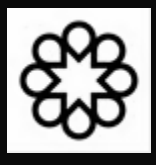
certora

Security Assessment & Formal Verification Report



Alluvial Finance - LiquidCollective

February 2024

Prepared for
Alluvial

Table of contents

Project Summary

Project Scope

Repo Name	Repository	Commits	Compiler version	Platform
LiquidCollective	https://github.com/liquid-collective/liquid-collective-protocol/tree/main/contracts/src	0b15fbf	Solidity 0.8.20	EVM

Project Overview

This document describes the specification and verification of the **Liquid Collective Liquid Staking protocol** using the Certora Prover and manual code review findings. The work was undertaken from **3 December 2023** to **8 February 2024**.

The following contract list is included in our scope:

`contracts/src/Initializable.sol`
`contracts/src/TLC.1.sol`
`contracts/src/OperatorsRegistry.1.sol`
`contracts/src/ELFeeRecipient.1.sol`
`contracts/src/WLSETH.1.sol`
`contracts/src/Firewall.sol`
`contracts/src/CoverageFund.1.sol`
`contracts/src/Allowlist.1.sol`
`contracts/src/Withdraw.1.sol`
`contracts/src/Oracle.1.sol`
`contracts/src/Administrable.sol`
`contracts/src/River.1.sol`
`contracts/src/RedeemManager.1.sol`
`contracts/src/TUPProxy.sol`

`contracts/src/libraries/LibSanitize.sol`
`contracts/src/libraries/LibAllowlistMasks.sol`
`contracts/src/libraries/LibBasisPoints.sol`
`contracts/src/libraries/LibErrors.sol`

contracts/src/libraries/LibAdministrable.sol
contracts/src/libraries/LibUnstructuredStorage.sol
contracts/src/libraries/LibBytes.sol
contracts/src/libraries/LibUint256.sol

contracts/src/state/tlc/IgnoreGlobalUnlockSchedule.sol
contracts/src/state/tlc/VestingSchedules.2.sol
contracts/src/state/tlc/VestingSchedules.1.sol
contracts/src/state/river/BalanceToRedeem.sol
contracts/src/state/river/Shares.sol
contracts/src/state/river/DepositContractAddress.sol
contracts/src/state/river/OperatorsRegistryAddress.sol
contracts/src/state/river/DepositedValidatorCount.sol
contracts/src/state/river/WithdrawalCredentials.sol
contracts/src/state/river/CLSpec.sol
contracts/src/state/river/ReportBounds.sol
contracts/src/state/river/AllowlistAddress.sol
contracts/src/state/river/CollectorAddress.sol
contracts/src/state/river/DailyCommittableLimits.sol
contracts/src/state/river/MetadataURI.sol
contracts/src/state/river/CLValidatorTotalBalance.sol
contracts/src/state/river/CoverageFundAddress.sol
contracts/src/state/river/CommittedBalance.sol
contracts/src/state/river/CLValidatorCount.sol
contracts/src/state/river/GlobalFee.sol
contracts/src/state/river/RedeemManagerAddress.sol
contracts/src/state/river/OracleAddress.sol
contracts/src/state/river/LastConsensusLayerReport.sol
contracts/src/state/river/BalanceToDeposit.sol
contracts/src/state/river/LastOracleRoundId.sol
contracts/src/state/river/SharesPerOwner.sol
contracts/src/state/river/ELFeeRecipientAddress.sol
contracts/src/state/oracle/ReportsVariants.sol
contracts/src/state/oracle/LastEpochId.sol
contracts/src/state/oracle/ReportsPositions.sol
contracts/src/state/oracle/OracleMembers.sol
contracts/src/state/oracle/Quorum.sol
contracts/src/state/shared/Version.sol
contracts/src/state/shared/ApprovalsPerOwner.sol
contracts/src/state/shared/AdministratorAddress.sol

contracts/src/state/shared/PendingAdministratorAddress.sol
contracts/src/state/shared/RiverAddress.sol
contracts/src/state/operatorsRegistry/ValidatorKeys.sol
contracts/src/state/operatorsRegistry/TotalValidatorExitsRequested.sol
contracts/src/state/operatorsRegistry/CurrentValidatorExitsDemand.sol
contracts/src/state/operatorsRegistry/Operators.2.sol
contracts/src/state/operatorsRegistry/Operators.1.sol
contracts/src/state/allowlist/Allowlist.sol
contracts/src/state/allowlist/DenierAddress.sol
contracts/src/state/allowlist/AllowerAddress.sol
contracts/src/state/redeemManager/BufferedExceedingEth.sol
contracts/src/state/redeemManager/RedeemDemand.sol
contracts/src/state/redeemManager/WithdrawalStack.sol
contracts/src/state/redeemManager/RedeemQueue.sol
contracts/src/state/slashingCoverage/BalanceForCoverage.sol
contracts/src/state/wlseth/BalanceOf.sol
contracts/src/state/migration/OperatorsRegistry_FundedKeyEventRebroadcasting_OperatorIndex.sol
contracts/src/state/migration/OperatorsRegistry_FundedKeyEventRebroadcasting_KeyIndex.sol

contracts/src/components/SharesManager.1.sol
contracts/src/components/OracleManager.1.sol
contracts/src/components/UserDepositManager.1.sol
contracts/src/components/ERC20VestableVotesUpgradeable.1.sol
contracts/src/components/ConsensusLayerDepositManager.1.sol

contracts/src/migration/TLC_globalUnlockScheduleMigration.sol

contracts/src/interfaces/IAllowlist.1.sol
contracts/src/interfaces/ITLC.1.sol
contracts/src/interfaces/IWithdraw.1.sol
contracts/src/interfaces/ICoverageFund.1.sol
contracts/src/interfaces/IDepositContract.sol
contracts/src/interfaces/IOracle.1.sol
contracts/src/interfaces/IELFeeRecipient.1.sol
contracts/src/interfaces/components/ISharesManager.1.sol
contracts/src/interfaces/components/IUserDepositManager.1.sol
contracts/src/interfaces/components/IOracleManager.1.sol
contracts/src/interfaces/components/IConsensusLayerDepositManager.1.sol

`contracts/src/interfaces/components/IERC20VestableVotesUpgradeable.1.sol`
`contracts/src/interfaces/IFirewall.sol`
`contracts/src/interfaces/IWLSETH.1.sol`
`contracts/src/interfaces/IAdministrable.sol`
`contracts/src/interfaces/IOperatorRegistry.1.sol`
`contracts/src/interfaces/IRiver.1.sol`
`contracts/src/interfaces/IRedeemManager.1.sol`

The Certora Prover demonstrated the implementation of the Solidity contracts above is correct with respect to the formal rules written by the Certora team. In addition, the team performed a manual audit of all the Solidity contracts. During the verification process and the manual audit, the Certora Prover discovered bugs in the Solidity contracts code, as listed below. As Alluvial was tasked with addressing the findings of this report on behalf of Liquid Collective, Alluvial's responses are also included.

Findings Summary

The table below summarizes the findings of the review, including type and severity details.

Severity	Discovered	Acknowledged	Code Fixed
Critical	1	1	1
High	2	2	1
Medium	4	4	2
Low	5	4	2
Informational	3	3	1
Total			

Detailed Findings

High Severity Concerns

Crit-1. Malicious operator could bypass WithdrawCredentials check via a Frontrun directly to the deposit contract

Severity: Critical
Contract: *OperatorsRegistry, protocol Design*
Impact: Critical - Direct drain of user funds. Effectively Uncapped (could be as large as the funds waiting to be deposited).
Probability: High - A time where there's an exceptionally high traffic of the protocol, could turn even a trusted party malicious. If no checks enforce this beforehand, it's almost bound to happen
Note: This issue was reported by a white Hacker to the AlluvialTeam during the initial time of the project. We've helped review the finding, and access the proposed possible solutions that Alluvial team's suggested to implement.
Description: To recap, the WithdrawCredentials (WC) is the address to which a validator's rewards and funds are sent once they request to skim or exit on the beacon chain. The WC of each Validator in this protocol are part of the validator data that is saved in the registry. Those details are later vetted by the admin (by increasing the operator's limit count) before the protocol could deposit them.
Those WC are actually set on the beacon chain once the validator made its first deposit, and on subsequent deposits this field is ignored.
The protocol itself doesn't force a registered validator to make a deposit (before vetting), and the blockchain history shows that validators used by the protocol had their first deposit from the protocol itself (meaning, they were susceptible to such a front-run).

Recommendation: While we discussed the specifics with the alluvial team in greater detail, in summary in order to reduce the (current) full trust on the Node Operators, and associated risk from it, we recommend that keys be vetted only once the WC is already initialized and can be properly checked. This can be either achieved via completely off-protocol deposits by the operator's initiative, or by the protocol implementing a mechanism to send the minimal deposit to each registered validator that is desired for getting vetted.

Liquid Collective's response:
This frontrun has been acknowledged, and the risks will be mitigated with some changes, as described in the following document. We are planning on opening up the document, and turning into a

blog post, the plan is to initially deploy the Trustful Solution, option 1, with a view to making the setup less trusted over time.

