate` when passed to `getNewApproxAvgInterestRateFromTroveChange`. However, it appears that the batch debt should also be included, i.e., `newWeightedRecordedDebt = (_batchEntireDebt+_troveChange.debtIncrease) * _annualInterestRate`, as the `oldWeightedRecordedDebt` is set to `vars.batch.weightedRecordedDebt`.

M9	Setting batch interest rate calculates the weighted management fee incorrectly	**RESOLVED**

In `BorrowerOperations::setBatchManagerAnnualInterestRate`, the new weighted management fee is computed as `newDebt * _newAnnualInterestRate`. This calculation incorrectly uses the new annual interest rate instead of the `annualManagementFee`.

`BorrowerOperations::setBatchManagerAnnualInterestRate:926-929`

```
batchChange.oldWeightedRecordedDebt = batch.weightedRecordedDebt;
batchChange.newWeightedRecordedDebt = newDebt * _newAnnualInterestRate;
```

```
batchChange.oldWeightedRecordedBatchManagementFee =
  batch.weightedRecordedBatchManagementFee;
batchChange.newWeightedRecordedBatchManagementFee =
  newDebt * _newAnnualInterestRate;
// Dedaub: it should be
// batchChange.newWeightedRecordedBatchManagementFee =
//   newDebt * batch.annualManagementFee
```

M10	Premature batch interest rate adjustments do not update the weighted management fee	**RESOLVED**

In `BorrowerOperations::setBatchManagerAnnualInterestRate`, in the `if` branch where the upfront fee is applied, the `batchChange.newWeightedRecordedDebt` is updated accordingly to take the upfront fee into account, but the `batchChange.newWeightedRecordedBatchManagementFee` is not.

M11	Incorrect calculation in batched trove redemptions leads to error in accrued interest and management fee	**RESOLVED**

First of all, in certain cases `TroveManager::_applySingleRedemption` handles differently the calculation of `oldWeightedRecordedDebt` and `oldWeightedRecordedBatchManagementFee` when it should not. More precisely, `troveOldWeightedRecordedDebt`, which is equal to the `min(trove.entireDebt - trove.redistBoldDebtGain, boldLot)`, is used in the calculation of `oldWeightedRecordedDebt`. On the contrary, `boldLot` is used to compute `oldWeightedRecordedBatchManagementFee` even though it might be greater than `trove.entireDebt - trove.redistBoldDebtGain`.

At the same time, it appears that neither of the two approaches is always correct, which can lead to errors in the accrued interest and management fee over time. If we leave out from the calculation contribution of the other troves of the batch and focus on the one being redeemed, we would expect the total debt to change by

`trove.redistBoldDebtGain - boldLot` or the `oldWeightedRecordedDebt` to be greater by `(boldLot - trove.redistBoldDebtGain) * batch.annualInterestRate` (this can underflow but let's not focus on that right now) from the `newWeightedRecordedDebt`. However, this is not always the case due to how `troveOldWeightedRecordedDebt` is calculated at the moment. We can distinguish the following 3 scenarios:

1. if `trove.entireDebt == boldLot` then
   `troveOldWeightedRecordedDebt = trove.entireDebt - trove.redistBoldDebtGain`
   which is equal to the expected
   `boldLot - trove.redistBoldDebtGain`

2. if `trove.entireDebt > boldLot` and
   `boldLot > trove.entireDebt - trove.redistBoldDebtGain` then
   `troveOldWeightedRecordedDebt = trove.entireDebt - trove.redistBoldDebtGain`
   which is greater than the expected
   `boldLot - trove.redistBoldDebtGain`

3. if `trove.entireDebt > trove.entireDebt - trove.redistBoldDebtGain` and `trove.entireDebt - trove.redistBoldDebtGain > boldLot` then
   `troveOldWeightedRecordedDebt = boldLot`
   which is greater than the expected
   `boldLot - trove.redistBoldDebtGain`

In scenarios 2 and 3 the `troveOldWeightedRecordedDebt` is incorrect, leading to errors in accrued interest and management fee over time.

