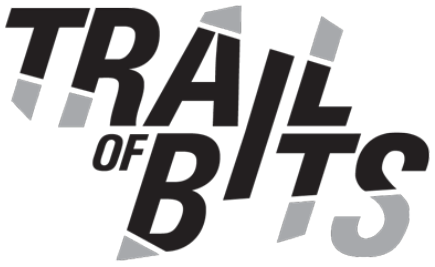

Liquity

Security Assessment

January 13, 2021

Prepared For:
Robert Lauko | *Liquity*
robert@liquity.org

Rick Pardoe | *Liquity*
rick@liquity.org

Prepared By:
Alexander Remie | *Trail of Bits*
alexander.remie@trailofbits.com

Gustavo Grieco | *Trail of Bits*
gustavo.grieco@trailofbits.com

Maximilian Krüger | *Trail of Bits*
max.kruger@trailofbits.com

Michael Colburn | *Trail of Bits*
michael.colburn@trailofbits.com

Changelog:
December 18, 2020: Initial report delivered
January 13, 2021: Added Appendix G. Fix Log

Executive Summary

From November 31 through December 18 2020, Liquity engaged Trail of Bits to review the security of their LUSD stablecoin. Trail of Bits conducted this assessment over the course of 8 person-weeks with 4 engineers working from commit `f0df3ef` of the `liquity/dev` repository.

During the first week, we focused on gaining an understanding of the codebase and started to review the two ERC20 tokens (LUSD and LQTY) and the trove and borrower operations against the most common Solidity flaws. During the second week, we focused on reviewing the smart contracts that perform trove and borrower operations to look for flaws that would allow an attacker to bypass access controls, disrupt or abuse the contracts in unexpected ways. The final week was dedicated to the review of the system properties as well as the more complex interactions between the troves and the pools.

Our review resulted in 14 findings ranging from high to informational in severity, with two important sources of issues. The most important category was missing or incorrect input validation, which produced the most severe issues, including arbitrary ERC20 restrictions when transferring (TOB-LQT-001), an improper handling of troves (TOB-LQT-013), unexpected failure when redeeming collateral for the first time (TOB-LQT-014).

The second category were issues related to source code maintenance or patching, which produced issues that could impact future code modifications or third-party interaction with them. These issues include duplicated code that could diverge in future revisions (TOB-LQT-011, TOB-LQT-012), lack of usage of immutable state variables (TOB-LQT-004).

In addition to the security findings, we discuss code quality issues not related to any particular vulnerability in Appendix C. Appendix D discusses several issues the Liquity team identified prior to this assessment and requested guidance on. Appendix E and Appendix F describe Slither scripts produced by Trail of Bits to facilitate our review.

Overall, the code follows a high-quality software development standard and best practices. It has a suitable architecture and is properly documented. The interactions between components are well-defined, and the testing is very extensive.

Trail of Bits recommends addressing the findings presented and expanding the property-based testing with more system invariants. Finally, we recommend performing an economic assessment to make sure the monetary incentives are properly designed.

Update: During the week of January 11, 2021, Trail of Bits reviewed fixes by Liquity for the issues presented in this report. See a detailed review of the current status of each issue in Appendix G.

Project Dashboard

Application Summary

Name	Liquity
Version	`f0df3ef`
Type	Smart Contracts
Platforms	Solidity

Engagement Summary

Dates	November 30 to December 18, 2020
Method	Whitebox
Consultants Engaged	4
Level of Effort	8 person-weeks

Vulnerability Summary

Total High-Severity Issues	1	■
Total Medium-Severity Issues	2	■ ■
Total Low-Severity Issues	8	■ ■ ■ ■ ■ ■ ■
Total Informational-Severity Issues	4	■ ■ ■
Total Undetermined-Severity Issues	0	
Total	14	

Category Breakdown

Access Controls	1	■
Auditing and Logging	1	■
Data Validation	5	■ ■ ■ ■ ■
Patching	4	■ ■ ■
Timing	1	■
Undefined Behavior	3	■ ■ ■
Total	14	

Code Maturity Evaluation

In the table below, we review the maturity of the codebase and the likelihood of future issues. In each area of control, we rate the maturity from strong to weak, or missing, and give a brief explanation of our reasoning.

Category Name	Description
Access Controls	**Strong.** Appropriate access controls were in place for performing privileged operations by certain contracts.
Arithmetic	**Moderate.** The contracts included use of safe arithmetics. No potential overflows were possible in areas where these functions were not used. However, certain arithmetic operations in corner cases could block the trove operations (TOB-LQT-013, TOB-LQT-014)
Assembly Use	**Strong.** The contracts only used assembly to fetch the `chainID` for ERC2612 `permit` functionality.
Centralization	**Strong.** While the protocol relied on an owner to correctly deploy the initial contracts, the ownership is renounced immediately after it. There are no other privileged accounts.
Contract Upgradeability	**Not Applicable.** The contracts contained no upgradeability mechanisms.
Function Composition	**Satisfactory.** While most of the functions and contracts were organized and scoped appropriately, the borrower operations contract code was difficult to review and should be refactored (Appendix C).
Front-Running	**Moderate.** Some functionality is affected by front-running attacks, with a moderate impact (Appendix D).
Monitoring	**Satisfactory.** The events produced by the smart contract code were sufficient to monitor on-chain activity.
Specification	**Moderate.** White papers describing the functionality of the protocol were available. However, some of the internal documentation regarding system properties was outdated and incomplete.
Testing & Verification	**Strong.** The contracts included a very large number of unit tests and even the use of automatic tools such as fuzzers.

Engagement Goals

The engagement was scoped to provide a security assessment of Liquity protocol smart contracts in the `liquity/dev` repository.

Specifically, we sought to answer the following questions:

- Are appropriate access controls set for system components?
- Is it possible to manipulate the system by front-running transactions?
- Is it possible for participants to steal or lose tokens?
- Are there any instances where arithmetic overflow may affect the system?
- Can participants perform denial-of-service or phishing attacks against any of the system components?
- Can flash loans negatively affect the system's operation?
- What impact would flash crashes or other oracle issues have on the system's stability?
- Does arithmetic regarding liquidations and pool bookkeeping hold?

Coverage

The engagement was focused on the following components:

- **BorrowerOperations:** This contract was an important entrypoint for managing the lifecycle of a trove as a borrower. We reviewed this contract to ensure appropriate balance tracking and that trove state changes were recorded properly, especially when transitioning between normal and recovery mode.
- **TroveManager:** The `TroveManager` contract exposed trove management functionality, including liquidation. We reviewed this contract to ensure liquidations were carried out properly, especially in the event of a state transition from normal to recovery mode or in the event of a partial liquidation operation.
- **Pool contracts:** The Liquity system was composed of several pool contracts that track amounts such as debt, rewards and staking. We reviewed the soundness of the arithmetic, the accuracy of bookkeeping operations, and as well as access controls that ensured functionality could only be invoked by the appropriate core contracts.
- **LUSD and LQTY tokens:** The Liquity system incorporated two separate tokens: LUSD, a stablecoin, and LQTY, which was used for staking and earning rewards. These contracts were implemented as standard ERC20 tokens with some extra functionality. We verified that all the expected token properties were correctly implemented and that the additional functionality did not introduce any security concerns.
- **Whitepaper:** The Liquity components were detailed in the project's whitepaper and repository README. We reviewed the documents to make sure it was consistent and it did not leave any important details unspecified.
- **Access controls.** Many parts of the system exposed privileged functionality that should only be invoked by other system contracts. We reviewed these functions to ensure they could only be triggered by the intended components and that they do not contain unnecessary privileges that may be abused.
- **Arithmetic.** We reviewed calculations for logical consistency, as well as rounding issues and scenarios where reverts due to overflow may negatively impact use of the protocol.

Off-chain code components were outside the scope of this assessment.

Automated Testing and Verification

Trail of Bits used automated testing techniques to enhance coverage of certain areas of the contracts, including:

- Slither, a Solidity static analysis framework. Slither can statically verify algebraic relationships between Solidity variables. We used Slither to detect invalid or inconsistent usage of the contracts' APIs across the entire codebase. Additionally we wrote custom Scripts that leverage Slither to find all functions where Ether is received, all call sites where Ether is sent and all uses of small integers. Those locations were then inspected manually. See Appendix E and Appendix F for details.
- Echidna, a smart contract fuzzer. Echidna can rapidly test security properties via malicious, coverage-guided test case generation. We used Echidna to test the expected system properties, including borrower operations, trove manager, pools and its dependencies.

Automated testing techniques augment our manual security review but do not replace it. Each technique has limitations: Slither may identify security properties that fail to hold when Solidity is compiled to EVM bytecode; Echidna may not randomly generate an edge case that violates a property. To mitigate these risks, we had extended fuzzing campaigns of 36 hours with Echidna and then manually reviewed all the results.

Automated Testing with Echidna

We managed to collect and implement more than fifty Echidna properties. These can be broadly divided in two categories: general properties of the contracts based on unit tests and ERC20 properties.

