Liquity
Smart Contract Audit
1. Executive Summary

2. Introduction

3. Assessment

4. Summary of Findings

5. Findings
   - CI-LQY-01 - TCR manipulation enables sudden liquidations
   - CI-LQY-02 - liquidateBatch does not verify processed troves are not closed
   - CI-LQY-03 - Inflating fees facilitated in low LQTY pool participation scenario
   - CI-LQY-04 - Liquidations incentives misaligned with system overall health (TCR)

6. Appendix

7. Disclaimer
1. Executive Summary

In March 2021, Liquity engaged Coinspect to perform its second third-party source code review of the smart contracts that comprise the Liquity Protocol. The goal of the project was to evaluate the security of the smart contracts.

The assessment was conducted on the private Github repository liquity/dev main branch up to commit hash dd7f59b9 over the course of 9 person-weeks.

The reviewed Solidity code is well written and very clear. The documentation is extensive, and includes formal mathematical proofs for the correctness of the math behind the protocol. A complete set of tests with almost perfect coverage is included in the repository as well.

The following issues were identified during the assessment:

<table>
<thead>
<tr>
<th>Level of Risk</th>
<th>Issue Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>High Risk</td>
<td>1</td>
</tr>
<tr>
<td>Medium Risk</td>
<td>2</td>
</tr>
<tr>
<td>Low Risk</td>
<td>1</td>
</tr>
</tbody>
</table>

The high risk issue CI-LQY-01 is about a missing requirement in function closeTrove that allows forcing the system to enter Recovery Mode in order to liquidate troves. This finding was promptly fixed by Liquity’s team during the assessment and the resulting code was verified by Coinspect.

The medium risk issue CI-LQY-03 shows how attackers could leverage flash loans to inflate system fees, especially during the first period after deployment when the system is expected to have low total debt, in the context of low participation in the LQTY staking pool.

The medium risk issue CI-LQY-04 calls attention to how after the introduction of batch liquidations, the liquidators incentives are misaligned with the system total collateralization ratio and could affect the health of the system during a ETH price drop.

The low risk issue CI-LQY-02 is about missing checks in the liquidateBatch function. This poses no risk at the moment. This finding has been correctly addressed by Liquity’s team.

Finally, the appendix in section 7 highlights minor suggestions regarding typos, test coverage, optimizations, etc.

Off-chain components such as the front-end were out of scope for this assessment, and it is recommended to audit them in the future.
2. Introduction

The audit started on March 1st and was conducted on the contracts from the private Github repository liquidity/dev main branch up to commit hash dd759b9:

commit dd759b980e7dab1cebc84c017db3a2c4aa522c
Author: Rick <ric..@..il.com>
Date:   Fri Mar 19 17:08:48 2021 +0000

contracts: Fix closeTrove test

The scope of the audit was limited to the following Solidity source files, shown here with their sha256sum hash:

```
fb6f8e6af4f6e392bdfde3fba7df8e6921666afe1bc3d3137ffe4f4b3808672c05c8a43571679f8089e7duc2d3a832a8758ee23a67b2b20e6b38e5609d34d3bca54571e9eac08327c3ca0ac69bbf62c62b0d876e6a7f5a9f98554a943aee383b79a1a604ff6b3bd2f29a90a38a957b893582e23f3d72b2a9e365706d7d89668986d067ed8e95ba9668e6af518e760988f32481726e4d3c424ea9f85774720d9f2e28e346a6f8546ee93955570ec68a30c21b3c5803e2a62156716a26b04b8f7e171e232baaee485e60443ee557f1416e6d5b0d4a27976b749a7c7a424a7f4ac7a308939d8e7c98963425ac5bab98d90c95712442889bfec87baaf0c6156822625a9a10c8c303ee9233e3ac8ba4bc8cb858506f747a4c4f6f1951bda8f605434b8ad8f05bc6f3c25cde49df09734a2c23a2daeb5525a84e2f675c5d258073d215577b7c954sf0eb03d9f36388cb73d05e12264a49d9e9a91329c26a676ca2c1e130220cf895a29d4d90975eb3c668d31742ab4ab12c952ceb91b4f7e383b9a8e98449dbd8c7fe15886ee6f3ef5f9af8c8ffdf35955f
```
Liquity is a collateralized debt platform that allows users to lock up ETH and borrow LUSD tokens. The Liquity protocol issues a stablecoin (LUSD), intended to maintain a value of 1 LUSD = $1 USD. The Liquity protocol allows users to redeem LUSD for ETH: for x LUSD you get x USD worth of ETH in return.