M12	BorrowerOperations::openTroveAndJoinInterestBatchManager does not take into account the accrued batch management fee	**RESOLVED**

In function `BorrowerOperations::openTroveAndJoinInterestBatchManager`, `vars.change` should also store the `batch.accruedBatchManagementFee`, as otherwise this amount does not reach the `ActivePool::mintAggInterestAndAccountForTroveChange` (called by `_openTrove`) to be minted as debt, while it is passed everywhere else (`TroveManager::onOpenTroveAndJoinBatch` and `_openTrove`) through the `vars.batch.entireDebtWithoutRedistribution` to update the state of the batch.

M13	TroveManager::onOpenTroveAndJoinBatch does not update the trove's lastInterestRateAdjTime	**RESOLVED**

The `TroveManager::onOpenTroveAndJoinBatch` function does not set the trove's `lastInterestRateAdjTime` to the `block.timestamp` as one would expect. In contrast, `TroveManager::onSetInterestBatchManager`, which essentially performs the `JoinBatch` part of the `onOpenTroveAndJoinBatch` function, updates the `lastInterestRateAdjTime` . As a result, when removing from a batch a trove that has been added to it by `onOpenTroveAndJoinBatch`, if the interest adjustment is considered premature because not enough time has passed since joining the batch, no premature adjustment fee will be paid as the `lastInterestRateAdjTime` has not been tracked correctly.

M14	BorrowerOperations::removeFromBatch interest rate premature adjustments checks are not strict enough	**RESOLVED**

The function `BorrowerOperations::removeFromBatch` checks if the removal of the trove from the batch will lead to a new interest rate for the trove and if this is true, checks if the last interest adjustment of the batch is old enough to not incur any interest rate premature adjustment fees.

`BorrowerOperations::removeFromBatch:1072-1078`

```
if (
  vars.batch.annualInterestRate != _newAnnualInterestRate &&
  block.timestamp < vars.batch.lastInterestRateAdjTime +
    INTEREST_RATE_ADJ_COOLDOWN
) {
  vars.trove.entireDebt = _applyUpfrontFee(
    vars.trove.entireColl,
    vars.trove.entireDebt,
    batchChange,
    _maxUpfrontFee
  );
}
```

However, the trove's individual `lastInterestRateAdjTime` is not taken into account in the aforementioned checks, meaning that if a trove joins the batch and is removed before `INTEREST_RATE_ADJ_COOLDOWN` has passed, there will be no upfront fee paid in case the last interest rate adjustment for the batch happened more than `INTEREST_RATE_ADJ_COOLDOWN` seconds ago. Instead, the time check should use the `vars.trove.lastInterestRateAdjTime`, which is equal to the maximum of the the `lastInterestRateAdjTime` of the trove and of its ex-batch.

LOW SEVERITY:

ID	Description	STATUS
L1	Incorrect requirement in CollateralRegistry's constructor	**RESOLVED**

In the `CollateralRegistry` constructor, the condition of the second `require` statement should be `numTokens <= 10` instead of `numTokens < 10`, since there are variables for at most 10 tokens (token0 - token9).

L2	No way to revoke a Remove Manager	**RESOLVED**

Currently there is no way to invalidate a "remove" manager by setting its `receiver` to `0` after it is set to a non-zero address value. However, the function `AddRemoveManagers::_requireSenderIsOwnerOrRemoveManager` requires that `msg.sender == _owner` when `receiver == address(0)` and at the same time returns the `_owner` as the `receiver`, thus there is no reason to not allow setting a remove manager's `receiver` to `0`.

L3	Looser modifier in BoldToken::returnFromPool	**RESOLVED**

`BoldToken::returnFromPool` requires that its caller is either the `TroveManager` or the `StabilityPool` (`_requireCallerIsTroveMorSP()`) when the current `TroveManager` implementation does not call `BoldToken::returnFromPool`.

`BoldToken::returnFromPool:133-138`

```
function _requireCallerIsTroveMorSP() internal view {
  require(
    troveManagerAddresses[msg.sender] ||
      stabilityPoolAddresses[msg.sender],
    "Bold: Caller is neither TroveManager nor StabilityPool"
  );
}
```

Thus, it should be enough to use `_requireCallerIsStabilityPool()`.

L4	Incorrect order of values in TroveUpdated and BatchedTroveUpdated events	**RESOLVED**

The last two fields of the `TroveUpdated` and `BatchedTroveUpdated` events are `_snapshotOfTotalDebtRedist` and `_snapshotOfTotalCollRedist`. When these two events are used however, `L_coll` is assigned to `_snapshotOfTotalDebtRedist` and `L_boldDebt` is assigned to `_snapshotOfTotalCollRedist`.

