Liquity

Security Assessment

March 25, 2021

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

Rick Pardoe | *Liquity*
rick@liquity.org

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

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

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

Changelog:

February 12, 2021:	Delivered initial report
March 2, 2021:	Added Appendix E (Fix Log)
March 25, 2021:	Added results of StabilityPool review

Executive Summary

From February 1 to February 12, 2021, Liquity engaged Trail of Bits to review the security of the Liquity protocol. Trail of Bits conducted this assessment over four person-weeks, with two engineers working from commit hash `8cec3fd` from the `liquity/dev` repository. Trail of Bits reviewed a previous version of this codebase in December 2020.

During the first week of the assessment, we familiarized ourselves with the changes introduced in this version of the codebase, focusing on the new recovery mode liquidation logic, the token lockup contract and its factory, and the Unipool contract. In the second week, we focused on the changes to borrower logic, the proposed gas optimizations, and the oracle components.

We identified three issues—two of medium severity and one of informational severity. The medium-severity issues involve situations in which the price feed may be allowed to report extreme price fluctuations. The informational-severity issue pertains to a situation in which a user could unexpectedly receive an error when trying to exit the staking pool. We discuss code quality concerns not related to any particular security issue in Appendix C. In Appendix D, we propose an alternative way to track the state of the oracles.

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

Trail of Bits recommends addressing the findings presented in this report, continuing to simplify the system's logic where possible, and developing a playbook for handling incident response, especially with respect to oracle degradation.

Update: On February 23, 2021, Trail of Bits reviewed fixes implemented by Liquity for the issues presented in this report. See Appendix E for a detailed review of each issue's current status.

Update: From March 15 to March 17, 2021, Trail of Bits performed a two-person-day review of the StabilityPool contract (from commit `2f05955`). This resulted in two additional low-severity issues (TOB-LQT-019 and TOB-LQT-020).

Project Dashboard

Application Summary

Name	Liquity
Version	`8cec3fd`, `2f05955`
Type	Smart Contracts
Platforms	Solidity

Engagement Summary

Dates	February 1 – February 12, 2021 March 15 – March 17, 2021
Method	Whitebox
Consultants Engaged	3
Level of Effort	4 person-weeks + 2 person-days

Vulnerability Summary

Total High-Severity Issues	0	
Total Medium-Severity Issues	2	■ ■
Total Low-Severity Issues	2	■ ■
Total Informational-Severity Issues	1	■
Total Undetermined-Severity Issues	0	
Total	5	

Category Breakdown

Data Validation	4	■ ■ ■ ■
Arithmetic	1	■
Total	5	

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 briefly explain our reasoning.

Category Name	**Description**
Access Controls	**Strong.** There were appropriate access controls for privileged operations performed by certain contracts.
Arithmetic	**Moderate.** The contracts used safe arithmetic. No overflows were possible in areas where these `SafeMath` functions were not used. However, it may be possible to trigger an underflow, which could cause a DoS during liquidation.
Assembly Use	**Strong.** The contracts used assembly only to fetch the `chainID` for the ERC2612 `permit` functionality.
Centralization	**Strong.** Ownership was renounced during the post-deployment setup process, and the system had a backup price oracle.
Contract Upgradeability	**Not Applicable.** The contracts contained no upgradeability mechanisms.
Function Composition	**Satisfactory.** Most of the functions and contracts were organized and scoped appropriately. Areas of improper function composition identified in the previous report have been somewhat improved.
Front-Running	**Moderate.** The Liquity team previously identified front-running scenarios that could occur in the stability pool or during price-fetching operations. No additional front-running concerns were identified.
Monitoring	**Satisfactory.** The events produced by the smart contract code were sufficient to monitor on-chain activity.
Specification	**Satisfactory.** The system had comprehensive documentation that was updated prior to or during our review.
Testing & Verification	**Strong.** The contracts included numerous unit tests and even used automated tools such as fuzzers.

Engagement Goals

The engagement was scoped to provide a security assessment of the changes made to the Liquity codebase in the `liquity/dev` repository since our December 2020 review.