General Properties

Property	Result
There is always at least one trove if the system is initialized	PASSED
The ETH balances of the pools are consistent	PASSED
The ETH balances of the tokens and sorted trove contracts are always zero	PASSED
Total of stakes is strictly positive if there is at least one trove	PASSED
Total of stakes is strictly positive if there is at least one trove	PASSED

Total of stakes does not exceed the sum of the active pool and the default pool LUSD debt	PASSED
LUSD token balance of any user does not exceed the sum of the active pool and the default pool LUSD debts	PASSED
LUSD token total supply is equal to the sum of the active pool and the default pool LUSD debts	PASSED
LUSD token supply is greater than the sum of stability pool debt, troves stake, gas pool debt and Liquity tokens	PASSED
All the troves in the sorted list are active	PASSED
All the troves in the sorted list have at least the minimal amount of collateral	PASSED
All the troves in the sorted list have strictly positive stoke values.	PASSED
All the troves in the sorted list are decreasingly sorted by the the current ICR	PASSED
Opening a trove with less collateral than the specified by CCR will fail	PASSED
Opening a trove never fails if their preconditions are met	PASSED
Opening a trove adds it in the list of sorted troves	PASSED
Opening a trove results in a new active trove	PASSED
Opening a trove will mint the correct amount of tokens in the owner account	PASSED
Opening a trove sets the correct amount of collateral of the new trove	PASSED
Adding collateral fails if the caller does not have an active trove	PASSED
Adding collateral never fails if their preconditions are met	**FAILED (TOB-LQT-013)**
After adding collateral, all the pending rewards are liquidated (if any)	PASSED
After adding collateral, the amount of collateral from the trove of the caller is increased	PASSED

Repaying LUSD fails if the caller does not have enough tokens	PASSED
Repaying LUSD never fails if their preconditions are met	PASSED
After repaying LUSD, the amount LUSD of the caller is reduced	PASSED
After repaying LUSD, all the pending rewards are liquidated (if any)	PASSED
Withdrawing LUSD fails in recovery mode.	PASSED
Withdrawing LUSD never fails if their preconditions are met	PASSED
After withdrawing LUSD, the caller LUSD balance is updated with the corresponding amount	PASSED
After withdrawing LUSD, all the pending rewards are liquidated (if any)	PASSED
Closing a trove fails in recovery mode	PASSED
Closing a trove never fails if their preconditions are met	**FAILED (TOB-LQT-009)**
After closing a trove, its debt and stake is zero	PASSED
Closing a trove removes it from the list of sorted ones	PASSED
Closing a trove changes its state to closed	PASSED
After closing a trove, all the pending rewards are liquidated (if any)	PASSED
Troves liquidation, either individually or in batch, never fails if their preconditions are met	**FAILED (TOB-LQT-013)**
If the trove is closed after its liquidation, the same preconditions should be met than closing that trove	PASSED
Liquidate the most undercollateralized troves never fails	**FAILED (TOB-LQT-013)**
Provide liquidity to the stability pool fails if the caller is registered as frontend	PASSED
Provide liquidity to the stability pool fails if the caller does not have enough tokens	PASSED

Provide liquidity to the stability pool never fails if their preconditions are met	PASSED
Withdrawing liquidity from the stability pool fails if the caller does not have any token deposited	PASSED
Withdrawing liquidity from the stability pool fails in recovery mode	PASSED
Withdrawing liquidity from the stability pool never fails if their preconditions are met	PASSED
Withdrawing liquidity from the stability pool reduces the amount of deposited tokens	PASSED
After withdrawing liquidity from the stability pool, if the balance of the pool was zero, then `epochToScaleToG(0, 0)` is strictly positive.	PASSED
Withdrawing ETH from the stability pool reduces the amount of deposited collateral	PASSED
Withdrawing ETH from the stability pool never fails if their preconditions are met	PASSED
Withdrawing collateral from troves fails if it is not active	PASSED
Withdrawing collateral from troves fails if caller does not have enough collateral in their trove	PASSED
Withdrawing collateral from troves fails in recovery mode	PASSED
Redeem collateral should never fail if it is provided with the recommended parameters and the call has enough gas	**FAILED (TOB-LQT-014)**

ERC20 Properties

Property	Result
Transferring tokens to the null address (`0x0`) causes a revert.	PASSED
The null address (`0x0`) owns no tokens.	PASSED
Transferring a valid amount of tokens to a non-null address reduces the current balance.	**FAILED (TOB-LQT-001)**
Transferring an invalid amount of tokens to a non-null address reverts or returns false.	PASSED

Self-transferring a valid amount of tokens keeps the current balance constant.	PASSED
Approving overwrites the previous allowance value.	PASSED
The balances are consistent with the `totalSupply`.	PASSED

Recommendations Summary

This section aggregates all the recommendations made during the engagement. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals.

Short term

❑ **Consider removing the transfer constraints to avoid breaking important ERC20 invariants.** This will avoid confusing users and third-party developers as to how to use the Liquity tokens. TOB-LQT-001

❑ **Add `_requireValidRecipient(account)` before any state changing operations in `_mint`.** This will prevent minting to invalid recipients addresses for any future code that calls `_mint`. TOB-LQT-003

❑ **Add `immutable` keyword to state variables `deployer`, `deploymentStartTime` and `lockupContractFactory`.** This way future assignments to these variables will result in compiler errors. TOB-LQT-004

❑ **Consider refactoring the code to avoid using the gas pool account.** Instead, use a state variable that keeps track of the amount of LUSD tokens available to compensate gas costs. This will remove the dependency on an arbitrary address that no one should control. TOB-LQT-005

❑ **Apply the checks-effects-interactions pattern and move the event emissions above the call to `_moveTokensAndETHfromAdjustment` to avoid the potential reentrancy.** This will mitigate this reentrancy attack. TOB-LQT-006

❑ **Add some basic input validation to all configuration functions.** For example, check for the `0x0` address as this is the default value for an uninitialized address. This will mitigate the possibility of an incorrect deployment. TOB-LQT-007

❑ **Properly document the possibility of griefing `permit` calls to warn users interacting with Liquity tokens.** This will allow users to anticipate this possibility and develop alternate workflows in case they are targeted by it. TOB-LQT-008

❑ **Properly document and test the requirement of holding enough tokens to close a trove. Also, add a suitable revert message**. This will make sure this important feature works as expected. TOB-LQT-009

❑ **Remove the boolean return from the `lockContract` functions, and remove the success check from the factory contract.** This will remove this unreachable error. TOB-LQT-010

❑ **Consolidate a single implementation of `getTCR/checkRecoveryMode` function in the entire system.** This will avoid any future issue when the code is modified. TOB-LQT-011

❑ **Consolidate a single implementation for `getEntireSystemColl()` and `getEntireSystemDebt()` across the entire codebase.** This will avoid any future issue when the code is modified. TOB-LQT-012

❑ **Avoid any trove to be (re-)inserted using ICB = 0.** This will avoid breaking important system invariants. TOB-LQT-013

❑ **Implement a check, with a proper revert message, that fails the transaction if there are no redeemable troves.** This will avoid any unexpected reverts. TOB-LQT-014, TOB-LQT-015

Long term

❑ **Use Manticore or Echidna to make sure ERC20 properties hold.** This will detect issues during the development process**.** TOB-LQT-001

❑ **Make sure every system property is carefully specified and use Echidna or Manticore to test it.** This will detect issues during the development process TOB-LQT-003, TOB-LQT-009, TOB-LQT-013, TOB-LQT-014, TOB-LQT-015.

❑ **Review every state variable to make sure they have the correct type constraints.** Type constraints enforced by the compiler offer an additional layer to enforce important system properties. TOB-LQT-004

❑ **Avoid hard-coding any account address.** Using hard-coded addresses can be a trust issue that can be avoided refactoring the code to use a state variable. TOB-LQT-005

❑ **Use Slither to detect potential reentrant function calls.** TOB-LQT-006

❑ **Always perform some type of input validation.** Use Slither to detect functions that would benefit from a `0x0` address check. TOB-LQT-007

❑ **Carefully monitor the blockchain to detect front-running attempts.** TOB-LQT-008

❑ **Make sure all paths through the code are reachable.** Unreachable code is useless and makes security assessments more difficult than needed**.** TOB-LQT-010

❑ **Avoid duplicating logic.** Duplicated code could be modified and diverge in the future, introducing a security or correctness issue**.** TOB-LQT-011, TOB-LQT-012

Findings Summary

#	Title	Type	Severity
1	ERC20 transfer restriction can disturb 3rd-party interactions	Data Validation	Medium
2	False positive: removed after discussion with Liquity team		
3	No restrictions on minting to invalid recipients	Data Validation	Informational
4	Insufficient enforcement of immutability of system logic variables through the Solidity compiler	Patching	Informational
5	Unclear if the gas pool account is controlled by someone	Access Controls	Low
6	Reentrancy could lead to incorrect order of emitted events	Undefined Behavior	Low
7	Missing address validation when configuring contracts	Data Validation	Informational
8	Permit opens the door for griefing contracts that interact with the Liquity Protocol	Timing	Informational
9	Closing troves requires owners to hold the full amount of LUSD minted	Data Validation	Low
10	Failed lockup error is never thrown	Auditing and Logging	Low
11	Duplicated functions pose future risk of diverging	Patching	Low
12	Total system collateral and debt calculations are repeated multiple times	Patching	Low

	risking divergence and decreasing readability		
13	Troves can be improperly removed under certain circumstances which results in unexpected reverts	Data Validation	High
14	Initial redeem can unexpectedly revert if no redeemable troves	Undefined Behavior	Low
15	Redeem without redemptions might still return success	Undefined Behavior	Medium

1. ERC20 transfer restriction can disturb 3rd-party interactions

Severity: Medium
Type: Data Validation
Target: `LUSDToken.sol, LQTYToken.sol`

Difficulty: Medium
Finding ID: TOB-LQT-001

Description
Certain transfer restrictions imposed in the Liquity ERC20 tokens can produce unexpected results in 3rd-party components interacting with the system.