The summary of the proposed solution is as follows:

Trustful solution

This solution implements some minor changes to Liquid Collective's deposit flow and a major change to the protocol's keeper, which triggers the deposit. In this approach the duty of verifying that a validator is not compromised is entrusted to the keeper, a daemon that initiates the transactions to deposit ETH to the consensus layer). The security of this approach is dependent on the assumption that the keeper, via off-chain calls to the consensus layer, correctly validates the state of the validators at a given deposit data root, and that that deposit root is still the deposit root that is available on the Execution Layer.

Here is what needs to be done to implement this solution:

1. Modify *depositToConsensusLayer* to be protected such that it is callable by a trusted entity only & accepts a deposit data root
2. Modify *OperatorRegistry* to provide a view-only *pickNextValidator* function
3. Modify Keeper's processValidatorToDeposit to:
    a. Check whether the keys being provided by read-only *pickNextValidator* function are uncompromised (no existing deposit with invalid withdrawal credentials)
    b. Call the depositToConsensusLayer function by passing the deposit data root

Pros: Relatively easy to implement
Cons: Entire trust is on the entity running the keeper as identified by the Liquid Collective, whereas now anyone can call this method

When: This solution is planned for the upcoming audit changes

This is the PR that shows the fixes:

https://github.com/liquid-collective/liquid-collective-protocol/pull/249

Fix review: Fixed

H-1. User Deposit can steal a day's rewards via frontrun to the report update

Severity: High
Contract: *River.sol (userDepositManager.sol), protocol Design*
Impact: High - Griefing rates / Draining partial rewards from other users
Probability: Medium - could occur only on specific conditions, and requires substantial capital, but also any allowed user could do that, and it doesn't involve any additional risk to the attacker/griefer
Description: Any (allowed) user could make a huge deposit in order to mint a huge amount of LsETH, that would get most of the following report(s) rewards.
While this might sound intended, that user could request to exit with most of those funds - thus gaining rewards on the expense of others while their ETH didn't have to work (as not the entire amount can become committed to deposit).

Detailed Scenario: On a particularly profitable day, a user with a high amount of available capital could deposit into the protocol to mint a significant amount of LsETH. This deposit could happen right before the report, and the redeem request for most of those funds could happen right after the report submission (front and back run it respectively).
If the report's mode (or the following report) allows for withdrawals from deposit buffer, only the DAILY_DEPOSIT_LIMIT amount would get committed and the rest could be later withdrawn without going through the "full route" and delay.

This would give favorable rates and rewards to this user, compared to the rest of the users, relative to what otherwise everyone should've gotten.

This means that there would be a threshold that depends mainly on that report's profitability and TVL, and then on how empty the redeem queue is and the ratio between the amount a user could muster to deposit to the daily committed staking limit. Once this threshold is passed, any user is incentivized to do that deposit. In other words, the only cost for them is the liquidity - and given enough protocol earnings, that might be worth the short wait.

Recommendation: this could be a tricky issue to solve, as the report evaluation is a constraint of Ethereum design. Consider adding a daily user deposit limit (especially if you already implement a daily limit to the committed amounts).
Alternatively, consider changing mint logic of LsETH to mint only for committed funds at commit time (which could be more complicated - as it probably means implementing an additional logic layer of a deposit queue).

Liquid Collective's response:
This frontrun has been acknowledged. The risks will be mitigated with some changes to minimize the impact of this frontrun. Given the stage of the protocol development and the permissioned nature of who can Mint and Redeem, we are comfortable minimizing the impact of this exploit as opposed to fully alleviating the issue.

The plan is to:

1. Set a global cap on the amount of LsETH that can be minted on a daily basis.
2. Increase the cap set on the amount of ETH that can be deposited to the consensus layer by the protocol.

When: Given that this is an architectural change to the protocol, this is something that will be worked on, and will be released in a subsequent version of the protocol.

H-2. Blacklisted users can still claim their redeem requests

Severity: High
Contract: *RedeemManager.sol*
Impact: High - important protocol functionality broken - the KYC feature that is central to the protocol's purpose, can be partially bypassed
Probability: Medium - Can only be triggered by a user that is already allowed/whitelisted, and can do its malicious acts and request a full withdrawal at once. This could limit profitability or execution window of potential vectors, but besides that it's always possible
Description: Once a user requested redeem, their LsETH is transferred to the RedeemManager and they would no longer ever be checked against the blacklist. It means that any user that knows they're about to be blacklisted, can request full redeem and they would be able to access the withdrawn funds once they'll be satisfied with a matching withdrawEvent. This could also be executed in a front-run.
Note: This might not seem like a commonly high-rated issue, but in this protocol the KYC feature is tied strongly to the protocol purpose and target audience, thus strongly expected to not be broken. Furthermore, while whitelisting there might be some reliance that any bad actor could be punished by the admin that would freeze their funds, so the fact that a malicious user could avoid that is impacting this design in a meaningful way.

Recommendation: Add a check against the AllowList contract also on the claim() function, for both the msg.sender and the recipient addresses.
Additionally, it would be required to store the msg.sender of the call to requestRedeem() into *RedeemRequest's* struct as well, and check that too against the allowlist. (i.e., check that the original

request's funds owner wasn't blacklisted, o/w they can create temporary addresses for the recipient address and claim()'s caller).

Liquid Collective's response:

This is a great find, thank you. The team has put work into analyzing how best to approach this situation. Details of this can be found internally. In terms of preserving the intended use of the DENY_MASK, the claim method will also ensure that wallets are not DENIED.

We have implemented the fixes, and they have come in three separate PRs:
https://github.com/liquid-collective/liquid-collective-protocol/pull/261 ,
https://github.com/liquid-collective/liquid-collective-protocol/pull/267 , and
https://github.com/liquid-collective/liquid-collective-protocol/pull/274 , and
https://github.com/liquid-collective/liquid-collective-protocol/pull/294

Fix review: Fixed

Medium Severity Concerns

M-1. Recovery of some users in the queue could be less than it should due to batching

Severity: Medium
Contract: *RedeemManager.sol*
Impact: High - inaccurate socialization of recovery/losses - Partial amount is lost for some users, and distributed among the rest that are still in the protocol or later in the queue
Probability: Low - requires a period of time with significant slashing, and gains, in which there were no or not enough exits, after which the exit batches together both periods in the same withdrawal event
Description: The leftover funds that river passes to the RedeemManager (withdraw queue) would only be processed on the following report. If a withdrawal event on a particular report could span several groups of redeem requests which have different rates, some of those leftover funds could be used to be redistributed to the one of the following batches that were already covered.

In other words, separating the Withdraw event into two withdraw events of the same sum, wouldn't yield the same result for some users in the queue. It is assumed that the share rate can only be influenced by the oracle report and deposits, but it is also affected by the exceeding ETH recovered from the redeemManager on the report update.

Numerical Example: (exaggerated to exemplify):

i. River Starting state - 1000 LsETH exits with 1000 ETH inside the system (deposited). Rate = 1.0
ii. User 1 requests to leave with 100 LsEth. Rate=1.0.
    1. Queue holds 100 LsETH, and an effective ETH demand of 100.
iii. 500 ETH Rewards arrive, but no exits yet. Rate=1.5
iv. User 2 requests to leave with 100 LsETH (150 eth). Rate=1.5
    1. Queue holds 200 LsETH, and an effective ETH demand of 250.
v. Some big slash happened prior to the exit report, of 300 ETH. Now the rate is 1.2
vi. A withdrawal event is reported with just enough LsETH, of 240 ETH. Rate=1.2
vii. In this case - both users are satisfied. User1 gets 100 ETH, user2 gets 120 eth.
viii. If we separated the 240 ETH into two withdrawal events, such that the first only fully satisfies the first user, the 2nd user would get something else. Examine:
ix. Event1 gives enough ETH for 100 LsETh (meaning 120). User1 is then satisfied with 100 ETH, and 20 ETH is leftover.
x. By Event2 we calculate the rate again, now having 1120 ETH for 900 LsETH. Rate= 1.244444…

xi. User2 is satisfied with 124.4444 ETH.

Recommendation: Consider changing the oracle off-chain calculation logic to be such that it would be aware of the state of the RedeemManager queue, and wouldn't batch together in the next fulfillment (withdraw Event) requests that are under the current rate with those that are over it. This could cause some delays, but on the flipside it's likely to not increase the already existing delay, as it'll be possible to always at least fully clear one "oracle report epoch" on each report.
Another more complete solution would be to implement logic on the oracle report and River for the creation of several withdrawEvents from a single report, at points/amounts and rates that the oracles calculate off-chain (e.g. doing the *_reportWithdrawToRedeemManager()* step multiple times in a single report).

Liquid Collective's response:
This is a great finding, acknowledged. Given the Dencun upgrade to Ethereum, some changes to the way that the Oracle works to make it more trustless in nature are being explored.
Exploration is underway to investigate when it will be most appropriate to address this issue, and this will be factored into Liquid Collective's Oracle Daemon roadmap.

When: This won't be fixed/addressed in the subsequent release.

M-2. Users could lose their deposit to credit the Firewall contract, due to the receive() logic

Severity: Medium
Contract: *Firewall.sol*
Impact: Medium - User funds temporarily locked. Could be claimed by any other permissioned user via front-run after the mistake.
Probability: Medium Users might try to use that protocol's feature
Description: River has a default receive function that allows users to deposit that way (instead of calling deposit()). When River would be placed behind a Firewall contract, and given that firewall's receive() functionality forwards a normal call, it would render this feature to the firewall contract depositing into River.
In a similar manner, via a fallback, a permissioned user could cause the Firewall to exit and then free those funds (and select an arbitrary recipient).

