Liquity

Smart Contract Audit

Smart Contract Audit

Liquity

Prepared for Liquity • March 2021

v210330

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:

High Risk	Medium Risk	Low Risk
1	2	1

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 liquity/dev `main` branch up to commit hash dd7f59b9:

```
commit dd7f59b980e7dab1cebc84c017db3a2c4caa522c
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:

```
fbe0e6a4f1e6392bdfde3fba7df869821666aefcbc3d3137fffef4b4308a672c  HintHelpers.sol
05c843751679ff0189a7ed5182aa743d7bd9360934d3dbca54571e9eac0327c7  ActivePool.sol
c3aca0fbbfb62c00d376feacf5af41f95654a9a43aee837b9a164a0ff03bd2f2  PriceFeed.sol
9aab938a8b7985e223e5e0d13bbd720d2a0e365706dd789668968d67ed8e9581  GasPool.sol
9468eae6f8518dee798083af24817c26ed3c742494095b65e63b95a984c7ac97  StabilityPool.sol
2ca1ccd4de76bad7c1bbae1969df879574720dfb2e83e46a6f0546ee39055570  BorrowerOperations.sol
ec68a3021cc91bc53803ea26325d71645bafffce7e11ce23baaae485e0b4453e  LQTY/LQTYStaking.sol
557f141e6dc650da5412966772168abb74e324c7bf47ac308930de7c09063425  LQTY/LockupContract.sol
ac5b84db980ac9571cb2442889beffc80ba9c61568522cf4aaa9c1c0c30c3ee9  LQTY/CommunityIssuance.sol
233e30ac84bc80b76db368716480f1c815834f3a0280bc92e5b7c794803376e2  LQTY/LQTYToken.sol
94fc9bd46be3ea8e2648843b7cbfd23a2241b7f9e5d82865ecee34065e9c3761  LQTY/LockupContractFactory.sol
a5616665b77782ca6ee3b2c0cb8505600f7a47ec4f66f915bda1b4fd60e543d1  TroveManager.sol
aa8dfdbc5cc6f3c25cd4e49dcb9737ac422c3ddea2b55228a4e2f675c5d25807  Proxy/StabilityPoolScript.sol
c3d215577b7c9845fc80e62d0393f36388cb73d05e12264d4a9dfe9a3129cdfa  Proxy/TroveManagerScript.sol
66f67a36c2e1c130220cf895a6294d93097be536c68d31742ba644b12c952ceb  Proxy/BorrowerOperationsScript.sol
b914f73e833ba81e98449d88e71ffee1588ef6ec3fde5ffa90fc8ff15dc935fb  Proxy/LQTYStakingScript.sol
70f21d80f2e8d777f9244beff40d0f9a4c277224c5f507092ddbb9a71ed3e62a  Proxy/ETHTransferScript.sol
c95314a512c6bfd75bc551f5c5c372102cb13d7bb1fd5dfd2f05990afd720fbb  Proxy/BorrowerWrappersScript.sol
17429b3ad70b9a424e8f38a579527463cfd1861313daeeca8cd1471b93e97be8  Proxy/TokenScript.sol
7e7c6a8d9f4dc43a6f02b1de70175a5964f7ef6e835492d426f8aa1854e20358  Interfaces/ILockupContractFactory.sol
a96e534c2e6db810aedf53af759702410ba19b9be7169709a73c5ad4695f7102  Interfaces/IBorrowerOperations.sol
1b25e623b3db2a2d18dd30d347823e8640f601747a3632fb941d9bcdf84fd898  Interfaces/ICommunityIssuance.sol
0976c8be61fa1cc23a3883dedbcc40a8132ad7a8a45b9e865aa1e4f8a1437b7a  Interfaces/ITroveManager.sol
3e743e3da65e5a3333140807fee464bd32655b6bf945436c1fa78709ae9d9a63  Interfaces/ITellorCaller.sol
b30789ec4ee77a4bd502aee8f1ca7b3368fc9250eac83ab4fbd737fdd094d2c1  Interfaces/IPriceFeed.sol
a00fad92e67bae55bde265e4fa6ab3ebed7f1db5b55c9e2f85ef28317d40f28b  Interfaces/ILQTYStaking.sol
30ce23ad28599dbf8c1273ff7d622a804297194fb0c3be89da25548f0b7c67a1  Interfaces/ILUSDToken.sol
37ab65ec774a51a0d5739e9dac82bfe85f9804109a91a26345d49ae61a5b50bb  Interfaces/IActivePool.sol
1756c28b2f3e6c8cc2a76cb041311862fe6471cc4f0aa09c46f76be16d714a6e  Interfaces/ISortedTroves.sol
30fdd16e7aa47d77d8174391302da8ada43c9e77951eb6dee463914f693ff781  Interfaces/IDefaultPool.sol
f2ae8a3cd138028fdba9d310ff79c00e1b1a843ef0fa1ebc272726172c7dbde0  Interfaces/IStabilityPool.sol
0de2b49acec5c18204a493d9bef11072a767111402928e62a0fe76dd569f3c09  Interfaces/ICollSurplusPool.sol
eafec838102636d0d8cbddc817cdf201ed7d6bd4a981d9edf6b54a1c2eb6ba7c  Interfaces/IPool.sol
6d54ecced315fda5a33ddeaf729f178fd55cb5d0990fa7157272fdf1efa2b9df  Interfaces/ILQTYToken.sol
9032ed2106a459e989145772ef21b17b9fa940d6443add7d9639b9aa39de031f  Interfaces/ILiquityBase.sol
49e0dd1154d00683c468515fa53184efa5ed29f6c26c4af90ee7d565366b935d  MultiTroveGetter.sol
645208d3053f1ee614b73776e9c638f4529062bfc3333fa06ba5663d9193405b  Dependencies/Ownable.sol
bf71564fc13f4494479f0e6f0fbe2e55d8d8ac9406795841b5d4d8e670a10681  Dependencies/TellorCaller.sol
caa5397440fd9a0988eb40c136bd7a58baad05012edcf244f6b586e167e531f6  Dependencies/SafeMath.sol
fe7de02fbe78bf1af499331c9a5a404299a7141f0800e942e29b55c8c64029dc  Dependencies/console.sol
93aa3e470a6e581ea7e515d2e34737f5e619af229637054ac09f0d303c0bddd2  Dependencies/LiquitySafeMath128.sol
83a6ac8c0f185342c500ada926f0dbe734a959382bbcbe8e1c69bcf7faa2454b  Dependencies/IERC2612.sol
0d418289d7ba0ab052d1fff29b3c832bd809175d9bd69b8cd5552184a5e1bf17  Dependencies/LiquityMath.sol
```