The Liquity ERC20 contracts forbid any transfer to a special set of addresses specified in the `_requireValidRecipient` function:

```
   function _requireValidRecipient(address _recipient) internal view {
       require(
           _recipient != address(0) &&
           _recipient != address(this),
           "LUSD: Cannot transfer tokens directly to the LUSD token contract or the zero
address"
       );
       require(
           _recipient != stabilityPoolAddress &&
           _recipient != troveManagerAddress &&
           _recipient != borrowerOperationsAddress,
           "LUSD: Cannot transfer tokens directly to the StabilityPool, TroveManager or
BorrowerOps"
       );
   }
```

Figure 1.1: `_requireValidRecipient` *function in* `LUSDToken.sol`.

This restriction was placed to enforce certain invariants as well protect the user from sending their token to a contract that cannot receive them. However, this can also produce unexpected results if any 3rd-party contracts interact with the Liquity ERC20 tokens.

Exploit Scenario
Alice writes a smart contract interacting with the Liquity tokens. Eve uses Alice's contract to produce a token transfer to a forbidden address. Then, Alice's contract unexpectedly reverts, blocking the rest of the interactions.

Recommendation
Short term, consider removing the transfer constraints to avoid breaking important ERC20 invariants.

Long term, use Manticore or Echidna to make sure ERC20 properties hold.

3. No restrictions on minting to invalid recipients

Severity: Informational
Type: Data Validation
Target: `LUSDToken.sol`

Difficulty: High
Finding ID: TOB-LQT-003

Description
Certain transfer restrictions imposed in the LUSD ERC20 token are not properly handled during the minting.

The Liquity ERC20 contracts forbids any transfer to a special set of addresses specified in the _requireValidRecipient function:

```
    function _requireValidRecipient(address _recipient) internal view {
        require(
            _recipient != address(0) &&
            _recipient != address(this),
            "LUSD: Cannot transfer tokens directly to the LUSD token contract or the zero
address"
        );
        require(
            _recipient != stabilityPoolAddress &&
            _recipient != troveManagerAddress &&
            _recipient != borrowerOperationsAddress,
            "LUSD: Cannot transfer tokens directly to the StabilityPool, TroveManager or
BorrowerOps"
        );
    }
```

Figure 3.1: `_requireValidRecipient` *in* `LUSDToken.sol`.

However, this restriction doesn't exist for `_mint`. This could lead to the minting of tokens to invalid recipient addresses.

```
    function _mint(address account, uint256 amount) internal {
        require(account != address(0), "ERC20: mint to the zero address");


        _totalSupply = _totalSupply.add(amount);
        _balances[account] = _balances[account].add(amount);
        emit Transfer(address(0), account, amount);
    }
```

Figure 3.2: `_mint` *in* `LUSDToken.sol`.

Exploit Scenario
Alice adds a new feature to the Liquity protocol code. Adding the feature results in code flows where invalid recipient addresses can be passed into `_mint`. Then, tokens can be minted to an invalid recipient address.

Recommendation
Short term, add `_requireValidRecipient(account)` before any state changing operations in `_mint`. This will prevent minting to invalid recipients addresses for any future code that calls `_mint`.

Long term, use Manticore or Echidna to make sure important system properties hold.

4. Insufficient enforcement of immutability of system logic variables through the Solidity compiler

Severity: Informational
Type: Patching
Target: `LQTYToken.sol`

Difficulty: High
Finding ID: TOB-LQT-004

Description
The system logic of the Liquity Protocol is supposed to be immutable after deployment. For this reason variables are marked immutable so they can only be assigned once in the constructor:

```
address public immutable communityIssuanceAddress;
address public immutable lqtyStakingAddress;
```

Figure 4.1: immutable system logic state variables in `LQTYToken.sol`.

The state variables `deployer`, `deploymentStartTime` and `lockupContractFactory` are part of the system logic but are mutable.

```
uint public deploymentStartTime;
address public deployer;

ILockupContractFactory public lockupContractFactory;
```

Figure 4.2: mutable system logic state variables in `LQTYToken.sol`.

While these state variables do not currently get assigned after construction, a future code update could add such an assignment without resulting in an error.

Exploit Scenario
Alice adds a new feature to the Liquity protocol code. Adding the feature results in an assignment to one of the state variables `deployer`, `deploymentStartTime` and `lockupContractFactory`. Then, system logic, which is supposed to be immutable, ends up being mutable.

Recommendation
Short term, add `immutable` keyword to state variables `deployer`, `deploymentStartTime` and `lockupContractFactory`. This way future assignments to these variables will result in compiler errors.

Long term, review every state variable to make sure they have the correct type constraints.

5. Unclear if the gas pool account is controlled by someone

Severity: Low
Type: Access Controls
Target: `TroveManager.sol, BorrowerOperations.sol`

Difficulty: High
Finding ID: TOB-LQT-005

Description
The special gas pool account is used to mint LUSD tokens and pay for the gas operations performed by users, but is unclear if someone owns its private key.

The Liquity protocol mint LUSD to compensate the gas spent by users when they call certain functions, such as `openTrove`:

```
    function openTrove(uint _LUSDAmount, address _hint) external payable override {
      ...
      // Move the LUSD gas compensation to the Gas Pool
      _withdrawLUSD(GAS_POOL_ADDRESS, LUSD_GAS_COMPENSATION, LUSD_GAS_COMPENSATION);


      emit TroveUpdated(msg.sender, rawDebt, msg.value, stake,
BorrowerOperation.openTrove);
      emit LUSDBorrowingFeePaid(msg.sender, LUSDFee);
  }
```

Figure 5.1: `_requireValidRecipient` *in* `LUSDToken.sol`.

This address is defined as `0x0000000000000000000000000000000000000009A5` and it's just like any other address that can hold LUSD tokens. It is unclear whether someone owns its private key or not. If it exists, it can be easily used to steal the tokens minted to compensate for the cost of gas.

Exploit Scenario
Eve somehow gets the private key of the gas pool account and is able to steal all the tokens from it. The Liquity protocol cannot be modified in any sense, and therefore it cannot change this address and it needs to be re-deployed.

Recommendation
Short term, consider refactoring the code to avoid using the gas pool account. Instead, use a state variable that keeps track of the amount of LUSD tokens available to compensate gas costs.

Long term, avoid hard-coding any account address.

6. Reentrancy could lead to incorrect order of emitted events

Severity: Low
Type: Undefined Behavior
Target: `BorrowerOperations.sol`

Difficulty: Medium
Finding ID: TOB-LQT-006

Description
The order of operations in the `_moveTokensAndETHfromAdjustment` function in the `BorrowOperations` contract may allow an attacker to cause events to be emitted out of order.

When executing operations on a trove, the internal `_adjustTrove` function calls `_moveTokensAndETHfromAdjustment`.

```
      _moveTokensAndETHfromAdjustment(msg.sender, L.collChange, L.isCollIncrease,
_debtChange, _isDebtIncrease, L.rawDebtChange);

        emit TroveUpdated(_borrower, L.newDebt, L.newColl, L.stake,
BorrowerOperation.adjustTrove);
        emit LUSDBorrowingFeePaid(msg.sender,  L.LUSDFee);
    }
```

Figure 6.1: A snippet of the `BorrowerOperations._adjustTrove` *function.*

If LUSD is to be burned, resulting in collateralized ETH being returned, then this function invokes `sendETH` in the `ActivePool` contract.

```
  function sendETH(address _account, uint _amount) external override {
     _requireCallerIsBOorTroveMorSP();
     ETH = ETH.sub(_amount);
     emit EtherSent(_account, _amount);

     (bool success, ) = _account.call{ value: _amount }("");
     require(success, "ActivePool: sending ETH failed");
  }
```

Figure 6.2: The `ActivePool.sendETH` *function body.*

This call will actually perform the transfer of ETH to the borrower. As this function uses `call.value` without a specified gas amount, all of the remaining gas is forwarded to this call.