A given position associated with an Ethereum address is called a “trove.” When a user makes a deposit of ETH, a trove is created with the collateral ETH provided by the user. In normal circumstances a user can add ETH to their collateral deposit, borrow LUSD, redeem LUSD for ETH at face value, or withdraw ETH from their collateral deposit, as long as a minimum collateralization ratio (MCR) of 110% is maintained. Troves with a collateralization ratio below 110% can be subject to liquidation.

The Liquity protocol also involves “stability providers” that deposit LUSD in a pool that is used for covering the debt of liquidated troves, and as an incentive they receive excess collateral from liquidated troves (this excess should be close to 10%, since troves can be liquidated as soon as their collateralization ratio drops below 110%).

The protocol also issues an LQTY token that is used to distribute among the holders a share of the revenue generated from redemption fees and LUSD issuance fees. LQTY tokens are also issued as incentive to third-parties that provide front-ends for the protocol (for example web front-ends or apps) and stability providers, and also (under a vesting schedule) to team members and partners.
3. Assessment

Most of the contracts (with the exception of a few dependencies and test contracts) are specified to be compiled with Solidity version 0.6.11. This is not the latest version of the 0.6.x series, and it is recommended to upgrade to 0.6.12 released in July 2020, and also consider using a newer series such as 0.7.x or 0.8.x.

The contracts compile without warnings, except six minor warnings about missing a SPDX license identifier and some functions that could be restricted to `view` or `pure` (see appendix). Linting the contract’s source files with `solhint` produces 412 errors, but all are unimportant, all are about lines exceeding 120 characters.

The repository includes tests, and they have very good coverage. However, it was found that most of the events emitted by the contracts are not tested (see appendix).

The reviewed Solidity code is well written and very clear. The documentation is very extensive, and includes formal mathematical proofs for the correctness of the math behind the protocol.

After initialization, all contracts are fully immutable. This means that the system is truly decentralized and algorithmic, without any governing entity like an owner or an administrator. This guarantees that the Liquity protocol will never be subject to tampering by external intervention, and makes it censorship resistant (as much as the Ethereum network itself is censorship resistant). However, it must be noted that the Liquity contracts rely on the Chainlink and Tellor oracles for retrieving the ETH price in USD, and the oracles themselves are not immutable (they are upgradeable).

With regards to Liquity’s utilization of price oracles, the following behaviours were observed:

1. By design Liquity has no way to upgrade oracles after deployment: if the oracles smart contracts stop being updated for any reason, there’s no way to change them.
2. There is no emergency mechanism, if no good value is found in both oracles, last known value is returned, operations do not stop or pause when both oracles are in a bad state:
   ```solidity
   if (_tellorIsBroken(tellorResponse)) {
     _changeStatus(Status.bothOraclesUntrusted);
     return lastGoodPrice;
   }
   ```
3. 4 hours need to pass without an update to consider Chainlink frozen in order for the price feed contract to switch from using Chainlink to using Tellor.
4. At least a 50% price difference between 2 consecutive Chainlink answers is needed to consider the new price suspicious and compare it with Tellor’s.
5. Tellor feed is never considered frozen, no matter how old the last answer is (there is a comment in the source code about Tellor needing “tipping”, this means Tellor workers decide which jobs to execute based on how big the current tip for this task is in comparison with other jobs waiting to be executed).
When the total collateralization ratio (TCR) falls below 150% the system enters Recovery Mode. During Recovery Mode, the liquidation threshold is raised from 110% up to the current TCR, and the system blocks borrower transactions that would further decrease the TCR. All operations going through the _adjustTrove function (e.g., withdrawals) are not permitted when the resulting TCR is less than 150%. This is intended to prevent operations that would put the system in Recovery Mode. However, it was found that the closeTrove function does not enforce this requirement, and this makes it possible to close troves and force the system into Recovery Mode without an ETH price swing (see CI-LQY-01).