L5	Trove's interest rate delegate's restrictions affect the owner	**RESOLVED**

The `BorrowerOperations::_requireInterestRateInDelegateRange` function, which is called only by `adjustTroveInterestRate`, checks if the trove's owner has set an interest rate delegate and if this is the case requires the new interest rate to be between the min and max values set for this delegate. However, the function does not distinguish the case in which the caller is the owner and thus should be able to bypass the restrictions that exist for the delegate.

`BorrowerOperations::_requireInterestRateInDelegateRange:1273-1280`

```
function _requireInterestRateInDelegateRange(
  uint256 _troveId, uint256 _annualInterestRate
) internal view {
  InterestIndividualDelegate memory individualDelegate =
    interestIndividualDelegateOf[_troveId];
  // Dedaub: the condition does not take into account the msg.sender
  if (individualDelegate.account != address(0)) {
    _requireInterestRateInRange(
      _annualInterestRate,
      individualDelegate.minInterestRate,
      individualDelegate.maxInterestRate
    );
  }
}
```

The condition should be changed to `individualDelegate.account != address(0) && msg.sender != owner` or just to `individualDelegate.account == msg.sender`.

L6	BorrowerOperations::setInterestIndividualDelegate is missing value sanitization checks	**RESOLVED**

The function `BorrowerOperations::setInterestIndividualDelegate` does not check that the `_delegate` parameter is not `address(0)`. Also, it does not ensure that `_minInterestRate` and `_maxInterestRate` are valid interest rates according to `_requireValidAnnualInterestRate` and that `_minInterestRate < _maxInterestRate`.

L7	StabilityPool::claimAllCollGains can be called even if the caller's stashed collateral balance is 0	**RESOLVED**

Anyone is able to call `StabilityPool::claimAllCollGains` to claim their stashed collateral balance, even if it is `0`. The function will mint any pending aggregate interest first and then it will set the caller's stashed collateral balance to 0 and send them their previously stashed amount.

`StabilityPool::claimAllCollGains:377-389`

```
function claimAllCollGains() external {
  _requireUserHasNoDeposit(msg.sender);
  // Dedaub: check that the caller's stashed balance is not 0

  activePool.mintAggInterest();

  uint256 collToSend = stashedColl[msg.sender];
  stashedColl[msg.sender] = 0;

  emit DepositOperation(msg.sender, Operation.claimAllCollGains,
    0, 0, 0, 0, 0, collToSend);
  emit DepositUpdated(msg.sender, 0, 0, 0, 0, 0, 0, 0);

  _sendCollGainToDepositor(collToSend);
}
```

Even though claiming a 0 balance does not appear to cause any harm, we would advise in favor of disallowing this possibility.

L8	StabilityPool::DepositOperation events do not take into account the keptYieldGain for the deposit change	**DISMISSED**

`StabilityPool::provideToSP` emits a `DepositOperation` event with `_depositChange == int256(_topUp)`, ignoring the `keptYieldGain` amount, which might be also adding to the deposit. The same is true for the `DepositOperation` event in `withdrawFromSP` where `_depositChange` should equal `-int256(boldToWithdraw) + int256(keptYieldGain)`

L9	Inaccuracy in the upfront fee calculation due to approximation in the weighted average interest rate	**ACKNOWLEDGED**

When a user opens or adjusts the debt of a trove or updates its interest rate within a short interval (sooner than the `INTEREST_RATE_ADJ_COOLDOWN` period), the protocol charges an upfront fee based on the `UPFRONT_INTEREST_PERIOD` of the average weighted interest rate. This average weighted interest rate is calculated by the `ActivePool::getNewApproxAvgInterestRateFromTroveChange` function.