In the event that the borrower is a contract, this could trigger a callback into `BorrowerOperations`, executing the `_adjustTrove` flow above again. As the `_moveTokensAndETHfromAdjustment` call is the final operation in the function the state of the system on-chain cannot be manipulated. However, there are events that are emitted after this call. In the event of a reentrant call, these events would be emitted in the incorrect order.

Exploit Scenario

Alice operates a trove through a smart contract. She invokes a change in her trove that causes the Liquity system to return ETH to her smart contract. This triggers the contract's fallback function, which calls back into the Liquity system and triggers another similar operation. The event for the second operation is emitted first, followed by the event for the first operation. Any off-chain monitoring tools may now have an inconsistent view of on-chain state.

Recommendation
Short term, apply the checks-effects-interactions pattern and move the event emissions above the call to `_moveTokensAndETHfromAdjustment` to avoid the potential reentrancy.

Long term, use Slither to detect potential reentrant function calls.

7. Missing address validation when configuring contracts

Severity: Informational
Type: Data Validation
Target: `Throughout`

Difficulty: High
Finding ID: TOB-LQT-007

Description
As part of the post-deployment setup of the Liquity system, several contracts have `setAddresses` functions that initialize links to other system components (Figure 7.1). These functions are each callable only once as the final step of each is to renounce ownership of the contract. However, none of these addresses are validated in any way before being set. If an incorrect address is set, then that contract must be redeployed and due to the connections between most contracts, this is likely to result in the entire system having to be redeployed.

```
function setAddresses(
    address _troveManagerAddress,
    address _activePoolAddress
)
    external
    onlyOwner
{
    troveManagerAddress = _troveManagerAddress;
    activePoolAddress = _activePoolAddress;

    emit TroveManagerAddressChanged(_troveManagerAddress);
    emit ActivePoolAddressChanged(_activePoolAddress);

    _renounceOwnership();
}
```

Figure 7.1: An example `setAddresses` *function, from the* `DefaultPool` *contract.*

This missing validation is present in configuration functions throughout the contracts. The affected functions are:

- All contracts with a `setAddresses` function
- `SortedTroves.setParams`

Exploit Scenario
An error or typo in the deployment script causes the contract address of one of the system components to be recorded as `0x0` in each of the system's contracts. As a result, the entire system must be redeployed at significant gas cost.

Recommendation
Short term, add some basic input validation to all configuration functions. For example, check for the `0x0` address as this is the default value for an uninitialized address.

Long term, always perform some type of input validation. Use Slither to detect functions that would benefit from a `0x0` address check.

8. Permit opens the door for griefing contracts that interact with the Liquity Protocol

Severity: Informational
Type: Timing
Target: `LUSDToken.sol, LQTYToken.sol`

Difficulty: Low
Finding ID: TOB-LQT-008

Description
The `permit` function can be front-run to break the workflow from third-party smart contracts.

The Liquity `ERC20` contracts implements `permit`, which allows the allowance of a user to be changed based on a signature check using `ecrecover`:

```
   function permit
   (
       address owner,
       address spender,
       uint amount,
       uint deadline,
       uint8 v,
       bytes32 r,
       bytes32 s
   )
       external
       override
   {
       require(deadline == 0 || deadline >= now, 'LUSD: Signature has expired');
       bytes32 digest = keccak256(abi.encodePacked(uint16(0x1901),
                        domainSeparator(), keccak256(abi.encode(
                        _PERMIT_TYPEHASH, owner, spender, amount,
                        _nonces[owner]++, deadline))));
       address recoveredAddress = ecrecover(digest, v, r, s);
       require(recoveredAddress != address(0) &&
               recoveredAddress == owner, 'LUSD: Recovered address from the sig is not the
owner');
       _approve(owner, spender, amount);
   }
```

Figure 8.1: `permit` *in* `LUSDToken.sol`*.*

While this function is correctly implemented in terms of functionality, there is a potential security issue users must be aware of when developing contracts to interact with Liquity tokens:

```
## Security Considerations

Though the signer of a `Permit` may have a certain party in mind to submit their
transaction, another party can always front run this transaction and call `permit` before
the intended party. The end result is the same for the `Permit` signer, however.
```

Figure 8.2: Security considerations for ERC2612.

Exploit Scenario
Alice develops a smart contract that leverages `permit` to perform a `transferFrom` of LUSD without requiring a user to call `approve` first. Eve monitors the blockchain and notices this call to `permit`. She observes the signature and replays it to front-run her call, which produces a revert in Alice's contract and halts its expected execution.

Recommendation
Short term, properly document the possibility of griefing `permit` calls to warn users interacting with Liquity tokens. This will allow users to anticipate this possibility and develop alternate workflows in case they are targeted by it.

Long term, carefully monitor the blockchain to detect front-running attempts.

9. Closing troves requires owners to hold the full amount of LUSD minted

Severity: Low
Type: Data Validation
Target: `TroveManager.sol`

Difficulty: Low
Finding ID: TOB-LQT-009

Description
Users should hold the complete amount of LUSD minted, before closing their trove, otherwise, the trove closing operation will fail.

The Liquity protocol allows any user to deposit collateral and open troves in order to mint LUSD tokens:

```
function openTrove(uint _LUSDAmount, address _hint) external payable override {
    uint price = priceFeed.getPrice();


    _requireTroveisNotActive(msg.sender);
    …
}
```

Figure 9.1: Header of the `openTrove` *function in* `BorrowOperations.sol`.

If any user wants to close their trove, they can use the `closeTrove` function:

```
 function closeTrove() external override {
     _requireTroveisActive(msg.sender);
     _requireNotInRecoveryMode();

     troveManager.applyPendingRewards(msg.sender);

     uint coll = troveManager.getTroveColl(msg.sender);
     uint debt = troveManager.getTroveDebt(msg.sender);

     troveManager.removeStake(msg.sender);
     troveManager.closeTrove(msg.sender);

     // Burn the debt from the user's balance, and send the collateral back to the user
     _repayLUSD(msg.sender, debt.sub(LUSD_GAS_COMPENSATION));
     activePool.sendETH(msg.sender, coll);
     // Refund gas compensation
     _repayLUSD(GAS_POOL_ADDRESS, LUSD_GAS_COMPENSATION);

     emit TroveUpdated(msg.sender, 0, 0, 0, BorrowerOperation.closeTrove);
 }
```

Figure 9.2: `closeTrove` *in* `BorrowOpertions.sol`..

This operation requires that the user account holds the corresponding amount of LUSD token minted. According to the code, these LUSD will be burned to remove the trove debt.

Hower, the fact the user should hold enough LUSD is not clearly documented or tested anywhere in the codebase. Also, it was unclear if the LUSD provided to the stability pool can be used directly to repay the debt.

```
assertion in closeTrove_should_not_revert: failed!💥
  Call sequence:
    openTrove_should_not_revert(1) Value: 0x10b66c7775da5e4
    provideToSP_should_not_revert()
    closeTrove_should_not_revert()
```

Figure 9.3: Echidna report of this issue.

Exploit Scenario
Alice creates a new trove. Then she calls `provideToSP` with any positive amount of tokens. Later, she wants to close her trove, but the call fails. So she is unable to close her trove.

Recommendation
Short term properly document and test this corner case and add a suitable revert message.

Long term, make sure every system property is carefully specified and use Echidna or Manticore to test it.

10. Failed lockup error is never thrown

Severity: Low | Difficulty: Low
Type: Auditing and Logging | Finding ID: TOB-LQT-010
Target: `LockupContractFactory.sol`, `OneYearLockupContract.sol`, `CustomDurationLockupContract.sol`

Description
The locking contract function fails to signal an error in the return value, causing some error code to be unreachable.

The `LockupContractFactory` contract checks that a `lockContract` call has succeeded by checking the returned boolean. However, the lockup contracts throw errors themselves. If the code execution reaches the factory contract after the call to `lockContract`, the success variable will always be `true`. Therefore, the error in the factory contract will never be thrown.

```
function lockOneYearContracts(address[] calldata addresses) external override {
    for (uint i = 0; i < addresses.length; i++ ) {

        ...

        bool success = oneYearlockupContract.lockContract();
        require(success, "LockupContractFactory: Failed to lock the contract");
    }
}
```

Figure 10.1: `lockOneYearContracts` *in* `LockupContractFactory.sol`.

```
function lockContract() external returns (bool) {
    _requireCallerIsLockupDeployer();
    _requireContractIsNotActive();
    _requireLQTYBalanceAtLeastEqualsEntitlement();

    active = true;
    lockupStartTime = block.timestamp;
    emit OYLCLocked(lockupStartTime);
    return true;
}
```

Figure 10.2: `lockContract` *in* `OneYearLockupContract.sol`.

Exploit Scenario
Bob is monitoring the factory contract for failed `lockContract` calls by checking if any transaction reverts with the message `LockupContractFactory: Failed to lock the contract`. Since this error is never thrown he will catch none of the failing `lockContract` calls.

Recommendation
Short term, remove the boolean return from the `lockContract` functions, and remove the success check from the factory contract.