Scenario Example: For this issue to happen, there's a specific setup/configuration scenario to be assumed (and a likely user error on top of that). Let's assume River.sol is configured behind a

firewall.sol contract (e.g., firewall.sol is whitelisted, and a bunch if not all of the users would only be able to access it that way).
In such a case, the front end would need to construct the deposit requests in a particular manner, which works.
But any user that might check the code of River (or due to the fact that River itself allows this as a functionality), they would see that recieve() should give them a normal deposit flow.
In the case users (by accident or by misunderstanding) somehow send it to the firewall's contract (e,g they look up the last address their wallet sent funds to), it'll trigger firewall's receive() and make a normal call, crediting the firewall instead.
For clarification, steps:

- Expected Flow:
    - User send funds to river (or other entrypoint) address
    - receive() function of river is triggered, user is msg.sender
    - User is credited LsETH
- Unexpected Flow:
    - User sends funds to the firewall before the river address
    - receive() function triggered on Firewall.sol, user is msg,sender
    - Normal call is forwarded to River.sol
    - receive() function is triggered on River, Firewall is msg.sender
    - Firewall is credited with LsETH

Recommendation: Revise and better document how the firewall contract is intended to be used. Even if it's only for the admins, it is hard to imagine a case where they'd intend the firewall to be credited with funds. The simplest solution might be removing the receive() functionality from firewall.sol, unless it's explicitly desired.

Liquid Collective's response:
Acknowledged Won't fix, we believe that the risk is minimal.

This risk is mitigated by ensuring the configuration of Firewall and Allowlist are always as follows:

1. The admin & executor of the Firewall contact will never get deposit permissions on Allowlist.

We will add this to our backlog, and revisit if we feel like we want to redeploy the firewall contract in the future.

M-3. Quorum documented invariant isn't enforced

Severity: Medium
Contract: *Oracle1.sol*

Impact: Medium - intended oracles quorum has no lower-bound.
Probability: Medium - only the admin can set the quorum to an unsatisfying low value.
Description: In the code-documentation above the function _clearReportsAndSetQuorum() it explicitly says (line 231-232):

```
The quorum value Q should respect the following invariant, where O is oracle member count
(O / 2) + 1 <= Q <= O
```

That comment implies both a lower and upper bound for the quorum size with respect to the current oracle member count, while only the upper bound is preserved.

Recommendation: Modify the revert statement / add an appropriate require statement that manifests both conditions.

Liquid Collective's response:

After consideration, and at this stage of the protocol's development, the documentation will be changed to reflect the desired behavior. The comment was incorrect; the code is as desired.

When: In the next protocol release.

Fixed https://github.com/liquid-collective/liquid-collective-protocol/pull/260/files

Fix review: Fixed

M-4. Deposits could be broken when totalSupply is 0, due to an assumption on initial ratio

Severity: Medium
Contract: *SharesManager.sol (River)*
Impact: Critical - Deposits to the Protocol would be completely broken, users would lose funds attempting deposits in that state
Probability: Very low - There aren't normal ways for users to burn their funds, so getting to this edge case via a "first depositor/user" is more likely only due to a malicious oracle consensus before any real user entered, or could happen normally after ALL the system's users have quit and their shares burned.
Description: on *_mintRawShares()*, the check for the initial state is done by querying the *assetBalance()* prior to the deposit, and o/w would calculate the proportional ratio. For the initial state, that's true. But it's assumed that the system can't reach a state with 0 totalSupply, and some balance in the internal accounting (due to rounding, for example).
This assumption is wrong, and although it's hard to manipulate directly, it could happen naturally for example in the case all the existing users have exited the system and received their funds (after some

time of operation, to allow for changes in rates and rounding errors to occur).
In this state, when totalSupply() is 0, the calculations of sharesToMint would always be zero multiplied by something, resulting in zero shares minted.

Affected code:

```
Unset
if (oldTotalAssetBalance == 0) {
      sharesToMint = _underlyingAssetValue;
      _mintRawShares(_owner, _underlyingAssetValue);
    } else {
     sharesToMint = (_underlyingAssetValue * _totalSupply()) /
oldTotalAssetBalance;
      _mintRawShares(_owner, sharesToMint);
    }
```

Recommendation: Consider what's the expected behavior in this scenario, and add this to the control flow. One example could be to consider *totalSupply()*==0 as an initial case, which would gift the residual funds to the first user to come, and put it in an OR operator to the existing condition (E.g., `oldTotalAssetBalance == 0 || _totalSupply() == 0`)

Liquid Collective's response:

Good find, thank you. This issue will be fixed as per the recommendation.

When: This will be included in the upcoming audit changes.

https://github.com/liquid-collective/liquid-collective-protocol/pull/256/

Fix review: Fixed

Low Severity Concerns

L-1. Withdrawal Event of 0 could be created in some cases, which leads to revert of some claim requests

Severity: Low
Contract: *RedeemManager.sol / River.sol*
Impact: Medium - temporary DOS for some users, to claiming user funds, but recoverable via manually crafting claim transactions
Probability: Low - requires some inactive days of the protocol, or broad delays in withdrawals for the operators although there is a demand in the withdrawal queue.
Description: A withdrawal Event of 0 amount (0 LsETH shares) could be created in the redeemManager, in the case that River's residual ethToRedeem balance is lower than a single share's worth. Then, if claimed "naively", any user's redeemRequest that is overlapping over several events and also a 0 amount event, would get a revert if they're trying to claim the entire request with a high iteration depth (and even if it's fully claimable just by the accounting amounts). That's due to a division by zero on the 0 shares event step.

Said affected users could use a 0 or smaller iteration depth on their claiming calls, and construct them such that the supplied EventIDS would skip the 0 withdrawalEvent ID.

Recommendation: If such an event is not desired to be created, then it would be better to explicitly check that either in the redeemManager (and then prepare for the revert on River), or on River itself (as it already checks if there's no ETH to send, but it didn't take into account the rounding down).

Liquid Collective's response:
Acknowledged; won't fix in the current release. We believe that this is not worth remediating due to the fact that this edge case will only be triggered when the residual ethToRedeem balance is lower than a single share's worth. Furthermore, user's will not be blocked, as long as they craft their transactions accordingly with a depth of 0, as per the description above.

When: This will be added to our backlog and we will revisit this in the future.

L-2. Round robin of node operators could be gamed by the operator at the last index, to always be on the most favorable count of the deviation threshold

Severity: Low
Contract: *OperatorsRegistry.sol*
Impact: Low - slight higher of reward shares might be gained, but within the accepted threshold
Probability: Medium-low - Incentive is there w/o any risk, but gains might be relatively low and also risk getting punished on off-chain.
Description: The round robin for choosing more validators to join or exit always starts from the first index. Thus, up to the tolerated threshold (currently 5 validators hard coded), the operator at the last index can make more deposits or exit requests such that their position would retain more active validators.
Note: The reward distribution between the operators happens outside the protocol (or possible off-chain completely). Such calculation could mitigate the above, but assuming a per-active share proportion of the rewards, this could make one operator slightly more favorable than another.

Recommendation: Either make sure the reward calculation renders this behavior irrelevant, or update the on-chain logic of the round robin to rotate the starting index (starting from the previous last index reached).

Liquid Collective's response:

Acknowledge, won't fix. We don't feel like this is something that needs addressing. As the way the protocol is architected, over time, our round robin algorithm is fair. This is due to the fact that the order in which operators are deposited to, via the round robin, is the same order from which they are exited from, and as a result the system is deemed "fair over time".

L-3. paused() function in TUPProxy.sol could be view function, and shouldn't have an access control restriction

Severity: Low
Contract: *TUPProxy.sol#L41*
Impact: Very Low - some functionality can't be gained, e.g. protocol integrations that check that
Probability: Medium - Constantly happens all the time, but querying this particularly in an integration doesn't seem all too important or likely either.
Description: As seen in the following snippet, the paused() function isn't a view, and has an ifAdmin modifier. This means no one but the configured admin could query on-chain whether the protocol is

paused via that function. Also it's likely many naive front-ends wouldn't be able to query that either.

```
Unset
function paused() external ifAdmin returns (bool) {
    return StorageSlot.getBooleanSlot(_PAUSE_SLOT).value;
  }
```

Recommendation: Make it a view function and remove the ifAdmin modifier.