```
0627bd40d58674014ea1b2f8f2643a22fb10b609671a0f0bae6fc1e4f51e7bf8  Dependencies/ITellor.sol
aa32079b9f38a1669beb9fefe2e0af95fc543e1b717617713f00a648f0dc4b66  Dependencies/CheckContract.sol
64f3e9f771f7ba660ba11cf966318da692834288126f5b58601d8ba3ffc1a3fa  Dependencies/IERC20.sol
9fb35c0f0d7df453a61dee9b4f8e7e16cf66fcea4c3031f3ba78b202dab6eb32  Dependencies/LiquityBase.sol
b2ebef44df63d7e32405e0eb398a5868123c5f1800005c02d44d53cec38a28df  Dependencies/AggregatorV3Interface.sol
fada1c3c95dcfa780a6d03bab5e1f329201a30ec94140b54818559f9e720cbc4  Dependencies/BaseMath.sol
d51c34e6b5b779da4ec2016fac2261d93432b8ea67cf76d31fbf677acb659969  LUSDToken.sol
8ccccd7b7cda827b7a839a8dae6fac75c85e4e8431d2ce8624470ae303aa281e  Migrations.sol
f96053d377ae700609dcd14da609738782c6b48c81f35665ec0d758f11ae12e1  CollSurplusPool.sol
dd41a2af476319a1843abcef17dc52b3c6715c05329772874910b7f9041d1106  SortedTroves.sol
5c9d8089247f5861301f26a378f70744fa710b43703fd3e63cbab81fe4a7e553  LPRewards/TestContracts/ERC20Mock.sol
a0a344fec8abc86cfd17606b31d333fc9fd45d038faed69ef78b60eebf6d0d0e  LPRewards/Interfaces/IUnipool.sol
bcfedabf6b5ae1487f11d40856510c1464068c720bb4ff42c22f6ea0bf311b6a  LPRewards/Interfaces/ILPTokenWrapper.sol
3681852ced35083475b9e0ce52a421ebe92168c5e9c3a7582888db8e6233a3a7  LPRewards/Dependencies/SafeERC20.sol
7a6f9acee77ad6aa4db616c473d507a5726698c3bd9005d6c54d8258c3cb6480  LPRewards/Dependencies/Address.sol
86ac8233f2b2b4b243f64becf9747d2a3eb0bef4ca53c79011f0b1e8a4dbb1a0  LPRewards/Unipool.sol
70594a39dc93cf8781fade93e4cc7962d086748788596fe2a824c47a446e65df  DefaultPool.sol
```

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 a LQTY token that is used to distribute among its holders a share of the revenue generated from redemption fees and LUSD issuance fees. LQTY tokens are 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:
```
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