Long term, make sure all paths through the code are reachable.

11. Duplicated functions pose future risk of diverging

Severity: Low
Type: Patching
Target: `TroveManager.sol, BorrowerOperations.sol`

Difficulty: High
Finding ID: TOB-LQT-011

Description
The `TroveManager` and `BorrowerOperations` contracts both contain a function to calculate the current TCR. Although they currently are identical in terms of outcome, they differ in their implementation. This could pose future problems in case one is updated while the other one is not.

```
function getTCR() public view override returns (uint TCR) {
    uint price = priceFeed.getPrice();
    uint entireSystemColl = getEntireSystemColl();
    uint entireSystemDebt = getEntireSystemDebt();

    TCR = LiquityMath._computeCR(entireSystemColl, entireSystemDebt, price);

    return TCR;
}
```

Figure 11.1: `getTCR` *in* `TroveManager.sol`.

```
function _getTCR() internal view returns (uint TCR) {
    uint price = priceFeed.getPrice();
    uint activeColl = activePool.getETH();
    uint activeDebt = activePool.getLUSDDebt();
    uint liquidatedColl = defaultPool.getETH();
    uint closedDebt = defaultPool.getLUSDDebt();

    uint totalCollateral = activeColl.add(liquidatedColl);
    uint totalDebt = activeDebt.add(closedDebt);

    TCR = LiquityMath._computeCR(totalCollateral, totalDebt, price);

    return TCR;
}
```

Figure 11.2: `_getTCR` *in* `BorrowerOperations.sol`.

It is recommended to not duplicate functions in different contracts. This increases the risk of diverging in the future, and could significantly affect the system properties. Instead, define the function in one place and let other contracts call that function.

The system also contains duplicated functions to check the recovery mode in `TroveManager` and `BorrowerOperations`.

Exploit Scenario

Alice, a developer of Liquity, is tasked to update the `BorrowerOperations` contract. Her update requires an update to the `_getTCR` function. However, Alice forgets to also update the `TroveManager.getTCR` function. The system now contains two functions to get the TCR that return different results.

Recommendation
Short term, consolidate a single implementation of `getTCR`/`checkRecoveryMode` function in the entire system.

Long term, avoid duplicating logic.

12. Total system collateral and debt calculations are repeated multiple times risking divergence and decreasing readability

Severity: Low
Type: Patching
Target: `TroveManager.sol, BorrowerOperations.sol`

Difficulty: Medium
Finding ID: TOB-LQT-012

Description
`TroveManager` contains the functions `getEntireSystemColl()` and `getEntireSystemDebt()`. The calculations they perform occur inline at least 5 additional times in `TroveManager.sol` and `BorrowerOperations.sol`. This could introduce bugs if only one of these sites is modified. It also makes the code harder to read.

```
function getEntireSystemColl() public view override returns (uint entireSystemColl) {
    uint activeColl = activePool.getETH();
    uint liquidatedColl = defaultPool.getETH();
    return activeColl.add(liquidatedColl);
}
```

Figure 12.1: `getEntireSystemColl` *in* `TroveManager.sol`*.*

```
function getEntireSystemDebt() public view override returns (uint entireSystemDebt) {
    uint activeDebt = activePool.getLUSDDebt();
    uint closedDebt = defaultPool.getLUSDDebt();
    return activeDebt.add(closedDebt);
}
```

Figure 12.2: `getEntireSystemDebt` *in* `TroveManager.sol`*.*

```
        uint totalColl = activePool.getETH().add(defaultPool.getETH());
        uint totalDebt = activePool.getLUSDDebt().add(defaultPool.getLUSDDebt());
```

Figure 12.3: Duplicate implementation of totals in `_getNewTCRFromTroveChange` *in* `BorrowerOperations.sol`*.*

Other examples of duplicate totals calculation implementations:

- `_getTCR` function in `BorrowerOperations.sol` (Figure 11.2)
- Lines 505-506 in `TroveManager.sol`
- Lines 640-641 in `TroveManager.sol`
- Lines 1202-1204 in `TroveManager.sol`

Exploit Scenario
Alice, a developer of Liquity, adds a new pool to the system that influences total system collateral. She changes the calculation of total system collateral in only two places. This

introduces a bug where different parts of the system see the system in a different state. Then, Eve can exploit this bug to interact with the system in ways that should not be possible.

Recommendation
Short term, consolidate a single implementation for `getEntireSystemColl()` and `getEntireSystemDebt()` across the entire codebase. .

Long term, avoid duplicating logic.

13. Troves can be improperly removed under certain circumstances which results in unexpected reverts

Severity: High
Type: Data Validation
Target: Throughout

Difficulty: High
Finding ID: TOB-LQT-013

Description
In certain circumstances, troves are not properly inserted in the sorted list and therefore, cannot be removed, causing unexpected revert and potential loss of funds.

The list of troves is kept sorted using functions like the reInsert function, which ensures troves are properly removed and then re-inserted in-order:

```
   function reInsert(address _id, uint256 _newICR, uint _price, address _prevId, address
_nextId) external override {
       _requireCallerIsBOorTroveM();
       // List must contain the node
       require(contains(_id));


       // Remove node from the list
       _remove(_id);


       if (_newICR > 0) {
           // Insert node if it has a non-zero ICR
           _insert(_id, _newICR, _price, _prevId, _nextId);
       }
   }
```

Figure 13.1: `reInsert` *from the* `SortedTroves` *contract.*

However, it is possible for `reInsert` to be called on a trove with ICR equal to zero. That trove will be removed, but not re-inserted. The discrepancy arises from TroveManager still believing that the trove exists, while it has been removed from the SortedTroves contract. Any following call that works with this particular trove will fail due to the TroveManager initiating a call that checks that the trove exists (and is expected to succeed) in SortedTroves, which it doesn't. Echidna finds a sequence of transactions to reproduce this issue:

```
assertion in addColl_should_not_revert: failed!💥
  Call sequence:
    openTrove_should_not_revert(0) Value: 0x10b30caa1ef1b01
    setPrice(1)
    addColl_should_not_revert() Value: 0x1
    addColl_should_not_revert() Value: 0x1
```

Figure 13.2: Echidna report of this issue.

Exploit Scenario
Alice opens a trove and deposits some collateral. Later, the price drops. If Alice deposits a small amount of collateral, her trove will be re-inserted with ICB = 0, causing any future calls for that trove to fail. Alice collateral will be impossible to redeem.

Recommendation
Short term, avoid any trove to be (re-)inserted using ICB = 0.

Long term, make sure every system property is carefully specified and use Echidna or Manticore to test it.

14. Initial redeem can unexpectedly revert if no redeemable troves

Severity: Low
Type: Undefined Behavior
Target: `TroveManager.sol`

Difficulty: High
Finding ID: TOB-LQT-014

Description
The initial redemption in the system can cause an assertion to unexpectedly fail if there are no troves with ICR >= MCR.

The `baseRate` is initially set to zero. When a redemption has succeeded the `baseRate` will be set to a non-zero value. Once it's non-zero it should not be possible to become zero again. For defense-in-depth the below assertion ensures this.

```
// Update the baseRate state variable
baseRate = newBaseRate < 1e18 ? newBaseRate : 1e18;  // cap baseRate at a maximum of 100%
assert(baseRate <= 1e18 && baseRate > 0); // Base rate is always non-zero after redemption
```

Figure 14.1: New `baseRate` *check in* `_updateBaseRateFromRedemption` *from the* `TroveManager` *contract.*

If the first ever redemption does not find any troves that can be redeemed from, the `newBaseRate` will be zero. This will make the above assertion fail. This reverts the transaction without an error message and uses up all the gas sent with the transaction.

The deeper issue here is that there is no check (with a descriptive error message) that reverts the transaction once it's determined that there are no troves with ICR >= MCR. Instead an assertion fails, which is not intended to ever be triggered.

```
assertion in redeemCollateral_should_not_revert: failed!💥
  Call sequence:
    openTrove_should_not_revert(1) Value: 0x10bb94cfd8e9714
    redeemCollateral_should_not_revert()
```

Figure 14.2: Echidna report of this issue.

Exploit Scenario
There are no troves in the system with ICR >= MCR, and no redemptions have yet been attempted. Alice initiates a redemption and the transaction reverts due to the assertion failure.

Recommendation
Short term, implement a check, with a proper revert message, that fails the transaction if there are no redeemable troves.

Long term, implement property-based testing using Echidna to catch these kinds of errors.

15. Redeem without redemptions might still return success

Severity: Medium
Type: Undefined Behavior
Target: `TroveManager.sol`

Difficulty: Medium
Finding ID: TOB-LQT-015

Description
A redeem when there are no redeemable troves does not cause the transaction to revert. Note that this excludes the very first redemption (see TOB-LQT-014).