Liquid Collective's response:

Acknowledge, and this has been fixed. That said, this won't be deployed to mainnet, as it would require a change to one of the protocol's proxy addresses.

https://github.com/liquid-collective/liquid-collective-protocol/pull/257

Fix review: Fixed

L-4. Wrong check in `SharesManager.1._sharesFromBalance()`

Severity: Low
Contract: SharesManager.1.sol#L211-L219
Impact: Medium - DOS due to divide by zero
Probability: Very Low - `_assetBalance()` shouldn't be equal to zero if `_totalSharesValue` isn't zero
Description:
The zero check inside _sharesFromBalance() is exactly the same as in _balanceFromShares(). However, given that `Shares.get()` is here in the numerator and not in the denominator, the function would already be returning a zero instead of reverting unexpectedly, making this check redundant.
This suggests a copy-paste mistake.
Recommendation: Check that the actual denominator isn't zero and return zero otherwise.

```
Unset
211:     function _sharesFromBalance(uint256 _balance) internal view
returns (uint256) {
- 212:         uint256 _totalSharesValue = Shares.get();
+ 212:         uint256 _totalUnderlyingSupply = _assetBalance();
213:
- 214:         if (_totalSharesValue == 0) {
+ 214:         if (_totalUnderlyingSupply == 0) {
215:             return 0;
216:         }
217:
- 218:         return (_balance * Shares.get()) / _assetBalance();
+ 218:         return (_balance * Shares.get()) /
_totalUnderlyingSupply;
219:     }
```

Liquid Collective's response:
Acknowledged, will fix.

When: This will be included in the upcoming audit changes.

https://github.com/liquid-collective/liquid-collective-protocol/pull/255

Fix review: Fixed

L-5. LsETH:ETH conversion rate is not stable due to rounding

Severity: Low
Contract: *River.sol, SharesManager.sol, OracleManager.sol*
Impact: Low - A property of the system (see River-5), where the conversion rate stays stable for each mint or burn in the system could be violated - as rounding errors accumulate to more than just 1 unit. Also, the direction the rounding favors could be both ways.
Probability: Medium - Those rounding errors are expected to happen frequently.
Description: The minting and burning of shares in the system adopts the pro-rata value calculation, such that the share rate (total underlying / total share supply) should be preserved when no external

rewards penalties/rewards are introduced. The method of the minted/burnt shares calculation is supposed to guarantee that the rounding errors favor the protocol, and the share price cannot decrease by more than one unit.
While this is true for minting shares (or their transfer) between any two parties, this is not the case for burning shares. For deposit and transfer, the share rate or any share value cannot increase by more than one uint after any operation (but can decrease by one share value due to rounding errors).
For shares burning, we were able to find, and prove, the upper bound on the share price loss of value.
Given:

X - Underlying supply of ETH (> 0)
S - Total supply of shares (> 0)
P_0 = floor(X / S) - initial share price

and redeeming an arbitrary amount of ETH (Δx) against (Δs) shares such that the price after is

P_1 = Δs == S ? 0 : floor((X - Δx) / (S - Δs))

The maximum loss of value is:

$$\Delta P = P_0 - P_1 \leq min(P_0, max(2, \frac{P_0^2}{X - \Delta x} + 1)) .$$

Usually the redeem amount is much smaller with respect to the total underlying supply (X >> Δx) so that effectively the maximum change is bound to 2. However, in the case that almost all of the ETH in the system is redeemed (X ~ Δx), the share price can change drastically.

Recommendation:
In the case where the theoretical limits described above pose a real issue, it's possible to slightly alter the burning mechanism such that the share price should not decrease in that manner, where rounding errors are in favor of the protocol. Alternatively, one can use a fixed-point math library for the shares-to-underlying conversions for accuracy improvement.

Liquid Collective's response:
Acknowledged, having had a look at your recommendation, we don't quite understand the proposal. It looks like the protocol is currently rounding in favour of the protocol, if you could elaborate a bit more here that would be great.

We have also worked through the algebra here, we feel like this edge case is only triggered in a rather extreme scenario whereby S^2/2. Find below our workings.

Any pointers here would be much appreciated, but if our workings are correct, we feel like this is an extreme edge-case.

Fix review: After discussion with the team we think this is an unrealistic edge case. Therefore there is no need for a fix.

$$P_0 = \frac{X}{S} \quad \begin{matrix} - \text{ Total ETH} \\ - \text{ Total LSETH} \end{matrix}$$

$$1 + \frac{P_0^2}{X - \Delta x} > 2$$

$$\Leftrightarrow \quad \frac{P_0^2}{X - \Delta x} > 1 \qquad [X - \Delta x > 0]$$

$$\Leftrightarrow \quad P_0^2 > X - \Delta x$$

$$\Leftrightarrow \quad \left(\frac{X}{S}\right)^2 > X - \Delta x \qquad [S^2 > 0]$$

$$\Leftrightarrow \quad X^2 > S^2(X - \Delta x)$$

$$X^2 - S^2 X + S^2 \Delta x > 0$$

$$X > \frac{S^2 + \overbrace{\sqrt{S^2(S^2 - 4\Delta x)}}^{>0}}{2} > \frac{S^2}{2}$$

$$\hookrightarrow \quad \frac{P_0^2}{X - \Delta x} + 1 > 2 \quad \Leftrightarrow \quad X > \frac{S^2}{2}$$

Extremely unlikely

Informational Concerns

I-1. RedeemManager's claimRequests could revert for the entire batch if one recipient doesn't accept ETH

Severity: Info
Contract: *RedeemMananger.sol*
Description: While in most cases, it would just be a user that makes their own TX revert, this may cause some issue with other protocol contracts that try to integrate with River/LiquidCollective in a particular manner.

Recommendation: If such a case is to be avoided by potential contract integrations, we suggest adding a recommendation in the docs for those other protocols to either rely on the redeemManager's logic directly (users withdraw directly from the RedeemManager), or implement their own withdraw Queue.

Liquid Collective's response:
Acknowledge, we will call this out in the relevant part of our documentation.

Users of the protocol should be aware that if they are sending their redemptions to a Smart Contract, that this Smart Contract must be able to receive ETH.

We have created a ticket to track this work in our backlog.

I-2. The regular ReentrancyGuard from OZ is not designed to work with upgradeable contracts created using a proxy pattern

Severity: Info
Contract: *WLsETH.sol*
Description: While everything functions as intended, this version of ReentrancyGaurd uses a regular storage slot and wouldn't be initialized in the same manner. It can also cause potential storage conflicts in the future, if one would assume the entirety of this contract's storage layout is unstructured like the rest of the variables.

Recommendation: Use OZ's ReentrancyGuardUpgradeable.sol instead of the regular ReentrancyGuard

```
Unset
File: WLSETH.1.sol

- 04: import
"openzeppelin-contracts/contracts/security/ReentrancyGuard.sol";

+ 04: import {ReentrancyGuardUpgradeable} from
"openzeppelin-contracts-upgradeable/contracts/security/ReentrancyGuar
dUpgradeable.sol";

...

- 19: contract WLSETHV1 is IWLSETHV1, Initializable, ReentrancyGuard {

+ 19: contract WLSETHV1 is IWLSETHV1, Initializable,
ReentrancyGuardUpgradeable {
```

Liquid Collective's response:
Acknowledge, this contract is currently not deployed, we will fix it in the repo for when we do decide to deploy this function.

When: We plan to get this in the audit changes.

https://github.com/liquid-collective/liquid-collective-protocol/pull/258

I-3. Internal returned array sizes in Operators.sol are too big

Severity: Info
Contract: *Operators.1.sol, Operators.2.sol*
Description: While things function as intended, the internal array that is returned for the *getAllExitable()* or the *getAllFundable()* functions could have a larger size than needed. (The extra members are not iterated over in the OperatorsRegistry contract, but it could cause trouble for future features if not iterated carefully). Also a fix to this would make it needless to return the size of the copied members, as it'd be possible to query the array's length.

Recommendation: For arrays that are initialized at an excessive size, it's possible to truncate them with the following before returning them:

Unset
```
    assembly ("memory-safe") {

        mstore(dummyArray, finalLength)

    }
```

So per instance:
herefore, the following arrays can be returned at their relevant size:

- OperatorsV2.getAllFundable()

Unset
```
    function getAllFundable() internal view returns (CachedOperator[]
memory, uint256) {

...

+        assembly ("memory-safe") {

+            mstore(fundableOperators, fundableCount)

+        }

        return (fundableOperators, fundableCount);

    }
```

- OperatorsV2.getAllExitable()

Unset
```
    function getAllExitable() internal view returns
(CachedExitableOperator[] memory, uint256) {

...
```

```
+        assembly ("memory-safe") {
+            mstore(exitableOperators, exitableCount)
+        }
        return (exitableOperators, exitableCount);
    }
```

Liquid Collective's response:
Acknowledged, will fix.

When: We plan to get this in the audit changes.

This was fixed: https://github.com/liquid-collective/liquid-collective-protocol/pull/259

Gas Optimizations Recommendations

G-1. `OperatorsV2.getAllActive()`: The double loop can be reduced to one loop

Instead of a double loop and reading twice the storage variables, OperatorsV2.getAllActive() can be simplified to only loop once and edit the array's length at the end of the function:

```
Unset
    function getAllActive() internal view returns (Operator[] memory)
{
        bytes32 slot = OPERATORS_SLOT;

        SlotOperator storage r;

        // solhint-disable-next-line no-inline-assembly
        assembly {
            r.slot := slot
        }

        uint256 activeCount = 0;
        uint256 operatorCount = r.value.length;
+       Operator[] memory activeOperators = new
Operator[](operatorCount);
+
        for (uint256 idx = 0; idx < operatorCount;) {
            if (r.value[idx].active) {
+               activeOperators[activeCount] = r.value[idx];
                unchecked {
                    ++activeCount;
                }
            }
            unchecked {
                ++idx;
```

```
            }
        }

+       assembly ("memory-safe") {
+           mstore(activeOperators, activeCount)
+       }

-       Operator[] memory activeOperators = new
Operator[](activeCount);

-       uint256 activeIdx = 0;
-       for (uint256 idx = 0; idx < operatorCount;) {
-           if (r.value[idx].active) {
-               activeOperators[activeIdx] = r.value[idx];
-               unchecked {
-                   ++activeIdx;
-               }
-           }
-           unchecked {
-               ++idx;
-       }
-
        return activeOperators;
    }
-           }
```

Liquid Collective's response:

Acknowledged, will fix.