Specifically, we sought to answer the following questions:

- Can the oracle be manipulated to fall back when it shouldn't?
- Have all possible combinations of oracle states been accounted for?
- Is it possible to evade the timelock in the token lockup contract?
- Can a trove adjustment move the trove or the system into an inconsistent state?
- Do the changes to liquidation logic introduce any cases that could allow troves in good standing to be liquidated?
- Can staker funds be locked in the staking pool?
- Are staking reward calculations sound?

Coverage

Our review focused on commit `8cec3fd`, changes to the PriceFeed contract in commit `dce38b7`, and a PR that introduced some gas optimizations. The following contracts in particular underwent significant changes:

TroveManager. The `TroveManager` contract exposed trove management functionality, including liquidation. Prior to this review, Liquity modified some recovery mode liquidation logic, removing the option to conduct partial liquidation processes. We reviewed this contract to ensure that it still properly executed liquidation transactions.

PriceFeed. The `PriceFeed` contract managed interactions with the system's two redundant price oracles and provided pricing data to the rest of the Liquity system. We reviewed this contract to ensure that it returned the appropriate oracle's price and properly handled all combinations of oracle states.

BorrowerOperations. This contract was an important entry point for the management of a trove's lifecycle as a borrower. The `_adjustTrove` function at the core of this contract underwent several changes. We reviewed these changes to ensure that the function correctly handled balance tracking and the recording of trove state changes.

Unipool. The `Unipool` contract incentivized early users of the pool to provide liquidity for a Uniswap LUSD/ETH pair by rewarding them with LQTY. We reviewed the contract to ensure that stakers' funds could not be locked, that there would always be a sufficient amount of rewards available for payouts, and that rewards were calculated correctly.

LockupContract & LockupContractFactory. These contracts managed the creation of the wallet contracts that lock up LQTY tokens to be vested after one year. We reviewed these contracts to verify that tokens would remain locked during the vesting period, that tokens would be available only to the beneficiary, and that the system would trust only factory-deployed lockup contracts.

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.
- Echidna, a smart contract fuzzer. Echidna can rapidly test security properties via malicious, coverage-guided test case generation.

Automated testing techniques augment our manual security review but do not replace it. Each method 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.

Automated Testing with Echidna

While reviewing the codebase, we identified various properties and invariants that should hold. Using Echidna, we implemented the following property-based tests:

Unipool Properties

ID	Property	Result
1	`Unipool.periodFinish()` monotonically increases.	PASSED
2	`Unipool.lastUpdateTime()` monotonically increases.	PASSED
3	`Unipool.rewardPerTokenStored() <= rewardPerToken()`.	PASSED
4	`Unipool.rewardPerTokenStored()` monotonically increases.	PASSED
5	`lastTimeRewardApplicable()` monotonically increases.	PASSED
6	`Unipool.rewardPerToken()` monotonically increases.	PASSED
7	The LQTY of every staker monotonically increases.	PASSED
8	The LQTY of `Unipool` monotonically decreases.	PASSED
9	`sum(Unipool.balanceOf(staker) for staker in stakers) == Unipool.totalSupply()`.	PASSED
10	The amount of pending rewards never exceeds the amount of LQTY to be paid out (i.e., `sum(Unipool.rewards(staker) for staker in stakers) <= LQTY.balanceof(Unipool)`).	PASSED

11	`4/3 million == LQTY.balanceOf(Unipool) + sum(LQTY.balanceOf(staker) for staker in stakers)`.	PASSED
12	`periodFinish != 0`.	PASSED
13	There is always LQTY available to be paid out during reward period (i.e., `0 < LQTY.balanceOf(Unipool) || seconds remaining in reward period == 0`).	PASSED
14	During the reward period, there is always enough LQTY available for payouts at the reward rate (i.e., `Unipool.rewardRate() * seconds remaining in reward period <= LQTY.balanceOf(Unipool)`).	PASSED
15	The reward rate is never exceeded (i.e., `sum(LQTY.balanceOf(staker) for staker in stakers) + sum(Unipool.reward(staker) for staker in stakers) <= Unipool.rewardRate() * seconds elapsed of reward period`).	PASSED
16	`Unipool.stake(x)` never errors out if `0 < x` and `x <= LQTYUniswapPair.balanceOf(msg.sender)` and `LQTYUniswapPair.allowance(msg.sender, Unipool)`.	PASSED
17	`Unipool.withdraw(x)` never errors out if `0 < x` and `x <= Unipool.balanceOf(msg.sender)`.	PASSED
18	If `Unipool.withdraw(x)` doesn't error out, `Unipool.balanceOf(msg.sender)` will be reduced by `x`.	PASSED
19	If `Unipool.exit()` and `Unipool.claimReward()` are successful, the `msg.sender` will no longer hold a stake, and the sender's LQTY balance will have increased.	PASSED
20	If `Unipool.exit()` is successful, `msg.sender` will no longer hold a stake, and the sender's `LQTYUniswapPair` balance will have increased by the amount of the sender's former stake.	PASSED
21	If `Unipool.claimReward()` doesn't error out, `msg.sender`'s stake will not change.	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