When a redemption is initiated the smart contract will check if there are any troves with ICR >= MCR, and use those to fulfill the redemption. If no redeemable troves are found the `currentBorrower` will equal `address(0).` This information could have been used to revert the transaction.

```
if (_isValidFirstRedemptionHint(_firstRedemptionHint, price)) {
    currentBorrower = _firstRedemptionHint;
} else {
    currentBorrower = sortedTroves.getLast();
    // Find the first trove with ICR >= MCR
    while (currentBorrower != address(0) && getCurrentICR(currentBorrower, price) < MCR) {
        currentBorrower = sortedTroves.getPrev(currentBorrower);
    }
}
```

Figure 15.1: The loop used to find redeemable troves in the `redeemCollateral` *function from the* `TroveManager` *contract.*

Instead, without redeemable troves the execution will continue. No changes will be made to the contract storage variables, and an event is emitted indicating that a redemption of zero occurred.

It is unlikely that regular users will check the emitted event. Instead, they will assume that if the transaction succeeded, at least some part of the redemption was fulfilled. However, this might not be the case.

Just as is in TOB-LQT-014, the deeper issue is that there is no check (with a descriptive error message) that reverts the transaction once it's determined that there are no troves with ICR >= MCR.

Exploit Scenario
Alice initiates a redemption. There are no troves with an ICR >= MCR. Alice's transaction succeeds. Alice believes her redemption request was (partially) fulfilled.

Recommendation
Short term, implement a check, with a proper revert message, that fails the transaction if there are no redeemable troves.

Long term, implement property-based testing using Echidna to catch these kinds of errors.

A. Vulnerability Classifications

Vulnerability Classes	
Class	**Description**
Access Controls	Related to authorization of users and assessment of rights
Auditing and Logging	Related to auditing of actions or logging of problems
Authentication	Related to the identification of users
Configuration	Related to security configurations of servers, devices, or software
Cryptography	Related to protecting the privacy or integrity of data
Data Exposure	Related to unintended exposure of sensitive information
Data Validation	Related to improper reliance on the structure or values of data
Denial of Service	Related to causing system failure
Error Reporting	Related to the reporting of error conditions in a secure fashion
Patching	Related to keeping software up to date
Session Management	Related to the identification of authenticated users
Testing	Related to test methodology or test coverage
Timing	Related to race conditions, locking, or order of operations
Undefined Behavior	Related to undefined behavior triggered by the program
<other class>	<add any other class which is missing for the audit, delete the row if everything fits into prior ones>

Severity Categories	
Severity	**Description**
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or Defense in Depth
Undetermined	The extent of the risk was not determined during this engagement

Low	The risk is relatively small or is not a risk the customer has indicated is important
Medium	Individual user's information is at risk, exploitation would be bad for client's reputation, moderate financial impact, possible legal implications for client
High	Large numbers of users, very bad for client's reputation, or serious legal or financial implications

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploit was not determined during this engagement
Low	Commonly exploited, public tools exist or can be scripted that exploit this flaw
Medium	Attackers must write an exploit, or need an in-depth knowledge of a complex system
High	The attacker must have privileged insider access to the system, may need to know extremely complex technical details, or must discover other weaknesses in order to exploit this issue

B. Code Maturity Classifications

Code Maturity Classes	
Category Name	**Description**
Access Controls	Related to the authentication and authorization of components.
Arithmetic	Related to the proper use of mathematical operations and semantics.
Assembly Use	Related to the use of inline assembly.
Centralization	Related to the existence of a single point of failure.
Upgradeability	Related to contract upgradeability.
Function Composition	Related to separation of the logic into functions with clear purpose.
Front-Running	Related to resilience against front-running.
Key Management	Related to the existence of proper procedures for key generation, distribution, and access.
Monitoring	Related to use of events and monitoring procedures.
Specification	Related to the expected codebase documentation.
Testing & Verification	Related to the use of testing techniques (unit tests, fuzzing, symbolic execution, etc.).

Rating Criteria	
Rating	**Description**
Strong	The component was reviewed and no concerns were found.
Satisfactory	The component had only minor issues.
Moderate	The component had some issues.
Weak	The component led to multiple issues; more issues might be present.
Missing	The component was missing.

Not Applicable	The component is not applicable.
Not Considered	The component was not reviewed.
Further Investigation Required	The component requires further investigation.

C. Code Quality

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

General suggestions:

- **Do not use one-letter variable names.** The smart contract code uses variables with very short names that can be difficult to parse when the code is modified or reviewed. Use full names, e.g., `liquidateFunction` instead of `L`.
- **Consider replacing `if(a == true)` with `if(a)` and `if(a == false)` with `if(!a)`.** This will make the code easier to understand, maintain, and review.

`BorrowOperations.sol`:

- **Consider refactoring the adjustTrove function.** This `adjustTrove` function implements some important operations, but it is difficult to review because of the lack of inline documentation and their checks cover a large variety of corner cases. This will make the code easier to understand, maintain, and review.
- **Consider using require instead of assert for regular checks.** The `_activePoolAddColl` checks the success of the call using an `assert` instead of a `require`. This is unlike all the other places where a similar check is performed. Use `assert` to ensure system invariants hold.

`TroveManager.sol`:

- **Consider adding a comment explaining the rationale behind the redistribution error computation.** The error computation code is not obvious at first glance and results in a slither warning due to division before multiplication. Adding a comment would ease understanding when the code is modified or reviewed.

```
uint ETHRewardPerUnitStaked = ETHNumerator.div(totalStakes);
uint LUSDDebtRewardPerUnitStaked = LUSDDebtNumerator.div(totalStakes);


lastETHError_Redistribution = ETHNumerator.sub(ETHRewardPerUnitStaked.mul(totalStakes));
lastLUSDDebtError_Redistribution =
LUSDDebtNumerator.sub(LUSDDebtRewardPerUnitStaked.mul(totalStakes));
```

Figure C.1: Redistribution Error computation.

- **Consider simplifying the recovery mode check.** The existing code to check this spans multiple lines and makes the code less readable. It could be replaced with simply: `return TCR < CCR`

```
if (TCR < CCR) {
```

```
    return true;
} else {
    return false;
}
```

Figure C.2: Recovery mode check.

`SortedTroves.sol`:

- **Consider normalizing the (capitalization of) names of variables and event arguments.** By normalizing the names the code becomes more readable and it is less likely a developer accidentally uses the wrong variable due to incoherent names. The affected names are:
  - `borrowerOperationsAddress` vs `TroveManagerAddress`.
  - `_newTrovelManagerAddress` vs `_borrowerOperationsAddress`.
- **Typo,** `_newTrovelManagerAddress` should be `_newTroveManagerAddress`.

`OneYearLockupContract.sol, CustomDurationLockupContract.sol`:

- **Consider normalizing the transfer recipient.** Since both of the currently used variables point to the same account, the code readability would improve if both used the same variable, either `msg.sender` or `beneficiary`.

```
lqtyToken.transfer(beneficiary, LQTYBalance);
```

Figure C.3: `OneYearLockupContract.sol` *withdrawal transfer.*

```
lqtyToken.transfer(msg.sender, LQTYBalance);
```

Figure C.4: `CustomDurationLockupContract.sol` *withdrawal transfer.*

- **Consider normalizing the order of operations in the withdrawLQTY functions and adhering to the Checks-Effects-Interactions pattern.** In some cases, the `active` variable is assigned first, and then rest of the code, while in the other, the order is changed.

```
active = false;

uint LQTYBalance = lqtyToken.balanceOf(address(this));
lqtyToken.transfer(beneficiary, LQTYBalance);
emit OYLCUnlockedAndEmptied(block.timestamp);
```

Figure C.5: `withdrawLQTY` *from* `OneYearLockupContract.sol`.

```
uint LQTYBalance = lqtyToken.balanceOf(address(this));
lqtyToken.transfer(msg.sender, LQTYBalance);

active = false;
emit CDLCUnlockedAndEmptied(block.timestamp);
```

Figure C.6: `withdrawLQTY` *from* `CustomDurationLockupContract.sol`.

`PriceFeed.sol`:

- **Consider changing the type of `TARGET_DIGITS` to `u8`.** `TARGET_DIGITS` is only used in operations where the other operand is `u8`. This was found using the script mentioned in Appendix D.

`LQTYStaking.sol`:

- **Consider removing unused return variables.** The `_sendETHGainToUser` function has a boolean return variable. However, the function does not explicitly return a value, nor do the calling functions check the function return value.
- **Consider only initiating ETH/Token transfers when necessary.** The `stake` function only gathers gains if the caller already had staked LQTY. However, at the end of the function the gains are transferred, regardless if there are no gains. Consider also surrounding the gains transfer lines with `if (currentStake != 0) {`

```
uint ETHGain;
uint LUSDGain;
// Grab any accumulated ETH and LUSD gains from the current stake
if (currentStake != 0) {
    ETHGain = _getPendingETHGain(msg.sender);
    LUSDGain = _getPendingLUSDGain(msg.sender);
}

...

// Send accumulated LUSD and ETH gains to the caller
lusdToken.transfer(msg.sender, LUSDGain);
_sendETHGainToUser(ETHGain);
```

Figure C.7: `stake` *function from* `LQTYStaking.sol`.

