Liquid Collective Security Review

Auditors

Saw-mon and Natalie, Lead Security Researcher

Optimum, Lead Security Researcher

Emanuele Ricci, Security Researcher

Danyal Ellahi, Junior Security Researcher

Matt Eccentricexit, Junior Security Researcher

Report prepared by: Pablo Misirov, Danyal Ellahi and Matt Eccentricexit

December 19, 2022

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Liquid Collective is an enterprise-grade liquid staking protocol, built on Ethereum. It allows institutional investors to stake and earn staking rewards while keeping access to staked collateral in the form of a liquid token. Liquid Collective offers a solution that caters to the needs of institutions including:

- KYC / AML whitelisting process for all participants (including validators)
- Top performing validators
- Governance by a consortium of partners

Disclaimer : This security review does not guarantee against a hack. It is a snapshot in time of the Liquid Collective protocol according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 10 days in total, Liquid Collective engaged with Spearbit to review the Liquid Collective protocol. In this period of time a total of 85 issues were found.

Note: The Alluvial team has changed the name of River Protocol to Liquid Collective Protocol.

Summary

Project Name	Liquid Collective
Repository	liquid-collective-protocol
Commit	778d71c5c2b0bb7d4...ee6a
Type of Project	Liquid Staking, DeFi
Audit Timeline	Aug 29th - Sept 9th
Two week fix period	Sep 9th - Sep 23rd
Fix period extension	Sep 26th - Sep 30th
Methods	Manual Review

Issues Found

Severity	**Count**	**Fixed**	**Acknowledged**
Critical Risk	3	3	0
High Risk	4	3	1
Medium Risk	15	12	3
Low Risk	5	4	1
Gas Optimizations	19	18	1
Informational	39	32	7
Total	**85**	**72**	**13**

5 Remediation Table

The following table contains all issues found during the audit together with its corresponding severity and fix PR

Number	Issue	Severity	PR
6.1.1	An attacker can freeze all incoming deposits and brick the oracle members' reporting system with only `1 wei`	Critical	SPEARBIT/4
6.1.2	`Operators._hasFundableKeys` returns true for operators that do not have fundable keys	Critical	SPEARBIT/3
6.1.3	`OperatorsRegistry._getNextValidatorsFromActiveOperators` can DOS Alluvial staking if there's an operator with `funded==stopped` and `funded == min(limit, keys)`	Critical	SPEARBIT/3
6.2.1	`Oracle.removeMember` could, in the same epoch, allow members to vote multiple times and other members to not vote at all	High	SPEARBIT/2
6.2.2	Order of calls to `removeValidators` can affect the resulting validator keys set	High	Acknowledged
6.2.3	`_hasFundableKeys` marks operators that have no more fundable validators as fundable	High	SPEARBIT/3
6.2.4	Non-zero `operator.limit` should always be greater than or equal to `operator.funded`	High	SPEARBIT/3
6.3.1	Decrementing the quorum in Oracle in some scenarios can open up a frontrunning/backrunning opportunity for some oracle members	Medium	PR 151
6.3.2	`_getNextValidatorsFromActiveOperators` can be tweaked to find an operator with a better validator pool	Medium	SPEARBIT/3
6.3.3	Dust might be trapped in `WlsETH` when burning one's balance	Medium	SPEARBIT/5
6.3.4	`BytesLib.concat` can potentailly return results with dirty byte paddings	Medium	SPEARBIT/6
6.3.5	The `reportBeacon` is prone to front-running attacks by oracle members	Medium	Acknowledged
6.3.6	Avoid multiple divisions when calculating `operatorRewards`	Medium	SPEARBIT/8
6.3.7	Shares distributed to operators suffer from rounding error	Medium	SPEARBIT/8
6.3.8	`OperatorsRegistry._getNextValidatorsFromActiveOperators` should not consider stopped when picking a validator	Medium	SPEARBIT/3
6.3.9	`approve()` function can be front-ran resulting in token theft	Medium	SPEARBIT/9
6.3.10	Add missing input validation on constructor/initializer/setters	Medium	SPEARBIT/10
6.3.11	LibOwnable.setAdmin allows setting address(0) as the admin of the contract	Medium	SPEARBIT/11 and SPEARBIT/33
6.3.12	`OracleV1.getMemberReportStatus` returns true for non existing oracles	Medium	SPEARBIT/12