When: We plan to get this in the audit changes.

This was fixed: https://github.com/liquid-collective/liquid-collective-protocol/pull/252

G-2. River.1._onDeposit() can be simplified

- River.1.sol#L304-L310

```
Unset
    function _onDeposit(address _depositor, address _recipient,
uint256 _amount) internal override {
        uint256 mintedShares = SharesManagerV1._mintShares(_depositor,
_amount);
        IAllowlistV1 allowlist = IAllowlistV1(AllowlistAddress.get());
+        allowlist.onlyAllowed(_depositor,
LibAllowlistMasks.DEPOSIT_MASK); // this call reverts if unauthorized
or denied
-        if (_depositor == _recipient) {
-            allowlist.onlyAllowed(_depositor,
LibAllowlistMasks.DEPOSIT_MASK); // this call reverts if unauthorized
or denied
-        } else {
-            allowlist.onlyAllowed(_depositor,
LibAllowlistMasks.DEPOSIT_MASK); // this call reverts if unauthorized
or denied
+        if (_depositor != _recipient) {
            if (allowlist.isDenied(_recipient)) {
                revert Denied(_recipient);
            }
            _transfer(_depositor, _recipient, mintedShares);
        }
    }
```

Liquid Collective's response:

Acknowledged, will fix.

When: We plan to get this in the audit changes.

This was fixed: https://github.com/liquid-collective/liquid-collective-protocol/pull/252

G-3. Administrable._setAdmin(): Redundant check for _notZeroAddress

`Administrable._setAdmin()` checks for `_notZeroAddress` at Administrable.sol#L53

```
Unset
File: Administrable.sol
52:     function _setAdmin(address _admin) internal {
53:         LibSanitize._notZeroAddress(_admin); //@audit-issue gas:
redundant, already called LibAdministrable._setAdmin
https://github.com/Certora/liquid-collective-protocol/blob/0b15fbfa36
af17c1f5e8059de40c2de325210204/contracts/src/state/shared/Administrat
orAddress.sol#L23
54:         LibAdministrable._setAdmin(_admin);
55:         emit SetAdmin(_admin);
56:     }
```

However, the call to `LibAdministrable._setAdmin()` calls `AdministratorAddress.set()` (LibAdministrable.sol#L25-L27):

```
Unset
File: LibAdministrable.sol
25:     function _setAdmin(address _admin) internal {
26:         AdministratorAddress.set(_admin);
27:     }
```

And `AdministratorAddress.set()` already checks for `_notZeroAddress` (AdministratorAddress.sol#L23)

```
Unset
File: AdministratorAddress.sol
22:     function set(address _newValue) internal {
23:         LibSanitize._notZeroAddress(_newValue);
24:
LibUnstructuredStorage.setStorageAddress(ADMINISTRATOR_ADDRESS_SLOT,
_newValue);
25:     }
```

Liquid Collective's response:

Acknowledged, will fix.

When: We plan to get this in the audit changes.

This was fixed: https://github.com/liquid-collective/liquid-collective-protocol/pull/268

G-4. `OperatorsRegistry.1.addValidators()`: Storage reading inside of a loop

`operator.keys` should be cached to avoid reading multiple times from storage:

```
Unset
    function addValidators(uint256 _index, uint32 _keyCount, bytes
calldata _publicKeysAndSignatures)
        external
        onlyOperatorOrAdmin(_index)
    {
....
+       uint256 totalKeys = uint256(operator.keys);
        for (uint256 idx = 0; idx < _keyCount;) {
...
```

```
-            ValidatorKeys.set(_index, operator.keys + idx,
publicKeyAndSignature);
+            ValidatorKeys.set(_index, totalKeys + idx,
publicKeyAndSignature);
            unchecked {
                ++idx;
            }
        }
-       OperatorsV2.setKeys(_index, operator.keys + _keyCount);
+       OperatorsV2.setKeys(_index, uint32(totalKeys) + _keyCount);

        emit AddedValidatorKeys(_index, _publicKeysAndSignatures);
    }
```

Liquid Collective's response:

Acknowledged, will fix.

When: We plan to get this in the audit changes.

This was fixed: https://github.com/liquid-collective/liquid-collective-protocol/pull/252

G-5. OperatorsRegistry.1.removeValidators(): Use existing cached totalKeys instead of reading from storage

Unset

```
function removeValidators(uint256 _index, uint256[] calldata _indexes)
external onlyOperatorOrAdmin(_index) {
...
-       bool limitEqualsKeyCount = operator.keys == operator.limit;
+       bool limitEqualsKeyCount = totalKeys == operator.limit;
...
```

```
        if (limitEqualsKeyCount) {
-           operator.limit = operator.keys;
+           operator.limit = totalKeys;
        } else if (lastIndex < operator.limit) {
            operator.limit = uint32(lastIndex);
        }
    }
```

Liquid Collective's response:

We don't believe that this is correct. If we use totalKeys to set the operator.limit we would end up setting an incorrect limit. This would happen as the removal of keys would lead to the keys being present for the operator to be lower. However, since the totalKeys doesn't reflect it we would end up setting the limit to a wrong value. The correct value is reflected by operator.keys because of the update done here. So, using operator.keys is the correct way to go about this.

G-6. RedeemManager.1.sol: struct ClaimRedeemRequestParameters can be tightly packed

Unset

```
File: RedeemManager.1.sol
320:     struct ClaimRedeemRequestParameters {
+ 3xx:         /// @custom:attribute The structure of the redeem
request to claim
+ 3xx:         RedeemQueue.RedeemRequest redeemRequest;
+ 3xx:         /// @custom:attribute The structure of the withdrawal
event to use to claim the redeem request
+ 3xx:         WithdrawalStack.WithdrawalEvent withdrawalEvent;
321:         /// @custom:attribute The id of the redeem request to
claim
322:         uint32 redeemRequestId;
```

```
- 323:         /// @custom:attribute The structure of the redeem
request to claim
- 324:         RedeemQueue.RedeemRequest redeemRequest;
325:         /// @custom:attribute The id of the withdrawal event to
use to claim the redeem request
326:         uint32 withdrawalEventId;
- 327:         /// @custom:attribute The structure of the withdrawal
event to use to claim the redeem request
- 328:         WithdrawalStack.WithdrawalEvent withdrawalEvent;
329:         /// @custom:attribute The count of withdrawal events
330:         uint32 withdrawalEventCount;
331:         /// @custom:attribute The current depth of the recursive
call
332:         uint16 depth;
333:         /// @custom:attribute The amount of LsETH
redeemed/matched, needs to be reset to 0 for each call/before calling
the recursive function
334:         uint256 lsETHAmount;
335:         /// @custom:attribute The amount of eth redeemed/matched,
needs to be rest to 0 for each call/before calling the recursive
function
336:         uint256 ethAmount;
337:     }
```

Liquid Collective's response:

Acknowledged, will fix.

When: We plan to get this in the audit changes.

This was fixed: https://github.com/liquid-collective/liquid-collective-protocol/pull/252

G-7. `--rest` can be safely unchecked as it's `> 0`

- OperatorsRegistry.1.sol#L816-L818

```
Unset
File: OperatorsRegistry.1.sol
816:                     if (rest > 0) {
- 817:                         --rest;
+ 817:                         unchecked { --rest; }
818:                     }
```

Liquid Collective's response:

Acknowledged, will fix.

When: We plan to get this in the audit changes.

This was fixed: https://github.com/liquid-collective/liquid-collective-protocol/pull/252

G-8. `++siblings` can be unchecked

- OperatorsRegistry.1.sol#L786

```
Unset
File: OperatorsRegistry.1.sol
785:                 if (activeCount == highestActiveCount) {
- 786:                     ++siblings;
+ 786:                     unchecked { ++siblings; }
787:                 } else if (activeCount > highestActiveCount) {
```

Liquid Collective's response:

Acknowledged, will fix.

When: We plan to get this in the audit changes.

This was fixed: https://github.com/liquid-collective/liquid-collective-protocol/pull/252

G-9. ++operatorIndex can be unchecked due to operatorIndex != type(uint256).max

- OperatorsRegistry.1.sol#L94

```
Unset
contracts/src/OperatorsRegistry.1.sol:
   93                 } else {
   94:                    ++operatorIndex;
   95                 }
```

Liquid Collective's response:

Acknowledged, will fix.

When: We plan to get this in the audit changes.