D. Known Issues

The following issues were identified by the client before starting this audit. We took a look at these to make sure they were reasonably rated in severity and include a recommendation to fix them.

LQT-D01. Loss evasion by Stability Pool depositors front-running liquidations:

Description
Users depositing liquidity into the Stability Pool could avoid liquidation by monitoring upcoming price drops or liquidations, and then front-running them with a call to `withdrawFromSP` to avoid the loss.

Recommendation
Short term, consider allowing to call `withdrawFromSP` during certain times of the day or week (e.g. 2 days per week), so users have to either withdraw their liquidity during that period or wait until the next one. This will reduce the front-running window of opportunity without adding a lot of the complexity other front-running mitigations would introduce.

Long term, monitor the blockchain to identify users front-running liquidations and recommend a suitable gas price for liquidations to minimize the risk.

LQT-D02: Making profiting front-running other transactions

Description
This issue comprises different situations where the user can make profit submitting transactions with high-gas prices to front-run unconfirmed operations. In particular, users could decide to speculate on:

- upcoming liquidations to front-run them with a `provideToSP` call, in order to make a profit.
- upcoming price changes to front-run them with a `redeemCollateral` call, in order to make a profit.

Recommendation
We do not think this poses an immediate risk, since the users will bring liquidity and make a fair profit of it. However, we do recommend monitoring the blockchain to identify any

users front-running liquidations and recommend a suitable gas price for liquidations or price updates to minimize the risk.

LQT-D03: Explore possible issue with `ecrecover`

Description
Signatures passed to `ecrecover` are subject to malleability. That is, given a valid signature, an attacker can modify it in a way that produces a different valid signature. This malleability was patched at the Ethereum transaction layer in the Homestead hardfork, but remains present in `ecrecover`. OpenZeppelin has developed a library, ECDSA.sol, within their suite of contracts that performs input validation that among other things prevents this type of malleability by requiring signatures to be in a particular range of values. However, Liquity uses the builtin `ecrecover` directly. In this case, the signature includes replay protection via a nonce that would prevent an attacker from replaying a modified signature. The only remaining potential avenue for an attacker would be griefing. In the event that a user expects to see a particular signature value appear on-chain, an attacker could frontrun their transaction with a modified signature. The outcome would be the same (i.e., the attacker would get no benefit), but the user's transaction would fail, producing a confusing user experience.

Recommendation
Though the current approach poses no immediate security risk, migrating to the OpenZeppelin ECDSA would enforce a canonical signature format and would provide defense in depth going forward. Additionally, as it's a commonly used library it may also provide a more familiar and user-friendly interface than using the builtin ecrecover directly.

E. Check operations on integers smaller uint256 with Slither

Trail of Bits used Slither, a Solidity static analysis framework, to automatically find and display uses of integer types smaller than `uint256` and `int256`. Examples of such integers used in the Liquity system are `uint8`, `uint80`, and `uint128`. All such uses were inspected manually. No issues were found.

```
- PriceFeed
  - getPrice() public
    - [x]   40: produces uint80: priceAggregator.latestRoundData()
    - [x]   45: produces uint8: priceAggregator.decimals()
    - [x]   49: reads uint8: answerDigits > TARGET_DIGITS
        - NOTE: TARGET_DIGITS could have type u8
    - [x]   50: reads uint8 and produces uint8: answerDigits - TARGET_DIGITS
    - [x]   52: reads uint8: answerDigits < TARGET_DIGITS
    - [x]   53: reads uint8: TARGET_DIGITS - answerDigits
```

Figure E.1: Excerpt of the output of the script.

Note that the script below produces some false positives.

```
from typing import Set, Optional

from slither.core.solidity_types.elementary_type import Int, Uint

from slither import Slither
from slither.slithir.operations import Operation, OperationWithLValue
from slither.core.solidity_types.type import Type
from slither.core.solidity_types import (
    ElementaryType,
    MappingType,
    ArrayType,
    UserDefinedType,
)
from slither.core.declarations import Structure

def get_type(actual: Type, expected: Set[Type]) -> Optional[Type]:
    if isinstance(actual, list):
        for x in actual:
            t = get_type(x, expected)
            if t:
                return t
    if actual in expected:
        return actual
    if isinstance(actual, ElementaryType):
        return None
    if isinstance(actual, MappingType):
        t = get_type(actual.type_from, expected)
        if t:
            return t
        return get_type(actual.type_to, expected)
    if isinstance(actual, UserDefinedType) and isinstance(actual.type, Structure):
        for elem in actual.type.elems.values():
```

```
            t = get_type(elem.type, expected)
            if t:
                return t
        return None
    if isinstance(actual, ArrayType):
        return get_type(actual.type, expected)

    return None


small_int_types = frozenset(
    ElementaryType(type_name)
    for type_name in Int + Uint
    if type_name not in set(["uint", "uint256", "int", "int256"])
)


def get_small_int(actual: Type) -> bool:
    return get_type(actual, small_int_types)


def process_ir(ir: Operation, lines_printed: Set[int]):
    line = ir.expression.source_mapping["lines"][0]
    if line in lines_printed:
        return

    lvalue = None
    if isinstance(ir, OperationWithLValue):
        if ir.lvalue:
            lvalue = get_small_int(ir.lvalue.type)

    read = next(
        filter(None, (get_small_int(x.type) for x in ir.read if hasattr(x, "type"))),
        None,
    )

    if lvalue and read:
        print(
            f"    - [ ] {line:4}: reads {read} and produces {lvalue}: {ir.expression}"
        )
    elif lvalue:
        print(f"    - [ ] {line:4}: produces {lvalue}: {ir.expression}")
    elif read:
        print(f"    - [ ] {line:4}: reads {read}: {ir.expression}")

    if lvalue or read:
        lines_printed.add(line)


slither = Slither(".")

for contract in slither.contracts:

    lower = contract.name.lower()
    if "echidna" in lower or "test" in lower or "properties" in lower:
        continue

    is_contract_printed = False

    def print_contract():
```

```
        global is_contract_printed
        if is_contract_printed:
            return
        print(f"- {contract.name}")
        is_contract_printed = True

    for v in contract.variables:
        if get_small_int(v.type):
            print_contract()
            print(f"  - {v.type} {v.name}")

    lines_printed = set()
    for function in contract.functions_and_modifiers_declared:
        # includes parameters and return values
        is_uint128_in_vars = any(
            True for v in function.variables if get_small_int(v.type)
        )
        if is_uint128_in_vars:
            print_contract()
            params_str: str = ", ".join(
                f"{p.type} {p.name}" for p in function.parameters
            )
            print(f"  - {function.name}({params_str}) {function.visibility}")
            for ir in function.slithir_operations:
                process_ir(ir, lines_printed)
```

Figure E.1: Slither script used to detect uses of small integer types.

F. Check receiving and sending of Ether with Slither

Trail of Bits used Slither, a Solidity static analysis framework, to automatically find and display where Ether is received and sent in the Liquity system. For the cases where Ether is sent, the externally callable paths leading to these were also detected with some limitations. These areas of code are especially sensitive. All results were inspected manually. No issues were found.

```
# ETH Egress

## StabilityPool._sendETHGainToDepositor(uint256) internal can send ETH

### in externally callable paths

- `StabilityPool.withdrawFromSP(uint256)` *external* ->
`StabilityPool._sendETHGainToDepositor(uint256)` *internal*
- `StabilityPool.provideToSP(uint256,address)` *external* ->
`StabilityPool._sendETHGainToDepositor(uint256)` *internal*
```

Figure F.1: Excerpt of the output of the script.