Regarding Recovery Mode, it is worth noting that the owners of troves under 150% collateralization rate should be aware of the possibility of being liquidated if the system’s total collateralization rate drops below 150%. This means that the liquidation threshold does not only depend on the trove’s collateralization ratio itself, but moves based on the system’s total collateralization ratio. As the TCR gets closer to 150%, all those troves below the TCR are at risk of being liquidated: a big enough trove with a low ICR could be created right before a price drop in ETH to amplify the effect on the TCR and trigger Recovery Mode in order to liquidate all those positions below TCR. This scenario is considerably more probable in the current context of mining pools offering Miner Extractable Value services (Ethermine Adds Front-Running Software to Help Miners Offset EIP 1559 Revenue Losses). Coinspect recommends this fact is clearly documented and that the front-end should warn the end user about the system state and potential risks when opening a new trove with a collateralization ratio below 150%.

While analyzing the possibility to use flash loans to attack the protocol, it was found that attackers could inflate the system fees in order to break the LUSD hard peg mechanism. The attack has a lower cost during the first period after deployment, when the system is expected to have low total debt. However, this attack is only possible if there is enough LQTY circulating in proportion to the LQTY amount staked in the LQTY pool (see CI-LQY-03).

Liquity solvency heavily relies on quick and efficient liquidation of debt. In order to achieve this, two liquidation options are available: sequential liquidations and batch liquidations. Originally only sequential liquidations were allowed, and this involved liquidating undercollateralized troves in order starting with the trove with lowest ICR (i.e., the riskiest trove). However, if there are enough small troves with low ICR this mechanism could be too slow or expensive in terms of gas costs, and this reasoning led to the introduction of batch liquidations that allow the liquidation of any set of undercollateralized troves in any order. But the introduction of batch liquidations affected the incentives mechanisms in place, because enables liquidators to target undercollateralized troves with higher ICR if their liquidation is more profitable because of their size (see CI-LQY-04).

The liquidation criteria depends on different scenarios. For example: when Liquity is in Recovery Mode, troves with a collateralization ratio below the system total collateralization ratio and above the minimal collateralization ratio (110%) can be liquidated but only from the Stability Pool funds; if the funds available in the Stability Pool are not enough to completely offset a trove’s debt this trove will not be liquidated until the trove crosses the minimal collateralization ratio threshold. In this context, the best liquidation strategy (for liquidators and for the system) consists in borrowing and depositing enough LUSD in the
Stability Pool in order to liquidate the target troves and liquidating them in a single transaction for a zero risk profit. Coinspect suggests that a liquidator smart contract reference implementation is provided by Liquity to guarantee the interested parties are always able to quickly liquidate positions in an optimal way in order to protect the platform’s overall health.

Also, it was found that the `liquidateBatch` function neglects to check that the troves specified to be liquidated are actually active troves, although this is not currently a problem (see CI-LQY-02).

During the first days of the assessment the Liquity team asked Coinspect’s input about a new issue that arose when a trove is liquidated and its stakes are subtracted from the total stakes in the system but its collateral is not subtracted from the total collateral in the system. In some circumstances, as shown by some tests, this can lead to loss of precision and can end up with the total stakes going to zero. Coinspect reviewed the code and the mathematical proofs associated with the math behind the code. It was shown that when a trove is liquidated and the total stake is decreased without decreasing the total collateral, this produces an immediate decrease in the stakes of the remaining active troves which in turn decreases the total stakes and produces again a decrease in the stakes of all the troves, and this process could at first sight possibly go on ad infinitum. This motivated the Liquity team to write a new test to make sure that stakes decrease in increasingly smaller amounts and don’t go to zero.