This was fixed: https://github.com/liquid-collective/liquid-collective-protocol/pull/252

G-10. ++_params.withdrawalEventId and --_params.depth can be unchecked

```
Unset
File: RedeemManager.1.sol
415:         if (
416:             _params.redeemRequest.amount > 0 &&
_params.withdrawalEventId + 1 < _params.withdrawalEventCount
417:                 && _params.depth > 0
418:         ) {
419:             WithdrawalStack.WithdrawalEvent[] storage
withdrawalEvents = WithdrawalStack.get();
420:
421:             ++_params.withdrawalEventId; //@audit-issue Gas: can
be unchecked due to IF statement
```

```
422:            _params.withdrawalEvent =
withdrawalEvents[_params.withdrawalEventId];
423:            --_params.depth; //@audit-issue Gas: can be unchecked
due to IF statement
```

Liquid Collective's response:

Acknowledged, will fix.

When: We plan to get this in the audit changes.

This was fixed: https://github.com/liquid-collective/liquid-collective-protocol/pull/252

G-11. OperatorsRegistry.1._pickNextValidatorsToDepositFromActiveOperators(): Unnecessary new returned values

`publicKeys` and `signatures` are already at their default zero value.

- OperatorsRegistry.1.sol#L678-L686

Unset

```
File: OperatorsRegistry.1.sol
678:    function
_pickNextValidatorsToDepositFromActiveOperators(uint256 _count)
679:        internal
680:        returns (bytes[] memory publicKeys, bytes[] memory
signatures)
681:    {
682:        (OperatorsV2.CachedOperator[] memory operators, uint256
fundableOperatorCount) = OperatorsV2.getAllFundable();
683:
684:        if (fundableOperatorCount == 0) {
```

```
- 685:             return (new bytes[](0), new bytes[](0));
+ 685:             return (publicKeys, signatures);
686:         }
```

Liquid Collective's response:

Acknowledged, will fix.

When: We plan to get this in the audit changes.

This was fixed: https://github.com/liquid-collective/liquid-collective-protocol/pull/269

G-12. Unchecking arithmetics operations that can't underflow/overflow

- contracts/src/CoverageFund.1.sol

Unset
```
# File: contracts/src/CoverageFund.1.sol

CoverageFund.1.sol:47:
BalanceForCoverage.set(BalanceForCoverage.get() - amount);
```

- contracts/src/OperatorsRegistry.1.sol

Unset
```
# File: contracts/src/OperatorsRegistry.1.sol

OperatorsRegistry.1.sol:236:         uint256 operatorIndex =
OperatorsV2.push(newOperator) - 1;

OperatorsRegistry.1.sol:733:             _count -= pickedKeyCount;
```

```
OperatorsRegistry.1.sol:826:            _count -=
optimalTotalDispatchCount;

OperatorsRegistry.1.sol:848:        return initialExitRequestDemand -
_count;
```

- contracts/src/RedeemManager.1.sol

Unset

```
# File: contracts/src/RedeemManager.1.sol

RedeemManager.1.sol:120:            lastWithdrawalEvent =
withdrawalEvents[withdrawalEventsLength - 1];

RedeemManager.1.sol:170:            WithdrawalStack.WithdrawalEvent
memory previousWithdrawalEvent = withdrawalEvents[withdrawalEventId -
1];

RedeemManager.1.sol:176:        _setRedeemDemand(redeemDemand -
_lsETHWithdrawable);

RedeemManager.1.sol:299:            RedeemQueue.RedeemRequest memory
previousRedeemRequest = redeemRequests[redeemRequestId - 1];

RedeemManager.1.sol:380:                vars.exceedingEthAmount =
vars.ethAmount - maxRedeemableEthAmount;
```

- contracts/src/River.1.sol

Unset

```
# File: contracts/src/River.1.sol
```

```
River.1.sol:120:        _setCommittedBalance(CommittedBalance.get() -
dustToUncommit);

River.1.sol:487:
_setBalanceToRedeem(availableBalanceToRedeem -
suppliedRedeemManagerDemandInEth);

River.1.sol:528:                        availableBalanceToDeposit,
redeemManagerDemandInEth - _exitingBalance - availableBalanceToRedeem

River.1.sol:547:                        ? (totalRequestedExitsCount -
totalStoppedValidatorCount)

River.1.sol:553:                        redeemManagerDemandInEth -
(availableBalanceToRedeem + _exitingBalance + preExitingBalance),
```

- contracts/src/WLSETH.1.sol

```
Unset
# File: contracts/src/WLSETH.1.sol

WLSETH.1.sol:142:        BalanceOf.set(msg.sender, shares - _shares);

WLSETH.1.sol:160:            _approve(_from, msg.sender,
currentAllowance - _value);
```

- contracts/src/components/ERC20VestableVotesUpgradeable.1.sol

```
Unset
# File: contracts/src/components/ERC20VestableVotesUpgradeable.1.sol

ERC20VestableVotesUpgradeable.1.sol:414:                return
LibUint256.min(vestedAmount, globalUnlocked) - releasedAmount;
```

- contracts/src/components/SharesManager.1.sol

```
Unset
# File: contracts/src/components/SharesManager.1.sol

SharesManager.1.sol:160:            _approve(_from, msg.sender,
currentAllowance - _value);
```

Liquid Collective's response:

Acknowledged, will add this to our backlog, we will fix it in a future release.

When: Added to the backlog for a future release.

G-13. Cache array length outside of loop

If not cached, the solidity compiler will always read the length of the array during each iteration. That is, if it is a storage array, this is an extra sload operation (100 additional extra gas for each iteration except for the first) and if it is a memory array, this is an extra mload operation (3 additional gas for each iteration except for the first).

Affected code:

- contracts/src/Allowlist.1.sol

```
Unset
# File: contracts/src/Allowlist.1.sol

Allowlist.1.sol:105:        for (uint256 i = 0; i < _accounts.length;)
{

Allowlist.1.sol:141:        for (uint256 i = 0; i < _accounts.length;)
{
```

- contracts/src/Firewall.sol

```
Unset
# File: contracts/src/Firewall.sol

Firewall.sol:40:         for (uint256 i; i <
_executorCallableSelectors.length;) {
```

- contracts/src/OperatorsRegistry.1.sol

```
Unset
# File: contracts/src/OperatorsRegistry.1.sol

OperatorsRegistry.1.sol:281:         for (uint256 idx = 0; idx <
_operatorIndexes.length;) {

OperatorsRegistry.1.sol:615:         for (uint256 idx = 0; idx <
_arr1.length;) {

OperatorsRegistry.1.sol:621:         for (uint256 idx = 0; idx <
_arr2.length;) {
```

- contracts/src/RedeemManager.1.sol

```
Unset
# File: contracts/src/RedeemManager.1.sol

RedeemManager.1.sol:122:         for (uint256 idx = 0; idx <
_redeemRequestIds.length; ++idx) {
```

- contracts/src/migration/TLC_globalUnlockScheduleMigration.sol

```
Unset
# File: contracts/src/migration/TLC_globalUnlockScheduleMigration.sol
```

```
TLC_globalUnlockScheduleMigration.sol:368:        for (uint256 i = 0;
i < migrations.length; i++) {
```

- contracts/src/state/oracle/OracleMembers.sol

```
Unset
# File: contracts/src/state/oracle/OracleMembers.sol

OracleMembers.sol:80:        for (uint256 idx = 0; idx <
r.value.length;) {
```

- contracts/src/state/oracle/ReportsVariants.sol

```
Unset
# File: contracts/src/state/oracle/ReportsVariants.sol

ReportsVariants.sol:80:        for (uint256 idx = 0; idx <
r.value.length;) {
```

Liquid Collective's response:

Acknowledged, will add this to our backlog, we will fix it in a future release.

When: Added to the backlog for a future release.

G-14. State variables only set in the constructor should be declared `immutable`

Variables only set in the constructor and never edited afterwards should be marked as immutable, as it would avoid the expensive storage-writing operation in the constructor (around **20 000 gas** per variable) and replace the expensive storage-reading operations (around **2100 gas** per reading) to a less expensive value reading (**3 gas**)

Affected code:

- contracts/src/Firewall.sol

```
Unset
# File: contracts/src/Firewall.sol

Firewall.sol:35:        destination = _destination;
```

Liquid Collective's response:

Acknowledged, will fix.

When: We plan to get this in the audit changes.

This was fixed: https://github.com/liquid-collective/liquid-collective-protocol/pull/252

G-15. `uint256` to `bool` `mapping`: Utilizing Bitmaps to dramatically save on Gas

https://soliditydeveloper.com/bitmaps

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/structs/BitMaps.sol

- BitMaps.sol#L5-L16:

```
Unset
/**
 * @dev Library for managing uint256 to bool mapping in a compact and
efficient way, provided the keys are sequential.
 * Largely inspired by Uniswap's
https://github.com/Uniswap/merkle-distributor/blob/master/contracts/M
erkleDistributor.sol[merkle-distributor].
 *
```

```
 * BitMaps pack 256 booleans across each bit of a single 256-bit slot
of `uint256` type.
 * Hence booleans corresponding to 256 _sequential_ indices would only
consume a single slot,
 * unlike the regular `bool` which would consume an entire slot for a
single value.
 *
 * This results in gas savings in two ways:
 *
 * - Setting a zero value to non-zero only once every 256 times
 * - Accessing the same warm slot for every 256 _sequential_ indices
 */
```

Affected code:

- contracts/src/state/tlc/IgnoreGlobalUnlockSchedule.sol

```
Unset
# File: contracts/src/state/tlc/IgnoreGlobalUnlockSchedule.sol

IgnoreGlobalUnlockSchedule.sol:15:        mapping(uint256 => bool)
value;
```

Liquid Collective's response:

Acknowledged, will add this to our backlog, we will fix it in a future release.