Note that the script below requires some manual review of the results since it can produce false positives.

```
from typing import Tuple, Set, FrozenSet
from slither import Slither
from slither.slithir.operations import (
    HighLevelCall,
    LowLevelCall,
    Send,
    Transfer,
)
from slither.core.declarations import Function

CONTRACTS_TO_IGNORE = {
    "BorrowerOperationsTester",
    "TestAUTO",
    "PropertiesAUTO",
    "ActivePoolTester",
    "EchidnaTester",
    "DefaultPoolTester",
    "NonPayable",
    "StabilityPoolTester",
    "EchidnaProxy",
    "Destructible",
}


def is_externally_callable(function: Function) -> bool:
    return function.visibility in ["public", "external"]


def find_externally_callable_paths(
```

```
    slither: Slither, target_function: Function, current_path: Tuple[Function, ...] = ()
) -> FrozenSet[Tuple[Function, ...]]:
    current_path = (target_function, *current_path)

    result_paths: Set[Tuple[Function, ...]] = set()

    for contract in slither.contracts:
        if contract.name in CONTRACTS_TO_IGNORE:
            continue
        for function in contract.functions:
            if function in current_path:
                continue

            called_functions = set()
            for _, f in function.high_level_calls:
                called_functions.add(f)
            for _, f in function.library_calls:
                called_functions.add(f)
            for f in function.internal_calls:
                if isinstance(f, Function):
                    called_functions.add(f)

            if target_function in called_functions:
                result_paths |= find_externally_callable_paths(
                    slither, function, current_path
                )

    if is_externally_callable(target_function):
        result_paths.add(current_path)

    return frozenset(result_paths)


def source_mapping_to_github_url(source_mapping) -> str:
    result = source_mapping["filename_relative"]
    lines = source_mapping["lines"]
    result += "#L" + str(lines[0])
    if len(lines) > 1:
        result += "-L" + str(lines[-1])
    return result


slither = Slither(".")


print("# ETH Egress")
print("")

for contract in slither.contracts:
    if contract.name in CONTRACTS_TO_IGNORE:
        continue
    if contract.is_interface:
        continue
    for function in contract.functions_and_modifiers_declared:
        for node in function.nodes:
            for ir in node.irs:
                if not isinstance(ir, (HighLevelCall, LowLevelCall, Transfer, Send)):
                    continue
                if ir.call_value is None:
                    continue
```

```
                print(
                    f"## {function.canonical_name} {function.visibility} can send ETH"
                )
                print("")
                source_mapping = ir.node.source_mapping
                paths = find_externally_callable_paths(slither, function)
                print("")
                print("### in externally callable paths")
                print("")
                if paths:
                    for path in paths:
                        path_str = " -> ".join(
                            f"`{x.canonical_name}` *{x.visibility}*" for x in path
                        )
                        print(f"- {path_str}")
                else:
                    print("none")
                print("")

print("")
print("# ETH Ingress")
print("")

for contract in slither.contracts:
    if contract.name in CONTRACTS_TO_IGNORE:
        continue
    if contract.is_interface:
        continue
    for function in contract.functions_and_modifiers_declared:
        if not is_externally_callable(function):
            continue
        if not function.payable:
            continue
        print(f"- `{function.canonical_name}` can receive ETH")
```

Figure F.2: Slither script used to detect receiving and sending of Ether.

G. Fix Log

During the week of January 11th 2021, Trail of Bits reviewed Liquity's fixes and mitigations to address issues TOB-LQT-001 through TOB-LQT-015.
The results of this fix review can be observed below:

#	Title	Severity	Status
1	ERC20 transfer restriction can disturb 3rd-party interactions	Medium	Not Fixed
2	Removed after discussion with Liquity team		
3	No restrictions on minting to invalid recipients	Informational	Not Fixed
4	Insufficient enforcement of immutability of system logic variables through the Solidity compiler	Informational	Fixed
5	Unclear if the gas pool account is controlled by someone	Low	Fixed
6	Reentrancy could lead to incorrect order of emitted events	Low	Fixed
7	Missing address validation when configuring contracts	Informational	Fixed
8	Permit opens the door for griefing contracts that interact with the Liquity Protocol	Informational	Fixed
9	Closing troves requires owners to hold the full amount of LUSD minted	Low	Fixed
10	Failed lockup error is never thrown	Low	Fixed
11	Duplicated functions pose future risk of diverging	Low	Fixed

12	Total system collateral and debt calculations are repeated multiple times risking divergence and decreasing readability	Low	Fixed
13	Troves can be improperly removed under certain circumstances which results in unexpected reverts	High	Fixed
14	Initial redeem can unexpectedly revert if no redeemable troves	Low	Fixed
15	Redeem without redemptions might still return success	Medium	Fixed

For additional information, please refer to the detailed fix log.

Detailed Fix Log

TOB-LQT-001: ERC20 transfer restriction can disturb 3rd-party interactions
Risk Accepted. Liquity decided that:

> *After some discussion we decided to keep these ERC20 token transfer restrictions in our system. We believe the benefit of preventing users from accidentally losing a large sum of money by transferring tokens to a core system contract is significant.*
>
> *Although these restrictions could break certain third-party interactions that send tokens to a core Liquity contract, we would consider any such interaction already dysfunctional since those tokens would be thereafter forever lost. It seems better overall that this action is not possible.*

We consider this a reasonable tradeoff.

TOB-LQT-003: No restrictions on minting to invalid recipients
Risk Accepted. Liquity decided that:

> *Currently, there is no logic in the system whereby a custom address gets passed to* `_mint`*. The function is only called in specific situations: minting tokens to the borrower when new debt is issued, and minting the LUSDfee to the staking contract.*
>
> *The issue suggests placing restrictions on* `_mint` *in case of some future new feature that allows* `_mint` *to receive an invalid address.*
>
> *Since we have no plans to add such a feature and the system will be immutable after launch, we decided not to add a* `requireValidRecipient` *here.*

TOB-LQT-004: Insufficient enforcement of immutability of system logic variables through the Solidity compiler
Fixed. In commit 2755a216b98ec4b5bb93cb274431e2dcc801b558 Liquity not only addressed the short term recommendations but also the long term recommendations by fixing additional occurrences of the issue.

TOB-LQT-005: Unclear if the gas pool account is controlled by someone
Fixed. In commit 95f64a60949f9e49ffae150b91a7f0e0dc4e16b5 Liquity replaced the `0x0000000000000000000000000000000000000009A5` gas address by a `GasPool` contract with empty implementation that is deployed as part of the Liquity system. This removes any chance of someone controlling the gas pool address. Note that the fix differs from the original recommendation, which was to use a state variable to keep track of the gas

balance. The additional contract introduces some complexity. That complexity is likely less than the complexity that would be introduced by using a state variable since using an address allows Liquity to reuse a lot of functionality.

TOB-LQT-006: Reentrancy could lead to incorrect order of emitted events
Fixed. In commit d1619bb7e9a172bcdbe09a8f643b2f4a5138dfb7 Liquity moved event emission before calls for the functions mentioned in the issue as well as additional functions, thereby fixing this issue.

TOB-LQT-007: Missing address validation when configuring contracts
Fixed. In commit b507f5ed93dacc8a5b734961de1e42509ba6e0f3 Liquity introduced a `checkContract` function that reverts if an address is `0x0` or does not have any bytecode. A close look at the changes revealed that `checkContract` is used to check all arguments to all functions named `setAddresses` thereby fixing the issue.

TOB-LQT-008. Permit opens the door for griefing contracts that interact with the Liquity Protocol
Fixed. In commit 52a97fcf2f1d4429638288819ecd07e399b97483 Liquity documents the possibility of `permit` being frontrun in the `README.md` of the Liquity codebase thereby fixing the issue.

TOB-LQT-009: Closing troves requires owners to hold the full amount of LUSD minted
Fixed. In commit 47039fe4b635bc944851f7fc69c05ca1c4609e93 Liquity adds a `require`, documentation and tests mentioning and ensuring that a trove owner must hold the full trove debt in LUSD to successfully close a trove. That fixes the issue.

TOB-LQT-010: Failed lockup error is never thrown
Fixed. In commit 8f5212a6479d5b5e40a5454a79c9c64a57c448a3 Liquity removes the return value from `lockContract` of the lockup contracts as well as the unreachable requires in `LockupContractFactory`, thereby fixing the issue.

TOB-LQT-011: Duplicated functions pose future risk of diverging
Fixed. In commit 986d2d53cedfccefe754e3afdbf67e28e6e9e19b Liquity removed the duplicate implementations of `getTCR` and `_checkRecoveryMode` from `BorrowerOperations` and `TroveManager`. They consolidated them into a single implementation in `LiquityBase` thereby fixing the issue.

TOB-LQT-012: Total system collateral and debt calculations are repeated multiple times risking divergence and decreasing readability
Fixed. In commit 9c8773c0079d52a1e340262655aa97a9ddd106e2 Liquity added `getEntireSystemColl` and `getEntireSystemDebt` to `LiquityBase` and replaced similar inline calculations with calls to those functions, thereby fixing the issue.

TOB-LQT-013: Troves can be improperly removed under certain circumstances which results in unexpected reverts
Fixed. In commit df289a9a239d04d3306fde4f9c42db0895f1b723 Liquity removed that `reInsert` does remove but not insert troves with `ICR == 0` making all future operations on that trove fail. Instead `reInsert` now reverts if `ICR == 0`, thereby fixing the issue.

TOB-LQT-014: Initial redeem can unexpectedly revert if no redeemable troves
Fixed. In pull request https://github.com/liquity/dev/pull/162/files Liquity adds the `require` that `getTCR() >= MCR` at the beginning of `TroveManager.redeemCollateral.` This implies that there is at least one trove with `ICR >= MCR` that can be redeemed from. This ensures `T.totalETHDrawn` will not be `0` and therefore `baseRate` will not be `0` as well. That fixes the issue. To ensure that the body of the trove redemption loop in `TroveManager.redeemCollateral` is executed at least once if the function doesn't revert before it, we recommend adding an assertion for that and running echidna again to check whether the assertion can fail.

TOB-LQT-015: Redeem without redemptions might still return success
Fixed. In pull request https://github.com/liquity/dev/pull/162/files Liquity adds a require that `getTCR() >= MCR` at the beginning of `TroveManager.redeemCollateral.` This implies that there is at least one trove with `ICR >= MCR` that can be redeemed from. This ensures `TroveManager.redeemCollateral` reverts if there are no redemptions, thereby fixing the issue.