Even though the front-end component was not in scope, Coinspect discussed with Liquity several improvements such as:

1. the utilization of EIP-712 to show the user the transaction being signed through the wallet interface in order to prevent malicious front-ends from, for example, keeping all the rewards for themselves by setting an arbitrary kickback parameter.
2. A signature verification mechanism so users know the front end operators did not tamper with the package reviewed or provided by Liquity.

After the engagement was finished, Liquity asked Coinspect to review minor modifications to the LQTY token issuance implemented in PR #435:

1. A new parameter was added to the LQTY token constructor. Liquity’s company allowance is now issued to the address provided in this parameter instead of being issued to the deployer address that is creating the token. This parameter is intended to be used with a multisig contract. The limitations regarding utilization of these funds still apply and were not modified by this pull request.
2. The initial LQTY token allocations were changed. Now, 2 million tokens are allocated to bounties and hackathons. Before, 1 million was being allocated for that purpose.
## 4. Summary of Findings

<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
<th>Risk</th>
<th>Fixed</th>
</tr>
</thead>
<tbody>
<tr>
<td>CI-LQY-01</td>
<td>TCR manipulation enables sudden liquidations</td>
<td>High</td>
<td>✔</td>
</tr>
<tr>
<td>CI-LQY-02</td>
<td>liquidateBatch does not verify processed troves are not closed</td>
<td>Low</td>
<td>✔</td>
</tr>
<tr>
<td>CI-LQY-03</td>
<td>Inflating fees facilitated in low LQTY pool participation scenario</td>
<td>Medium</td>
<td>✗</td>
</tr>
<tr>
<td>CI-LQY-04</td>
<td>Liquidations incentives misaligned with system overall health (TCR)</td>
<td>Medium</td>
<td>✗</td>
</tr>
</tbody>
</table>
5. Findings

**CI-LQY-01**  
TCR manipulation enables sudden liquidations

<table>
<thead>
<tr>
<th>Total Risk</th>
<th>Impact</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>High</td>
<td>High</td>
<td>BorrowerOperations.sol</td>
</tr>
<tr>
<td>Fixed</td>
<td>Medium</td>
<td></td>
</tr>
</tbody>
</table>

**Description**

Missing restrictions in `closeTrove` enable attackers to force the system into Recovery Mode without any ETH price swing. All operations going through the `_adjustTrove` function (e.g., withdrawals) are not permitted when the resulting TCR is less than 150%. This is intended to prevent operations that would put the system in Recovery Mode.

This is the relevant code in `_adjustTrove`:

```solidity
/*
 * When the adjustment withdraws collateral or increases debt, make sure it is a valid change for the trove's ICR and for the system TCR, given the current system mode.
 */
if (_collWithdrawal != 0 || _isDebtIncrease) {
    assert(_collWithdrawal <= vars.coll);
    uint newTCR = _getNewTCRFromTroveChange(vars.collChange, vars.isCollIncrease, vars.netDebtChange, _isDebtIncrease, vars.price);
    _requireValidNewICRandValidNewTCR(isRecoveryMode, vars.oldICR, vars.newICR, newTCR);
}
```

And this is `_requireValidNewICRandValidNewTCR`:

```solidity
function _requireValidNewICRandValidNewTCR(bool _isRecoveryMode, uint _oldICR, uint _newICR, uint _newTCR) internal /*pure*/ view {
    _requireICRisAboveMCR(_newICR);
    if (!_isRecoveryMode) {
        // When the system is in Normal Mode, check that this operation would not push the system into Recovery Mode
        _requireNewTCRisAboveCCR(_newTCR);
    } else {
        // When the system is in Recovery Mode, check that this operation would not worsen the trove's ICR (and by extension, would not worsen the TCR)
        _requireNewICRisAboveOldICR(_newICR, _oldICR);
    }
}
```
However, this requirement is not enforced in the closeTrove function. Then, it is possible to withdraw all funds from troves and force the system into Recovery Mode without any ETH price swing.