`ActivePool::getNewApproxAvgInterestRateFromTroveChange:138-164`

```
function getNewApproxAvgInterestRateFromTroveChange(
  TroveChange calldata _troveChange
) external view returns (uint256) {
  // We are ignoring the upfront fee when calculating the approx.
  // avg. interest rate.
  // This is a simple way to resolve the circularity in:
  // fee depends on avg. interest rate -> avg. interest rate is
  // weighted by debt -> debt includes fee -> ...
  assert(_troveChange.upfrontFee == 0);

  if (hasBeenShutDown) return 0;
```

```
  uint256 newAggRecordedDebt = aggRecordedDebt;
  // Dedaub: the pending interest of all the troves is added
  newAggRecordedDebt += calcPendingAggInterest();
  newAggRecordedDebt += _troveChange.appliedRedistBoldDebtGain;
  newAggRecordedDebt += _troveChange.debtIncrease;
  newAggRecordedDebt -= _troveChange.debtDecrease;

  uint256 newAggWeightedDebtSum = aggWeightedDebtSum;
  // Dedaub: the new weighted debt sum takes into account only
  // the change in the trove under consideration and not the
  // pending interest of all the other troves
  newAggWeightedDebtSum += _troveChange.newWeightedRecordedDebt;
  newAggWeightedDebtSum -= _troveChange.oldWeightedRecordedDebt;

  // Avoid division by 0 if the first ever borrower tries to borrow 0 BOLD
  // Borrowing 0 BOLD is not allowed, but our check of debt >= MIN_DEBT
  // happens _after_ calculating the upfront fee, which involves getting
  // the new approx. avg. interest rate
  return newAggRecordedDebt > 0 ?
    newAggWeightedDebtSum / newAggRecordedDebt : 0;
}
```

As the function's name suggests, the calculated rate is approximate, primarily because it includes the updated debt of the trove without factoring in the upfront fee to avoid cyclic dependency (upfront fee depends on the average interest which depends on the upfront fee). However, further inaccuracies stem from how this average is computed:

- The denominator includes the total recorded debt of all troves, along with the pending debt.
- The numerator, which calculates the weighted sum of interest rates, only adds the pending interest of the updated trove, neglecting the pending interests of other troves.

This leads to the weighted average being calculated with weights that sum to less than one, potentially making the computed average interest rate lower than all individual trove interest rates, which is counterintuitive. The inaccuracy increases as the pending fees become a larger proportion of the total debt.

Comments:
Since this inaccuracy will be negligible under most circumstances, and a fix would require significant code changes, the Liquity team has decided not to address this issue. More details for this issue can be found in the list of known issues.

L10	BorrowerOperations::setInterestIndividualDelegate does not check the status of the Trove	**RESOLVED**

The function `BorrowerOperations::setInterestIndividualDelegate` does not require the trove's to be active (`status == ITroveManager.Status.active`) or even open (`status == ITroveManager.Status.active || status == ITroveManager.Status.unredeemable`).

OTHER / ADVISORY ISSUES:

This section details issues that are not thought to directly affect the functionality of the project, but we recommend considering them.

ID	Description	STATUS
A1	Applying Trove interest sets Trove properties Twice	**RESOLVED**
In `TroveManager::onApplyTroveInterest` the `coll`, `debt` and `lastDebtUpdateTime` properties of a batch are set twice, once outside `_updateBatchShares` and once inside it, overriding the first assignment.		
A2	Functions that can be external instead of public	**RESOLVED**
The following functions, which are currently `public`, could be made `external`: • `CollateralRegistry::getEffectiveRedemptionFeeInBold`		
A3	BorrowerOperations::removeFromBatch does not directly check if the to be removed trove belongs in a batch	**RESOLVED**
`BorrowerOperations::removeFromBatch` does not check early enough that the trove to be removed does indeed belong to a batch. The execution will revert when the `vars.sortedTroves.removeFromBatch(_troveId)` is reached.		
A4	Storage variable than can be made immutable	**RESOLVED**
The following storage variables can be made immutable: • CollSurplusPool::borrowerOperationsAddress • CollSurplusPool::troveManagerAddress • CollSurplusPool::activePoolAddress • DefaultPool::troveManagerAddress • DefaultPool::activePoolAddress		

- StabilityPool::borrowerOperations
- StabilityPool::troveManager
- StabilityPool::boldToken
- StabilityPool::sortedTroves
- LiquityBase::activePool
- LiquityBase::defaultPool
- LiquityBase::priceFeed
- TroveNFT::troveManager
- SortedTroves::borrowerOperationsAddress
- SortedTroves::troveManager

A5	Deprecated documentation link	**RESOLVED**

The documentation link in GasPool.sol points to the Liquity v1 Github repo.

A6	Uninitialized storage	**RESOLVED**

The `StabilityPool` contract does not initialize/set the `defaultPool` storage variable that inherits from the `LiquityBase` contract. The `DefaultPoolAddressChanged` event defined by the `StabilityPool` is also not used. We would advise to define a LiquityBase constructor that would be responsible for initializing its storage.