ID	Description	Risk	Fixed
CI-LQY-01	TCR manipulation enables sudden liquidations	High	✔
CI-LQY-02	liquidateBatch does not verify processed troves are not closed	Low	✔
CI-LQY-03	Inflating fees facilitated in low LQTY pool participation scenario	Medium	✘
CI-LQY-04	Liquidations incentives misaligned with system overall health (TCR)	Medium	✘

5. Findings

CI-LQY-01	TCR manipulation enables sudden liquidations	
Total Risk **High**	Impact High	Location BorrowerOperations.sol
Fixed ✔	Likelihood Medium	

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`:

```
/*
 * 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`:

```
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

Total Risk	Impact	Location
Low	Low	TroveManager.sol
Fixed ✔	Likelihood: Low	

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:

```
function liquidate(address _borrower) external override {
    _requireTroveisActive(_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`:

```
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.

CI-LQY-03 Inflating fees facilitated in low LQTY pool participation scenario

Total Risk	Impact	Location
Medium	High	TroveManager.sol
Fixed ✗	Likelihood Low	

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.

CI-LQY-04 Liquidations incentives misaligned with system overall health (TCR)

Total Risk	Impact	Location
Medium	High	TroveManager.sol
Fixed ✗	Likelihood Medium	

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 liquidate 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

```
// 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;
}
```

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, DefaultPoolLUSDDebtUpdated, DefaultPoolETHBalanceUpdated, OwnershipTransfered, Transfer, ETHBalanceUpdated, LUSDBalanceUpdated, TotalLQTYIssuedUpdated, NodeAdded, NodeRemoved, StakeChanged, StakingGainsWithdrawn, F_ETHUpdated, F_LUSDUpdated, TotalLQTYStakedUpdated, StakerSnapshotsUpdated, ActivePoolLUSDDebtUpdated, ActivePoolETHBalanceUpdated, LUSDBorrowingFeePaid, BaseRateUpdated, LastFeeOpTimeUpdated, TotalStakesUpdated, SystemSnapshotsUpdated, LTermUpdated, TroveSnapshotsUpdated, TroveIndexUpdated, LockupContractDeployedThroughFactory, StabilityPoolETHBalanceUpdated, StabilityPoolLUSDBalanceUpdated, P_Updated, S_Updated, G_Updated, EpochUpdated, ScaleUpdated, FrontEndRegistered, FrontEndTagSet, DepositSnapshotUpdated, FrontEndSnapshotUpdated, UserDepositChanged, FrontEndStakeChanged, 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:

```
//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:

```
function _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`:

```
// 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`:

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

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

```
* 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:

```
/* 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`*:

```
// Check 'After' snapshots
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.