❑ **Make `claimReward` return early instead of reverting if `reward == 0`.** This will prevent users from becoming confused when they attempt to exit the pool. TOB-LQT-016

❑ **Add a `_bothOraclesLiveAndUnbrokenAndSimilarPrice` check to Case 4 in the `PriceFeed` before the system resumes using Chainlink if Tellor is not frozen.** This will prevent the price feed from reporting erroneous price swings. TOB-LQT-017

❑ **Update Case 5 in the PriceFeed contract such that it returns the `lastGoodPrice` if the price returned by Chainlink changed by more than 50%.** This will prevent the price feed from reporting erroneous price swings. TOB-LQT-018

❑ **Update the `StabilityPool` code such that the first call to `provideToSP` reverts if the passed-in `_frontEndTag` differs from the stored `frontEnd`.** Additionally, update the documentation to explain the `frontEnd` selection mechanism to users. TOB-LQT-019

❑ **Calculate the `LUSDLossNumerator` in the StabilityPool._computeRewardsPerUnitStaked only when necessary (i.e., move it inside the `else` block).** This will help prevent reverts caused by underflows. TOB-LQT-020

Long Term

❑ **Ensure that there are no unexpected reverts.** TOB-LQT-016

❑ **Enumerate scenarios in which one or both of the oracles could experience long-term degradation and develop a strategy for intervening (or deciding whether to intervene).** TOB-LQT-017, TOB-LQT-018

❑ **Try to prevent the system from silently ignoring function parameters.** This poor coding practice makes it more difficult for external parties to understand the implementation and increases the likelihood of bugs. TOB-LQT-019

❑ **Using Echidna, carry out property-based testing to ensure that unforeseen underflows cannot cause a temporary DoS in the system.** This will help prevent unexpected reverts. TOB-LQT-020

Findings Summary

#	Title	Type	Severity
1	exit reverts if there are no claimable rewards	Data Validation	Informational
2	Chainlink may report erroneous prices when unfrozen	Data Validation	Medium
3	Chainlink may report erroneous prices when Tellor remains untrusted after the last update	Data Validation	Medium
4	frontEnd mechanism may confuse users	Data Validation	Low
5	Underflow in `_computeRewardsPerUnitStaked` might cause DoS during liquidations	Arithmetic	Low

1. `exit` reverts if there are no claimable rewards

Severity: Informational
Difficulty: Medium
Type: Data Validation
Finding ID: TOB-LQT-016
Target: `Unipool.sol`

Description
If the liquidity pool lacks claimable staker rewards (which is possible under certain conditions), `exit` will always revert. Such cases are likely to surprise users and cause them to waste gas.

Exploit Scenario
There has been constant staking over a number of seconds, and no rewards have been accrued. Alice is unaware of this and stakes a certain number of liquidity pool tokens. Alice later calls `exit` to leave the pool, and the transaction unexpectedly reverts. This may be confusing to Alice, who will still be able to call `withdraw` to recover her tokens but will have lost the gas she paid for the `exit`.

Recommendation
Short term, make `claimReward` return early instead of reverting if `reward == 0.`

Long term, ensure that there are no unexpected reverts.