A7	TroveManager::getLatestBatchData duplicated calculation	**RESOLVED**

The function `TroveManager::_getLatestBatchData` unnecessarily computes `latestBatchData.recordedDebt * latestBatchData.annualManagementFee` twice.

`TroveManager::_getLatestBatchData():1004-1007`

```
latestBatchData.accruedManagementFee =
  _calcInterest(latestBatchData.recordedDebt *
    latestBatchData.annualManagementFee, period);
// Dedaub: weightedRecordedBatchManagementFee could be computed first to
```

```
// be reused in the calculation of the accruedManagementFee
latestBatchData.weightedRecordedBatchManagementFee =
  latestBatchData.recordedDebt * latestBatchData.annualManagementFee;
```

A8	TroveManager::urgentRedemption could fail early	**RESOLVED**

`TroveManager::urgentRedemption` does not check that its caller (`msg.sender`) holds the specified redeemed amount of Bold tokens (`_boldAmount`) until the very end of the function's execution where the burning of the tokens occurs. Adding the respective check in the beginning of the function would result in early failures and save users gas.

A9	TroveManager::urgentRedemption could break early	**RESOLVED**

`TroveManager::urgentRedemption` loops over the `_troveIds` array and calls `_urgentRedeemCollateralFromTrove` on each one of the provided troves, decreasing the `remainingBold` amount in every iteration. The loop could break early in case `remainingBold == 0` to avoid performing another iteration when there are no more funds available.

A10	Unnecessary storage read	**RESOLVED**

In function `TroveManager::batchLiquidateTroves` the storage variable `activePool` is read twice, once to be stored in the `activePoolCached` local variable and once more to execute `activePool.mintAggInterestAndAccountForTroveChange(troveChange, address(0));` when the cached value (`activePoolCached`) could be used.

A11	Unnecessary call to TroveManager::_computeNewStake	**RESOLVED**

In `TroveManager::onRemoveFromBatch`, the `_computeNewStake` function is called to compute the new stake of the trove when we know that there is no difference between

the old (in the batch) and new (out of the batch) stake as there has been no direct adjustment on the trove's collateral.		
A12	Division op can be turned to multiplication	**RESOLVED**
In `BorrowerOperations::_requireDebtRepaymentGeCollWithdrawal` the condition `_troveChange.debtDecrease < _troveChange.collDecrease * _price / DECIMAL_PRECISION` can be turned to `_troveChange.debtDecrease * DECIMAL_PRECISION < _troveChange.collDecrease * _price`.		
A13	Deprecated TODO comment	**RESOLVED**
There exists a deprecated TODO comment in `BorrowerOperations::setBatchManagerAnnualInterestRate`.		
A14	Unused event	**RESOLVED**
The `TroveManager::TroveNotOpen` error is never used.		
A15	Unused internal function	**RESOLVED**
The internal function `_requireValidKickbackRate` within the `StabilityPool` contract is never called by any other function in the codebase and therefore it should be removed.		
A16	Unused return value	**RESOLVED**
`BorrowerOperations::closeTrove` returns the trove's entire collateral amount before closure (`trove.entireColl`) but no caller uses this information.		
A17	Compiler bugs	**INFO**
The code is compiled with Solidity `0.8.18`. Version `0.8.18`, in particular, has some known bugs, which we do not believe affect the correctness of the contracts.		

DISCLAIMER

The audited contracts have been analyzed using automated techniques and extensive human inspection in accordance with state-of-the-art practices as of the date of this report. The audit makes no statements or warranties on the security of the code. On its own, it cannot be considered a sufficient assessment of the correctness of the contract. While we have conducted an analysis to the best of our ability, it is our recommendation for high-value contracts to commission several independent audits, a public bug bounty program, as well as continuous security auditing and monitoring through Dedaub Security Suite.

ABOUT DEDAUB

Dedaub offers significant security expertise combined with cutting-edge program analysis technology to secure some of the most prominent protocols in DeFi. The founders, as well as many of Dedaub's auditors, have a strong academic research background together with a real-world hacker mentality to secure code. Protocol blockchain developers hire us for our foundational analysis tools and deep expertise in program analysis, reverse engineering, DeFi exploits, cryptography and financial mathematics.