As a result, it is possible to suddenly close many troves with high ICR, triggering Recovery Mode, thus making all troves up to 150% ICR liquidatable. All those positions that were created with ICR above 110% but less than 150% when the system had a high TCR, could be targeted for liquidation now and can not close their troves nor withdraw funds until the system is back to normal mode.

This is how one way an attack would look like:

1. Create big troves in order to inflate TCR > 150%.
2. Wait for victims to create troves with ICR < 150% as the overall TCR permits it and encourages it.
3. Trigger recovery mode: close big troves so the resulting TCR is below 150%.
4. Victim troves can not be closed now (they must top up collateral).
5. Profit by liquidating the victim troves.
6. Can do steps 3, 4 and 5 in a single transaction so victim troves have no opportunity to top up collateral.

The attacker needs to have LUSD deposited in the Stability Pool in order to be able to liquidate the troves with ICR > 110%, as those troves are not liquidatable by redistribution.

The troves harmed in this scenario are those that trusted the system apparent overall collateralization ratio (and/or ETH price expectations) and created troves with ICR close to 150%. Those troves with individual collateralization ratio above 150% would not be affected.

In order to exploit this issue, attackers need enough capital to inflate the TCR. This is more likely during the system bootstrap stage. Another scenario where less capital would be required is if the TCR is already close to 150%.

Recommendation

Restrict the function closeTrove when the resulting TCR < 150% in a consistent way with other restrictions imposed in other functions.

Resolution

The Liquity team opened issue #368 to address this problem. This issue was resolved by PR #395. Coinspect verified the proposed fix is correct and solves the problem identified. Additionally, the existing conditions in _adjustTrove were further improved in PR #403.
**CI-LQY-02  liquidateBatch does not verify processed troves are not closed**

<table>
<thead>
<tr>
<th>Total Risk</th>
<th>Impact</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>Low</td>
<td>TroveManager.sol</td>
</tr>
<tr>
<td>Fixed ✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Likelihood</td>
<td>Low</td>
<td></td>
</tr>
</tbody>
</table>

**Description**

The `liquidateBatch` function in TroveManager.sol does not check if the parameters passed to the function are active troves. On the other hand, the `liquidate` function does check:

```solidity
function liquidate(address _borrower) external override {
    _requireTrovesisActive(_borrower);

    address[] memory borrowers = new address[](1);
    borrowers[0] = _borrower;
    batchLiquidateTroves(borrowers);
}
```

Down the road, `liquidateBatch` eventually ends up ignoring closed troves, because of their “infinite” CR as returned by LiquityMath.sol:

```solidity
function _computeCR(uint _coll, uint _debt, uint _price) internal pure
returns (uint) {
    if (_debt > 0) {
        uint newCollRatio = _coll.mul(_price).div(_debt);

        return newCollRatio;
    }
    // Return the maximal value for uint256 if the Trove has a debt of 0.
    // Represents "infinite" CR.
    else { // if (_debt == 0)
        return 2**256 - 1;
    }
}
```

This happens because the function `closeTrove` sets the trove’s debt to 0 besides setting the status to closed.

This is not an exploitable issue right now but:
1. The result is not as clear and explicitly documented as other parts of the source code.
2. Depends on many functions not changing in the future.
3. Could end reverting the whole batch if an action is attempted with the trove, which does check for closed troves itself, for example.
Recommendation

Consider explicitly verifying if each trove is active before processing it in a similar way as it is performed for non-batched liquidations. If the team decides to add this check, make sure closed troves are ignored instead of reverting the transaction, so the batch continues to be processed.