2. Chainlink may report erroneous prices when unfrozen

Severity: Medium
Difficulty: High
Type: Data Validation
Finding ID: TOB-LQT-017
Target: `PriceFeed.sol`

Description
When the `PriceFeed` is in the `usingTellorChainlinkFrozen` state (Case 4), if it detects that Chainlink has started working again, it will report that price without validating that it is in the expected range; it will then move the system into the `chainlinkWorking` state. In all other instances in which the system transitions to this state, it ensures that both oracles are reporting consistent prices (by calling `_bothOraclesLiveAndUnbrokenAndSimilarPrice`) before transitioning.

If Chainlink unfreezes but begins to report erroneous prices, the system will accept the prices; they will not be detected unless another large price swing occurs or Chainlink enters an untrusted state.

Exploit Scenario
Chainlink is frozen, and the system is using Tellor for pricing information. Chainlink resumes posting updates, but they include wildly inflated prices. On the next call to `fetchPrice`, the system detects that Chainlink has unfrozen and begins using it again. Due to the erroneous pricing data, the system undervalues users' collateral, and many users become subject to liquidation .

Recommendation
Short term, add a `_bothOraclesLiveAndUnbrokenAndSimilarPrice` check to Case 4 in the `PriceFeed` before the system resumes using Chainlink if Tellor is not frozen.

Long term, enumerate scenarios in which one or both of the oracles could experience long-term degradation and develop a strategy for intervening (or deciding whether to intervene).

3. Chainlink may report erroneous prices when Tellor remains untrusted after the last update

Severity: Medium
Difficulty: High
Type: Data Validation
Finding ID: TOB-LQT-018
Target: `PriceFeed.sol`

Description
When the `PriceFeed` is in the `usingChainlinkTellorUntrusted` state (Case 5) and Tellor is frozen or broken, `fetchPrice` will return the Chainlink price without validating whether it changed by >50% between the last 2 rounds. Usually, if Chainlink is used while Tellor is frozen or broken, the system performs this validation (as in Case 1).

Exploit Scenario
The system is using Chainlink for pricing information. Tellor is untrusted. Chainlink resumes posting updates, but they include wildly inflated prices, while Tellor remains broken or frozen. On the next call to `fetchPrice`, the system returns the inflated Chainlink price.

Recommendation
Short term, update Case 5 in the PriceFeed contract such that it returns the `lastGoodPrice` if the price returned by Chainlink changed by more than 50%.

Long term, enumerate scenarios in which one or both of the oracles could experience long-term degradation and develop a strategy for intervening (or deciding whether to intervene).

4. `frontEnd` mechanism may confuse users

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

Difficulty: High
Finding ID: TOB-LQT-019

Description
An undocumented cache of the `_frontEndTag` in `provideToSP` may lead users to misuse the function.

`provideToSP` has a `_frontEndTag` parameter representing the `frontEnd` to which LQTY gains should be paid out. When `provideToSP` is called, `_frontEndTag` is saved for the caller:

```
if (initialDeposit == 0) {_setFrontEndTag(msg.sender, _frontEndTag);}
uint depositorETHGain = getDepositorETHGain(msg.sender);
uint compoundedLUSDDeposit = getCompoundedLUSDDeposit(msg.sender);
uint LUSDLoss = initialDeposit.sub(compoundedLUSDDeposit); // Needed only for event log

// First pay out any LQTY gains
address frontEnd = deposits[msg.sender].frontEndTag;
_payOutLQTYGains(communityIssuanceCached, msg.sender, frontEnd);

// Update front end stake
uint compoundedFrontEndStake = getCompoundedFrontEndStake(frontEnd);
uint newFrontEndStake = compoundedFrontEndStake.add(_amount);
_updateFrontEndStakeAndSnapshots(frontEnd, newFrontEndStake);
emit FrontEndStakeChanged(frontEnd, newFrontEndStake, msg.sender);
```

Figure 4.1: `StabilityPool.sol#L327-L340`

However, if a subsequent call to `provideToSP` has a different `_frontEndTag`, the stored version will be used, and the new `_frontEndTag` will be ignored. The user may incorrectly assume that the new value will be used; as such, the `frontEnd` that receives LQTY gains may not be the one chosen by the user.