When: Added to the backlog for a future release.

G-16. Increments/decrements can be unchecked in for-loops

In Solidity 0.8+, there's a default overflow check on unsigned integers. It's possible to uncheck this in for-loops and save some gas at each iteration, but at the cost of some code readability, as this uncheck cannot be made inline.

ethereum/solidity#10695

The change would be:

```
Unset
- for (uint256 i; i < numIterations; i++) {
+ for (uint256 i; i < numIterations;) {
 // ...
+   unchecked { ++i; }
}
```

These save around **25 gas saved** per instance.

The same can be applied with decrements (which should use `break` when `i == 0`).

The risk of overflow is non-existent for `uint256`.

Affected code:

- contracts/src/OperatorsRegistry.1.sol

```
Unset
# File: contracts/src/OperatorsRegistry.1.sol

OperatorsRegistry.1.sol:737:         for (uint256 idx = 0; idx <
fundableOperatorCount; ++idx) {
```

- contracts/src/RedeemManager.1.sol

```
Unset
# File: contracts/src/RedeemManager.1.sol

RedeemManager.1.sol:122:         for (uint256 idx = 0; idx <
_redeemRequestIds.length; ++idx) {
```

- contracts/src/migration/TLC_globalUnlockScheduleMigration.sol

```
Unset
# File: contracts/src/migration/TLC_globalUnlockScheduleMigration.sol

TLC_globalUnlockScheduleMigration.sol:368:        for (uint256 i = 0;
i < migrations.length; i++) {

TLC_globalUnlockScheduleMigration.sol:370:            for (uint256 j =
0; j < migration.scheduleCount; j++) {
```

Liquid Collective's response:

Acknowledged, will add this to our backlog, we will fix it in a future release.

When: Added to the backlog for a future release.

G-17. River.1.claimRedeemRequests() can be simplified

It's possible to call RedeemManager.1.claimRedeemRequests()#L152 instead of RedeemManager.1.claimRedeemRequests()#L142.

- River.1.sol#L198-L205

```
Unset
File: River.1.sol
198:     function claimRedeemRequests(uint32[] calldata
_redeemRequestIds, uint32[] calldata _withdrawalEventIds)
199:         external
200:         returns (uint8[] memory claimStatuses)
201:     {
202:         return
IRedeemManagerV1(RedeemManagerAddress.get()).claimRedeemRequests(
- 203:             _redeemRequestIds, _withdrawalEventIds, true,
type(uint16).max
+ 203:             _redeemRequestIds, _withdrawalEventIds
```

```
204:         );
205:     }
```

Liquid Collective's response:

Acknowledged, will add this to our backlog, we will fix it in a future release.

When: Added to the backlog for a future release.

G-18. LibUint256.toLittleEndian64(): unnecessary assignment

Unset

```
File: LibUint256.sol
10:     function toLittleEndian64(uint256 _value) internal pure
returns (uint256 result) {
- 11:         result = 0; //@audit-issue gas: this is already
initialized to 0
```

Liquid Collective's response:

Acknowledged, will fix.

When: We plan to get this in the audit changes.

This was fixed: https://github.com/liquid-collective/liquid-collective-protocol/pull/252

Formal Verification

Assumptions and Simplifications Made During Verification

General Assumptions

A. Loops (‘for’ and ‘while’) are unrolled to a few iterations - see contracts assumptions for specific loop unrolling amount.

Configuration and Trusted entities are assumed correct

Due to the centralized nature of the protocol, and the technical limitation of LiquidStaking protocols over ethereum in general, some entities had to be trusted and filtered out from certain rule or invariant checks, e.g. the oracle report on River has to be fully trusted with the reported amount of ETH on the beacon chain.
In addition, it was assumed that the contracts were correctly configured to point at each other (which is an operational consideration). E.g. the address of River was correctly configured on RedeemManager, etc.

Deposit Contract

It was assumed that the deposit contract interface and behavior is correct. The validator keys and WC supplied to the protocol are assumed to be working as intended (off-chain responsibility of the operators and the admin’s vetting).

Formal Verification Properties

Notations

✅Indicates the rule is formally verified.
❌Indicates the rule is violated.

Since the protocol consists of different contracts, we will present the relative properties for each of the main contracts in separate sections.

The following contracts were formally verified, and the properties are listed below per library/contract:

A. River.sol
B. RedeemManager.sol
C. OperatorsRegistry.sol
D. Oracle.sol

RiverV1.sol

Definitions

- The assets value (in ETH) of any *account* is the sum of its native balance and the value of its shares, where the latter is the value calculated by the function *underlyingBalanceFromShares(balanceOf(account))*.

Assumptions

- We verified the contract functions against an arbitrary storage state.
- Some of the rules have an exception to the assumption above, where we assume the following precondition: if the total supply is non-zero, then the total amount of ETH or shares cannot exceed 2^128 and must be at least 10^16 (= 0.01ETH).
- The contract was assumed to be in the post-initialization state, i..e after the function initRiverV1_1() was executed for the current implementation.

Properties

1. ✅ Correct accounting of underlying eth
    a. Total supply of shares must match the shares corresponding to the underlying tokens.
2. ✅ Share amounts are monotonic with ETH amount in the protocol
3. ❌ The shares (LsETH) total supply == 0 iff the protocol underlying ETH == 0 w

4. ✅ The River ETH balance covers the validators balance + balance to deposit + committed balance + balance to redeem.
5. ❌ The conversion rate of LsETH to ETH remains stable for every normal user's action (up to one uint256 error).
    - See issue L-5 for more information, note the following property River-6 was proven.
6. ✅ The conversion rate of LsETH to ETH remains stable for every normal user's action, up to rounding errors.
7. ✅ The allowance of spender given by owner can always be decreased to 0 by the owner.
8. ✅ It is impossible to increase any allowance by calling decreaseAllowance or transferFrom.
9. ✅ Allowance increases only by the owner.
10. ✅ Correctness of transferFrom() / transfer():
    a. Balances are updated accordingly
    b. Balances other than the recipient's and sender's are untouched.
    c. totalSupply() is conserved.
11. ✅ It is never beneficial for any user to deposit in multiple smaller patches instead of one big patch.[1]
12. ✅ For every action that transfers shares, the sum of balances and total supply must not change.
13. ✅ The share balance of any user is never larger than the total supply of shares.[2]
14. ✅ Shares transfer doesn't increase assets value.[3]
15. ✅ A user cannot increase the value of his own assets (except by claimRedeemRequests()). [4] [5]
16. ❌ The assets of a black-listed user are immutable.
17. ✅ When a user deposits, there is no additional gift component to the deposit - the recipient assets value is increases by at most msg.value (+1 error).
18. ✅ The deposit of any amount of ETH to the protocol, satisfies the following shares-additivity rule:
    The number of minted shares by calling two subsequent deposits (deposit(x), deposit(y)) yields the same amount of minted shares for a combined deposit(x+y) with the following error:
    -2 <= Shares({x+y}) - shares({x, y}) <= 2 + (y + 1) / (X + x),
    where X > 0 is the initial underlying balance of the protocol.
    For X == 0, the upper bound for the discrepancy is 2.
19. ✅ The validators count from the last consensus report can never exceed the deposited validators count.
20. ❌ For any permissible CL spec configuration, there is only one valid epoch per timestamp.

[1] Any split can at most yield one unit (uint256(1)) with respect to one big deposit.
[2] By assuming that the sum of shares of any two members is less than the total supply we are able to prove this claim.
[3] The value can increase the most by one unit - uint256(1).
[4] The value can increase the most by one unit - uint256(1).
[5] A "user" address is excluded from being any of the system contracts addresses.

RedeemManagerV1.sol

Assumptions

- We verified the contract functions against an arbitrary storage state.
- The total number of redeem requests / withdrawal events was assumed to be less than max(uint32).
- claimRedeemRequests() recursion depth is at most 2 (max(dept) = 2).
- Loops are assumed to iterate at most 3 times.
- No custom fallback/receive functions were assumed to be called. All calls of the form `receiver.call{value: value}("");` were assumed to accept ETH optimistically.