Resolution

This issue was resolved by PR #427. Coinspect verified the proposed fix is correct and solves the problem identified.
Description

In Liquity, a redemption is the process of exchanging LUSD for ETH at face value, as if 1 LUSD is exactly worth $1. That is, for x LUSD you get x Dollars worth of ETH in return. Users can redeem their LUSD for ETH at any time without limitations. However, a redemption fee might be charged on the redeemed amount. LUSD redemptions are enabled after 14 days have passed since system bootstrap.

The system fees for borrowing LUSD and for redeeming LUSD for ETH:
1. Are incremented each time a redemption is made (in order to progressively discourage them), proportionally to how much LUSD is being redeemed and the total LUSD debt in the system.
2. And decay as time passes with a 12 hours half life.

Borrowing fees are limited to 5% and redemption fees can grow up to 100% of the operation. Users can specify a maximum fee they are willing to pay in order to prevent slippage.

Attackers with the goal of inflating the fee rate are limited by the cost of their own operations while doing this, as each time they increment the base rate, the new rate is applied to their redeems.

But because these fees are distributed among the LQTY pool stakers, in the unlikely scenario the LQTY circulating supply represents a big proportion of the LQTY staked in the pool, the cost of moving the fees up could be instantaneously recouped by the attackers. The attackers can flash loan the circulating LQTY and stake it in order to recover the fees charged each time they redeem LUSD and the base rate increases.

More concretely, this is the sequence of operations the attackers would perform in one transaction:
1. Flash loan as much LQTY as possible.
2. Stake the LQTY in the LQTY pool.
3. Open a new trove with the lowest ICR above 110 in the system.
4. Redeem LUSD for ETH from his own trove, this step will increase the base rate.
5. Close his trove.
6. Unstake the LQTY in order to recoup the fees paid in step 4.
7. Payback the loaned LQTY.
As a result of using self-minted LUSD, the base rate would go up in each iteration, while the LUSD supply remains constant and its price is unaffected. And by using flash loans or flash swaps, there would be no exposure to LQTY nor LUSD.

The attack cost will depend on how much LQTY pool share the attackers are able to loan and how much is staked in the pool. Each time the attackers redeem LUSD, a part of this operation will be recovered and another part will be lost proportionally to the share of the LQTY pool owned.

Because the base rate increases in proportion to how much LUSD is redeemed and the total LUSD debt in the system, this attack would be cheaper when the system total debt is low, for example during the first months after deployment.

Though unlikely, there are a few potential scenarios that could result in a big proportion of the LQTY being not staked are:

1. Community issuance LQTY tokens being dumped by front-end operators and stability providers during the first months of the system.
2. LQTY distribution concentration in a few hands.
3. Lack of LQTY staking interest because of other more profitable alternatives. For example: high incentives to provide LQTY liquidity in an AMM resulting in more profit than staking in the LQTY pool because of low number of operations in Liquity.

Redemptions are responsible for keeping the LUSD hard peg floor at $1 USD. When the LUSD price is less than $1 USD arbitrageurs are incentivized to redeem LUSD, pushing the price back again to 1.

As a consequence of the elevated redemptions fees, the LUSD price floor would move further down while the base rate is maintained elevated. If this is maintained for an extended period of time, redemptions would become unprofitable and this could hurt users confidence in the protocol, during the first months of operations, as they would be unable to redeem their LUSD for ETH. Also, new loans would become unattractive.

**Recommendation**

In order to further bulletproof Liquity during its bootstrap months Coinspect recommends considering a faster base fee decay speed, at least during this initial period, in order to make this attack even more expensive for an attacker.
Liquidations incentives misaligned with system overall health (TCR)

<table>
<thead>
<tr>
<th>Total Risk</th>
<th>Impact</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Medium</td>
<td>High</td>
<td>TroveManager.sol</td>
</tr>
<tr>
<td>Fixed ✗</td>
<td>Medium</td>
<td></td>
</tr>
</tbody>
</table>

Description

Because of the low collateralization ratio required, Liquity solvency heavily relies on quick and efficient liquidation of debt. In order to achieve this, two liquidation options are available.