Exploit Scenario
Bob calls `provideToSP`, passing in the `_frontEndTag` of `frontEnd` A. Bob wants to make another deposit and switch to a different `frontEnd` (B). Bob calls `provideToSP` with `_frontEndTag` B. The call succeeds; however, it uses `frontEnd` A and silently ignores the passed-in `frontEnd` (B).

Recommendation
Short term, update the system such that the first call to `provideToSP` reverts if the passed-in `_frontEndTag` differs from the stored `frontEnd`. Additionally, update the documentation to explain the `frontEnd` selection mechanism to users.

Long term, try to prevent the system from silently ignoring function parameters. This poor coding practice makes it more difficult for external parties to understand the implementation and increases the likelihood of bugs.

5. Underflow in `_computeRewardsPerUnitStaked` might cause DoS during liquidation

Severity: Low
Difficulty: High
Type: Arithmetic
Finding ID: TOB-LQT-020
Target: `StabilityPool.sol`

Description
The rounding feedback mechanism of the `_computeRewardsPerUnitStaked` function might cause an underflow, leading the liquidation process to revert. The reversion could prevent certain liquidatable troves from being liquidated.

```
uint LUSDLossNumerator = _debtToOffset.mul(DECIMAL_PRECISION).sub(lastLUSDLossError_Offset);
uint ETHNumerator = _collToAdd.mul(DECIMAL_PRECISION).add(lastETHError_Offset);


if (_debtToOffset >= _totalLUSDDeposits) {
    LUSDLossPerUnitStaked = DECIMAL_PRECISION;  // When the Pool depletes to 0, so does each
deposit
    lastLUSDLossError_Offset = 0;
} else {
    /*
     * Add 1 to make error in quotient positive. We want "slightly too much" LUSD loss,
     * which ensures the error in any given compoundedLUSDDeposit favors the Stability Pool.
     */
    LUSDLossPerUnitStaked = (LUSDLossNumerator.div(_totalLUSDDeposits)).add(1);
    lastLUSDLossError_Offset =
(LUSDLossPerUnitStaked.mul(_totalLUSDDeposits)).sub(LUSDLossNumerator);
}
```

Figure 5.1: `StabilityPool.sol#L535-L549`

The `_computeRewardsPerUnitStaked` function is called from the `offset` function, which is called from the liquidation functions in the `TroveManager`. Whenever `_computeRewardsPerUnitStaked` is called, a rounding error "offset" is stored in `lastLUSDLossError_Offset`. During the next call of this function, `lastLUSDLossError_Offset` will be subtracted from `_debtToOffset.mul(DECIMAL_PRECISION)`. If this subtraction causes an underflow, the use of SafeMath will cause a revert, and the transaction will fail.

There are several safeguards, such as a minimum trove debt amount, that should prevent a very low `_debtToOffset` value from being used and causing an underflow. However, they do not eliminate that possibility.

Exploit Scenario

The TroveManager liquidates a trove, which saves a rounding “offset” value of `X`. The TroveManager tries to liquidate another trove with a debt, `Y`, such that `(Y * 1e18) < X`. The transaction reverts because of an underflow. As a result, the trove cannot be liquidated.

Recommendation
Short term, calculate the `LUSDLossNumerator` only when necessary, i.e. move it inside the `else`.

Long term, using Echidna, **carry out property-based testing to ensure that** unforeseen underflows cannot cause a temporary DoS in the system.

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 a 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 the order of operations
Undefined Behavior	Related to undefined behavior triggered by the program

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 users’ information is at risk; exploitation could pose reputational, legal, or moderate financial risks to the client.
High	The issue could affect numerous users and have serious reputational, legal, or financial implications for the client.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is commonly exploited; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of a complex system.
High	An attacker must have privileged insider access to the system, may need to know extremely complex technical details, or must discover other weaknesses 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 purposes
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 the 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.

`PriceFeed.sol`:

- **Use the style `a = b; c = d`, rather than `(a, c) = (b, d)`.** The former assignment style is easier to read and facilitates the detection of mistakes.

- **Use correct indentation levels to increase readability.** Line 202 is set off by an extra level of indentation.

- **Remove unnecessary code.** Certain code (`_storeChainlinkPrice(chainlinkResponse);`) in line 204 is redundant because it is also included in line 208; this code should be removed.