Properties

1. ✅ The height of any redeem request is at least the sum of any of its previous request amount and height (HeightOfSubequentRequest).
2. ✅ The owner of any redeem request is not the zero address (NonZeroRedeemRequestOwner).
3. ✅ The height of the first withdrawal event is zero (HeightOfFirstEventIsZero).
4. ✅ The Height difference of two adjacent withdrawal events is the amount of the former (WithdrawalEventsHeightDifferenceIsAmount).
5. ✅ If any two requests match the same withdrawal event, then their height difference must be smaller than the event size (twoRequestsMatchTheSameEvent).
6. ✅ If any two events match the same redeem request, then their height difference must be smaller than the request size (twoEventsMatchTheSameRequest).
7. ❌ Any registered withdrawal event has non-zero amount (WithdrawalEventHasNonZeroAmount).
8. ✅ The height of the first redeem request, right after creation, is zero (firstRedeemRequestHeightIsZero).
9. ✅ For every redeem request, the sum of its height and shares amount is preserved once created (sumOfRequestHeightAndAmountIsPreserved).
10. ✅ The redeem requests array cannot drop requests - length doesn't decrease (redeemRequestsNeverDrop).
11. ✅ The redeem amount of any request cannot increase (redeemRequestAmountCannotIncrease).
12. ✅ The height of any withdrawal event is at least the sum of any of its previous event amount and height (HeightOfSubequentEvent).
13. ✅ No previous claim request can change the amount redeemed by another request (subequentRequestCannotStealClaim)
14. ✅ The claimRedeemRequest() function is depth-associative (for each request separately) (claimRequestDepthAssociative).
15. ✅ The claimRedeemRequest() function is amount-additive (for each request separately) (claimRequestAdditive).
16. ✅ The claimRedeemRequests() function is request-commutative (claimRequestCommutative1).
17. ✅ The claimRedeemRequests() function is event-commutative (claimRequestCommutative2).
18. ✅ Given 2 consequent redeem requests and a single withdrawal event, if the second request is fully claimed then the first request cannot be partially claimed (singleEventMustSatisfyEarlierRequestFirst).
19. ✅ The output length of claimStatuses equals the length of redeem requests (claimStatusesSizeEqualsRequestsSize).

20. ✅ The terminal (immutable) state of any redeem request is when its amount is zero (fullClaimIsTerminal).
21. ✅ Request Ids are incremental, hence unique (incrementalRedeemRequestId).
22. ✅ If the total height of an event is larger than the total height of a request, then this event must fully satisfy this redeem request (withdrawalHeightSatisfiesRequestLowerHeight).
23. ✅ Cannot drain a withdrawal event more than its size (maximumDrainOfWithdrawalEvent).
24. ✅ Only a larger event amount can fully satisfy a request than is otherwise partially satisfied (cannotFullyClaimForASmallerAmount).
25. ✅ The claimed amount is monotonically increasing with the event size (claimedAmountIsMonotonicWithEventSize).
26. ✅ The resolveRedeemRequests() is requests-associative (resolutionRequestsAssociative).
27. ✅ The resolution of resolveRedeemRequests() is always a match for the request (resolutionRequestsYieldsAMatch).
28. ✅ Any pushed event comes together with the event ETH deposit to the contract (eventIsPushedTogetherWithETH).
29. ✅ It's impossible for any matching request to drain an event more than its withdrawn ETH (cannotDrainMoreThanEventETH).
30. ✅ The change in the contract ETH balance is equal to the change in the request max redeemable ETH (changeOfContractEthBalanceIsChangeOfMatchingPair).
31. ✅ The maximum amount of redeemed ETH is equal to maxRedeemableEth (maxRedeemedAmount).
32. ❌ The request "share price" (= maxRedeemableEth/amount in request) cannot increase (requestSharePriceCannotIncrease).
33. ✅ The share price of redeemed ETH is the minimum between the withdrawal event and request share prices (minimalSharePrice).
34. ✅ The redeem manager cannot send shares back (redeemManagerDoesntSendShares).

OperatorsRegistryV1.sol

Assumptions

- We verified the contract functions against an arbitrary storage state.
- Most of the invariant/rules for this contract were checked against two operators in the state, as in most cases even a single operator's valid state shouldn't be affected by others
- In a similar manner, it was assumed for most rules that each operator has a maximum of 3 validator keys
- For some rules, like OperatorsRegistry-3, some specific method's arguments were limited to restrict number of iterations (e.g., removeValidators with one index)
- NOTICE - the *removeValidators()* function couldn't always be proven for all rules, and whenever there was a technical gap it's mentioned in the property below, and was covered via manual analysis. All those instances were found to hold the property by said manual checks.

Properties

1. ✅ Operators' validator counts are valid, i.e available keys >= limit >= funded >= requestedExist keys. (operatorsStatesRemainValid)
2. ✅ If a key is over the limit, there's no action that makes it go under the limit but the limit didn't increase. (validatorStateTransition_2_1_index_limit)
3. ✅ Inactive operators can't be funded (inactiveOperatorsRemainNotFunded)
4. ❌Operators addresses are unique (operatorsAddressesRemainUnique)
    a. Violated by *setOperatorAddress* and admin methods, customer clarified it's an expected violation that doesn't break any offchain functionality
5. ✅ Only specified methods can de/activate node operator (whoCanDeactivateOperator)
    a. Only *initOperatorsRegistryV1_1* and *setOperatorStatus* can de/activate node operator
6. ✅ Only specified methods can add/remove operators (whoCanChangeOperatorsCount)
    a. Only *addOperator* and *initOperatorsRegistryV1_1* can increase the number of operators
    b. Only *initOperatorsRegistryV1_1* can decrease the number of operators
7. ✅ RequestValidatorExits selects operators such that difference in allocations decreases (exitingValidatorsDecreasesDiscrepancy)
8. ✅ When a new node operator is added, it starts with having 0 keys (newNOHasZeroKeys)
9. ✅Operator.funded and Operator.exitingValidatorsDecreasesDiscrepancy
10. can never decrease (fundedAndExitedCanOnlyIncrease)
    a. Proven for all methods except *removeValidators()*
11. ✅ Revert conditions for *removeValidators(uint256 _index, uint256[] _indexes)*
    a. Method reverts if *_indexes* contains the same element twice (removeValidatorsRevertsIfKeysDuplicit)
    b. Method reverts if *_indexes* are not sorted from largest to smallest (removeValidatorsRevertsIfKeysNotSorted)
12. ✅State-transitions of validators
    a. Proven for all methods except *removeValidators()*
    b. A validator can only become *exited* if it is currently *funded (< Operator.funded)* (validatorStateTransition_4in)
    c. A validator can only become *funded* if it is currently *fundable (< Operator.limit)* (validatorStateTransition_3in)
    d. A validator can only become *fundable* if it is currently *available (< Operator.keys)* (validatorStateTransition_2in)
    e. A validator can only become *available* if it is currently *fundable* or *not present* (validatorStateTransition_1in)
    f. A validator can only become *not present* if it is currently *fundable* or *available* (validatorStateTransition_0in)
13. ✅The difference of running validator counts between two operators can only decrease due to round-robin selection on pickNextValidatorToDeposit() or requestValidatorExits() - up to the allowed difference threshold.
    a. Rule - (exitingValidatorsDecreasesDiscrepancy)
    b. Rule - (startingValidatorsDecreasesDiscrepancy)

OracleV1.sol

Assumptions

- We verified the contract functions against an arbitrary storage state.
- We assume that the contract initialization is controlled by a trusted party.
- We assume (based on the parameters of the CL Spec in River), that for every frame there exists only one valid epoch.

Properties

1. ✅ Correctness of the acceptAdmin() function (acceptAdminIntegrity):
    a. The new admin must be the proposed one.
    b. Only the proposed admin can accept.
    c. The pending admin after the acceptance is zero.
2. ✅ Correctness of the proposeAdmin() function (proposeAdminIntegrity):
    a. The admin address cannot change while proposing a new one.
    b. Only the admin can propose a new admin.
    c. The pending admin must be set correctly.
3. ✅ Only the pending admin or the admin can change the pending admin (onlyPendingAdminOrAdminCanChangePendingAdmin).
4. ✅ Only the admin can change the quorum (onlyAdminChangesQuorum).
5. ❌ The quorum value Q should respect the following invariant, where O is oracle member count (QuorumBounds).
6. ✅ The zero address is never an oracle member (ZeroAddressIsNotAMember).
7. ✅ No addresses duplicates in oracle members index (NoMembersDuplicates).
8. ✅ The setMember() function (setMemberIntegrity):
    a. Cannot assign the same member again.
    b. Doesn't set the zero address as the zero address.
    c. Revokes the old member, only if it was a member before.
    d. Adds the new member, only if it was not a member before.
9. ✅An account cannot be added to the members list if it's already a member.
10. ✅An account cannot be removed from the members list if it's not a member
11. ✅Only the setMember(), removeMember() and addMember() can change the isMember() status
12. ✅only the admin can add/remove members. Only a member can revoke himself.
13. ✅ An actor who is not an admin cannot front-run and make setQuorum() revert.
14. ✅ Any submitted report ID must be after the last reported epoch.
15. ❌ No one can front-run a submission of a report with the same report submission and make the second one revert.
16. ✅ An oracle cannot submit reports in the same epoch (timestamp).
17. ✅ A non oracle member cannot change the number of votes.
18. ✅ Votes can either increase by 1 or be nullified.
19. ✅ Correctness of reportConsensusLayerData():
    a. If the member has voted before, it can only vote again for a later epoch than the one which was last submitted.

   b. After the function call, the member status must be true, or result in the report being submitted to the river (and as a result nullifying all votes).
20. ✅ If a report has been submitted, then all previous votes must be nullified and so are the report variants.

Disclaimer

The Certora Prover takes a contract and a specification as input and formally proves that the contract satisfies the specification in all scenarios. Notably, the guarantees of the Certora Prover are scoped to the provided specification and the Certora Prover does not check any cases not covered by the specification.

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects, and incident response.