The original liquidation mechanism (still in place as an alternative), sequential liquidations, starts with the trove with lowest ICR (the riskiest trove) and liquidates as many troves as possible.

The newer mechanism, batch liquidations, was introduced to prevent an attack where many small troves with low ICR are created in order to prevent prompt liquidations from happening in order to damage system health as measured by the TCR during an ETH price drop scenario. According to Liquity’s documentation, the current implementation is able to liquidate up to 90-95 troves in one transaction with the current block gas limit of 12.5m.

Batch liquidations introduced a fairness issue, because it enables liquidators to target higher ICR troves. This was deemed as a necessary drawback to counter the riskier security issue described before, as explained in Liquity Releases Updated Whitepaper | by Robert Lauko | Liquity | Feb, 2021:

> Given that our system heavily relies on quick and efficient liquidation of debt, we need to offer an easy way to specifically liquate large Troves, making sure that a substantial portion of the debt can be cleared quickly. To that end, we decided to allow liquidations of arbitrary batches of Troves, expecting that liquidation bots will prioritize the larger ones thanks to the higher gas compensation (see below). The system can tolerate smaller Troves remaining unliquidated for a longer time, as it is not the number, but the total pending debt that matters in the end.

Liquidators are compensated with:

a. Fixed 50 LUSD per trove liquidated, plus
b. Variable 0.5% of each liquidated trove’s total collateral

```solidity
// Return the amount of ETH to be drawn from a trove's collateral and sent as gas compensation.
function _getCollGasCompensation(uint _entireColl) internal pure returns (uint) {
    return _entireColl / PERCENT_DIVISOR;
}
```

© 2021 Coinspect
Then, liquidators are incentivized to liquidate troves with more collateral. If these troves with more collateral are the ones with ICR close to the TCR (more collateral but less debt in proportion), the impact on the TCR is less than the impact from liquidating those troves with smaller ICR.

As a consequence, liquidators have more incentive to liquidate troves that have less impact on the overall health of the system as measured by the TCR.

During a fast ETH price drop, automated bots will liquidate those troves that generate more profit for them, but are not optimal for the system's health. Those lower ICR troves will remain unliquidated, and will eventually cross the 100% collateralization rate line where liquidating them results in a loss for other system participants.

Recommendation

Review the current liquidation incentives in the context above explained, and tweak them so they are better aligned with the overall system health.
6. Appendix

The following recommendations are aimed at improving the project's overall quality, readability, test coverage and documentation. Even though they represent no immediate risk, it is recommended to take them into consideration when possible.

(These suggestions have been addressed by PR #421 and PR #426.)

The tests don't cover most of the events emitted by the contracts. It is recommended to test that each event is emitted when expected, and that the parameters are set to the correct values. This is important because off-chain components such as the front-ends rely on the correctness of events.

Events not tested include: all *AddressChanged and *AddressSet events, TroveCreated, TroveLiquidated, CollBalanceUpdated, EtherSent, DefaultPoolLUSDBalanceUpdated, DefaultPoolETHBalanceUpdated, OwnershipTransferred, Transfer, ETHBalanceUpdated, LUSDBalanceUpdated, TotalLQTYIssuedUpdated, NodeAdded, NodeRemoved, StakeChanged, StakingGainsWithdrawn, F_ETHUpdated, F_LUSDUpdated, TotalLQTYStakedUpdated, StakerSnapshotsUpdated, ActivePoolLUSDBalanceUpdated, ActivePoolETHBalanceUpdated, LUSDBorrowingFeePaid, BaseRateUpdated, LastFeeOpTimeUpdated, TotalStakesUpdated, SystemSnapshotsUpdated, LTermUpdated, TroveSnapshotsUpdated, TroveIndexUpdated, LockupContractDeployedThroughFactory, StabilityPoolETHBalanceUpdated, StabilityPoolLUSDBalanceUpdated, P_Updated, S_Updated, G_Updated, EpochUpdated, ScaleUpdated, FrontEndRegistered, FrontEndTagSet, DepositSnapshotUpdated, FrontEndSnapshotUpdated, LQTYPaidToDepositor, LQTYPaidToFrontEnd, StakeChanged, LockupContractCreated, LockupContractEmptied.