`Unipool.sol`:

- **This contract has few comments and minimal accompanying documentation.** More documentation would improve readability and help users understand the contract's expected behavior.

- **`_requireCallerIsLQTYToken` is not used.** Consider removing unused functions.

- **`_updateReward`'s special handling of `address(0)` is confusing.** `_updateReward` is called with `address(0)` in `_notifyRewardAmount`, and `_updateReward` then handles `address(0)` as a special case. This makes the code difficult to understand. To simplify the code, consider replacing `_updateReward(address(0))` with `rewardPerTokenStored = rewardPerToken(); lastUpdateTime = lastTimeRewardApplicable();` and removing `if (account != address(0))` from `_updateReward`.

`BorrowerOperations.sol`:

- **Consider removing the externally accessible `adjustTrove` function and enumerating the other expected means of explicitly calling `_adjustTrove`.** The parameters that trigger paths through `_adjustTrove` can be combined in many ways, and any changes could have unintended results.

`StabilityPool.sol`:

- **Consider renaming the variable `snapshot_P` as `P_Snapshot`.** Standardizing the contract's naming scheme would increase the code's readability.

- **Consider saving the result of calling `getDepositorETHGain()` in the `withdrawETHGainToTrove` function in a variable and reusing the result.** This would prevent `getDepositorETHGain` from performing the same operation twice, wasting users' gas.

- **Update the comment describing the e_2 formula at the top of the file such that it accurately reflects the code.**

```
e_2 = d_t * (S - S_t) / (P_t * 1e9)
```

Figure C.1: `StabilityPool.sol#L105`

```
e_2 = d_t * (S) / (P_t * 1e9)
```

Figure C.2: The corrected version of figure C.1

D. Recommendations for Simplifying `fetchPrice`

The `fetchPrice` function is almost 200 lines long and includes up to 3 levels of nested `if` statements. Its complexity makes the code difficult to understand and test and increases the likelihood of errors.

To reframe the function, split `fetchPrice` into three functions, each of which focuses on one of the following aspects of it:

1. Determining the next state of Chainlink
2. Determining the next state of Tellor
3. Determining the price to return

`fetchPrice` returns one of the following three prices:

1. The current price returned by Chainlink
2. The current price returned by Tellor
3. The `lastGoodPrice`, which reflects the last price returned by `fetchPrice`

The following logic determines the price to return:

1. If Chainlink is **trusted** and **live**, return the price from Chainlink.
2. Else if Tellor is **trusted** and **live**, return the price from Tellor.
3. Else return the last good price.

"Live" means not broken or frozen. A broken or frozen oracle has no current price; as such, no price can be returned from it. For security reasons, no price is returned from an oracle that is not **trusted**. Using Chainlink is preferable to using Tellor.

Currently, `fetchPrice` conflates the logic that determines whether to trust an oracle with the logic that determines which price to return. Separating these two aspects of the logic would make the code simpler and more focused—and simpler code tends to have fewer bugs.

Suggestions for implementing this separation and simplifying `fetchPrice` are `provided` below.

State Variables

Replace

```
   enum Status {
        chainlinkWorking,
        usingTellorChainlinkUntrusted,
        bothOraclesUntrusted,
        usingTellorChainlinkFrozen,
        usingChainlinkTellorUntrusted
    }

    // The current status of the PricFeed, which determines the conditions for the next
price fetch attempt
    Status public status;
```

Figure D.1: The current method of oracle-state tracking in `PriceFeed.sol`

with

```
    enum OracleState {
        TrustedAndLive,
        TrustedAndFrozen,
        NotTrusted
    }

    OracleState public chainlinkState;
    OracleState public tellorState;
```

Figure D.2: An alternative method of oracle-state tracking in `PriceFeed.sol`

Using two state variables instead of one increases gas costs, while encoding the variables into one state variable would reduce gas costs. We recommend making the encoding explicit through the `encodeOracleState` and `decodeOracleState` functions.

fetchPrice

The following alternative version of `fetchPrice` is just around 10 lines. The logic for deciding which price to return is obvious.

```
    function fetchPrice() external override returns (uint) {
        // Get current and previous price data from Chainlink, and current price data from
Tellor
```

```
        ChainlinkResponse memory chainlinkResponse = _getCurrentChainlinkResponse();
        ChainlinkResponse memory prevChainlinkResponse =
_getPrevChainlinkResponse(chainlinkResponse.roundId, chainlinkResponse.decimals);
        TellorResponse memory tellorResponse = _getCurrentTellorResponse();

        chainlinkState = _determineChainlinkState(chainlinkState, tellorState,
chainlinkResponse, prevChainlinkResponse, tellorResponse);
        tellorState = _determineTellorState(chainlinkState, tellorState, chainlinkResponse,
prevChainlinkResponse, tellorResponse);

        if (chainlinkState == OracleState.TrustedAndLive) {
            return _storeChainlinkPrice(chainlinkResponse);
        }
        if (tellorState == OracleState.TrustedAndLive) {
            return _storeTellorPrice(tellorResponse);
        }
        return lastGoodPrice;
    }
```

Figure D.3: An alternative version of `fetchPrice` in `PriceFeed.sol`

Determining the Next Oracle State

The extracted state transition logic is incorporated into the `view` functions `_determineChainlinkState` and `_determineTellorState`, each of which determines the next state of one oracle. Additional refactoring could make those functions `pure` and very easy to test.

```
    function _determineTellorState(
        OracleState chainlinkState,
        OracleState tellorState,
        ChainlinkResponse memory prevChainlinkResponse,
        ChainlinkResponse memory chainlinkResponse,
        TellorResponse memory tellorResponse
    ) internal view returns (OracleState) {
        bool isBroken = _tellorIsBroken(tellorResponse);
        bool isFrozen = _tellorIsFrozen(tellorResponse);

        if (tellorState == OracleState.TrustedAndLive) {
            if (isBroken) {
                return OracleState.NotTrusted;
            } else if (isFrozen) {
                return OracleState.TrustedAndFrozen;
            } else {
                return OracleState.TrustedAndLive;
            }
        else if (tellorState == OracleState.TrustedAndFrozen) {
            if (isBroken) {
                return OracleState.NotTrusted;
            } else if (isFrozen) {
                return OracleState.TrustedAndFrozen;
            } else {
                return OracleState.TrustedAndLive;
            }
        } else if (tellorState == OracleState.NotTrusted) {
            if (_bothOraclesLiveAndUnbrokenAndSimilarPrice(chainlinkResponse,
prevChainlinkResponse, tellorResponse)) {
                return OracleState.TrustedAndLive;
            } else {
                return OracleState.NotTrusted;
            }
        }
    }
```

Figure D.4: A proposed alternative for _determineTellorState in `PriceFeed.sol`

This code is intentionally kept simple and could be made even shorter. Because the code focuses just on the state transition logic of Tellor, the transition process of Tellor (such as from `NotTrusted` to `TrustedAndLive`) is clear.

The implementation of `_determineChainlinkState` would be similar and only slightly more complex.

E. Fix Log

On February 23, 2021, Trail of Bits reviewed the fixes and mitigations implemented by Liquity to address the issues identified in this report. For additional information, please refer to the detailed fix log.

#	Title	Severity	Status
1	`exit` reverts if there are no claimable rewards	Informational	Fixed
2	Chainlink may report erroneous prices when unfrozen	Medium	Fixed
3	Chainlink may report erroneous prices when Tellor is untrusted	Medium	Fixed

Detailed Fix Log

TOB-LQT-016: `exit` reverts if there are no claimable rewards
Fixed. Liquity identified this reversion as intended behavior and renamed the function to more clearly describe the user experience. (PR 295)

TOB-LQT-017: Chainlink may report erroneous prices when unfrozen
Fixed. Liquity updated the oracle logic to include a price check; when Chainlink unfreezes, the check verifies that the two oracles agree on the current price before they resume returning values reported by Chainlink. (PR 303)

TOB-LQT-018: Chainlink may report erroneous prices when Tellor is untrusted
Fixed. Liquity updated the oracle logic to include a check that prevents Chainlink from reporting wild price swings when it is the only oracle trusted. (PR 303)