There are some unnecessary uses of SafeMath functions that can be avoided to slightly reduce gas costs. For example, there is some room for optimization in LiquityMath. When dividing by something that is known to be nonzero it is unnecessary to use SafeMath's div and instead / can be used directly; this happens in function decMul when dividing by DECIMAL_PRECISION, and in functions _computeNominalCR and _computeCR when dividing by _debt. In function _getAbsoluteDifference it is unnecessary to use SafeMath's sub and - can be used directly. And in function _decPow, n.div(2) can be replaced with n>>1, and n.sub(1) can be replaced with n-1. In general in all contracts, all divisions by constants such as DECIMAL_PRECISION, BETA, SECONDS_IN_ONE_MINUTE, SCALE_FACTOR, etc, can be performed directly without SafeMath (this includes multiple cases in TroveManager, BorrowOperations and StabilityPool).

In CommunityIssuance.sol there is an unfinished TODO comment:

```solidity
//TODO: Decide upon and implement LQTY community issuance schedule.
contract CommunityIssuance is ICommunityIssuance, Ownable, CheckContract, BaseMath {

```

In LockupContractFactory.sol and LockupFactory.sol _beneficiary is never checked to be nonzero, and this could result in tokens locked forever.
In `LQTYStaking.sol` the event `EtherSent` should be emitted before the call in order to guarantee correct event ordering in case of reentrancy:

```solidity
defunction _sendETHGainToUser(uint ETHGain) internal {
    (bool success, ) = msg.sender.call(value: ETHGain)("");
    require(success, "LQTYStaking: Failed to send accumulated ETHGain");
    emit EtherSent(msg.sender, ETHGain);
}
```

Typos in `BorrowerOperations.sol`:

```solidity
// A doubly linked list of Troves, sorted by their sorted by their collateral ratios
[...]
"Max fee percentage must less than or equal to 100%"
```

The functions `_requireAtLeastMinNetDebt` in `BorrowerOperations.sol` and `_calcRedemptionFee` in `TroveManager.sol` can be further restricted to `pure` functions instead of only `view`.

The source file `ILiquityBase.sol` is missing an SPDX licence identifier.

Typos in `Unipool.sol`:

```solidity
* - Liquidity providers accrue rewards, proportional to the amount of staked tokens and staking time
```

Incorrect comment in `TroveManager.sol`, should read 50 instead of 10:

```solidity
* Called when a full redemption occurs, and closes the trove.
* The redeemer swaps (debt - 10) LUSD for (debt - 10) worth of ETH, so the 10 LUSD gas compensation left corresponds to the remaining debt.
* In order to close the trove, the 10 LUSD gas compensation is burned, and 10 debt is removed from the active pool.
```

Incorrect comment in `TroveManager.sol`, should read 2000 instead of 10:

```solidity
/* Max array size is 2**128 - 1, i.e. ~3e30 troves. No risk of overflow, since troves have minimum 10 LUSD
```

In `test/StabilityPoolTest.js`; `alice_snapsoht_Before` should be `alice_snapshot_After`:

```javascript
const alice_snapshot_After = await stabilityPool.depositSnapshots(alice)
const alice_snapshot_S_After = alice_snapshot_After[0].toString()
const alice_snapshot_P_After = alice_snapshot_After[1].toString()
const alice_snapshot_G_After = alice_snapshot_Before[2].toString()
```
7. Disclaimer

The information presented in this document is provided "as is" and without warranty. Source code reviews are a “point in time” analysis and as such it is possible that something in the code could have changed since the tasks reflected in this report were executed. This report should not be considered a perfect representation of the risks threatening the analyzed system.