6.3.13	Operators might add the same validator more than once	Medium	Acknowledged
6.3.14	`OracleManager.setBeaconData` possible front running attacks	Medium	Acknowledged
6.3.15	`SharesManager._mintShares - Depositors` may receive zero shares due to front-running	Medium	SPEARBIT/4
6.4.1	Orphaned (index, values) in `SlotOperator` storage slots in `operatorsRegistry`	Low	SPEARBIT/14
6.4.2	A malicious operator can purposefully mismanage its validators to benefit from `lsETH` market price movements	Low	Acknowledged
6.4.3	Warn the `admin` and the `operator` that `setOperatorName` has other side-effects	Low	SPEARBIT/14
6.4.4	`OperatorsRegistry.setOperatorName` Possible front running attacks	Low	SPEARBIT/14
6.4.5	Prevent users from burning token via `lsETH` / `wlsETH` `transfer` or `transferFrom` functions	Low	SPEARBIT/9
6.5.1	In `addOperator` when emitting an event use stack variables instead of reading from memory again	Gas Op	PR 151
6.5.2	Avoid slicing and concatenating by passing a different pattern of bytes to `addValidators`	Gas Op	SPEARBIT/24
6.5.3	Cache `_indexes[indexes.length - 1]` in `removeValidators` to avoid reading from storage multiple times	Gas Op	SPEARBIT/24
6.5.4	Avoid updating `operator.keys` storage slot inside of a loop in `removeValidators`	Gas Op	SPEARBIT/24
6.5.5	Cache storage related variables in `WLSETH.1.sol::burn` to save gas	Gas Op	SPEARBIT/5 and SPEARBIT/33
6.5.6	Replace `pad64` with `abi.encodePacked`	Gas Op	SPEARBIT/6
6.5.7	Unroll loops with a fixed number of iterations to avoid extra gas costs.	Gas Op	SPEARBIT/25
6.5.8	Rewrite `pad64` so that it doesn't use `BytesLib.concat` and `BytesLib.slice` to save gas	Gas Op	SPEARBIT/6
6.5.9	Cache `r.value.length` used in a loop condition to avoid reading from the storage multiple times	Gas Op	SPEARBIT/14
6.5.10	Cache the `r.value.length - 1` value to avoid reading from the storage multiple times	Gas Op	SPEARBIT/14
6.5.11	Caching `activeCount` in `Operators.sol::`getAllActive/getAllFundable to storage to save gas	Gas Op	Acknowledged
6.5.12	Rewrite the for loop in `ValidatorKeys.sol`::getKeys to save gas	Gas Op	SPEARBIT/24
6.5.13	`Operators.get` in `_getNextValidatorsFromActiveOperators` can be replaced by `Operators.getByIndex` to avoid extra operations/gas	Gas Op	SPEARBIT/14
6.5.14	Avoid unnecessary equality checks with `true` in `if` statements	Gas Op	SPEARBIT/14

6.5.15	Rewrite `OperatorRegistry.getOperatorDetails` to save gas	Gas Op	SPEARBIT/14
6.5.16	Rewrite/simplify `OracleV1.isMember` to save gas	Gas Op	SPEARBIT/20
6.5.17	Cache `beaconSpec.secondsPerSlot * beaconSpec.slotsPerEpoch` multiplication in to save gas	Gas Op	SPEARBIT/20
6.5.18	Avoid to waste gas distributing rewards when the number of shares to be distributed is zero	Gas Op	SPEARBIT/8
6.5.19	`_rewardOperators` could save gas by skipping operators with no active and funded validators	Gas Op	SPEARBIT/8
6.6.1	Consider adding a strict check to prevent `Oracle` admin to add more than 256 members	Informational	Acknowledged
6.6.2	`ApprovalsPerOwner.set` does not check if `owner` or `spender` is `address(0)`	Informational	PR 151
6.6.3	Quorum could be higher than the number of oracles, DOSing the Oracle contract	Informational	SPEARBIT/15
6.6.4	`ConsensusLayerDepositManager.depositToConsensusLayer` should be called only after a quorum has been reached to avoid rewarding validators that have not performed during the frame	Informational	SPEARBIT/8
6.6.5	Document the decision to include `executionLayerFees` in the logic to trigger `_onEarnings` to distribute rewards to Operators and Treasury	Informational	SPEARBIT/8
6.6.6	Consider documenting how and if funds from the execution layer fee recipient are considered inside the `annualAprUpperBound` and `relativeLowerBound` boundaries	Informational	Acknowledged
6.6.7	`Allowlist.allow` allows arbitrary values for `_statuses input`	Informational	Acknowledged
6.6.8	Consider exploring a way to update the `withdrawal credentials` and document all the possible scenarios	Informational	Acknowledged
6.6.9	`Oracle` contract allows members to skip frames and report them (even if they are past) one by one or all at once	Informational	Acknowledged
6.6.10	Consider renaming `OperatorResolution.active` to a more meaningful name	Informational	SPEARBIT/14
6.6.11	`lsETH` and `WlsETH's name()` functions return inconsistent name	Informational	SPEARBIT/18
6.6.12	Rename modifiers to have consistent naming and patterns `only<ROLE>`	Informational	SPEARBIT/30
6.6.13	`OperatorResolution.active` might be a redundant struct field which can be removed	Informational	SPEARBIT/14
6.6.14	Inline the known value of the boolean `opExists` with its value	Informational	SPEARBIT/14
6.6.15	The expression for `selectedOperatorAvailableKeys` in `OperatorsRegistry` can be simplified	Informational	SPEARBIT/3
6.6.16	The unused constant `DELTA BASE` can be removed	Informational	SPEARBIT/30
6.6.17	Remove unused modifiers	Informational	SPEARBIT/30
6.6.18	Modifier names do not follow the same naming patterns	Informational	SPEARBIT/30

6.6.19	In `AllowlistV1.allow` the input variable `_statuses` can be renamed to better represent that values it holds	Informational	SPEARBIT/30
6.6.20	`riverAddress` can be renamed to `river` and we can avoid extra interface casting	Informational	SPEARBIT/30
6.6.21	Define named constants for numeric literals	Informational	SPEARBIT/19
6.6.22	Move `memberIndex` and `ReportsPositions` checks at the beginning of the `OracleV1.reportBeacon` function	Informational	SPEARBIT/20
6.6.23	Document what incentivizes the operators to run their validators when `globalFee` is zero	Informational	Acknowledged
6.6.24	Document how Alluvial plans to prevent institutional investors and operators get into business directly and bypass using the `River` protocol	Informational	Acknowledged
6.6.25	Document how operator rewards will be distributed if `OperatorRewardsShare` is zero	Informational	SPEARBIT/8
6.6.26	Current operator reward distribution does not favor more performant operators	Informational	SPEARBIT/8
6.6.27	`TRANSFER_MASK == 0` which causes a no-op	Informational	SPEARBIT/21
6.6.28	Reformat numeric literals with many digits for better readability	Informational	SPEARBIT/19
6.6.29	`Firewall` should follow the two-step approach present in `River` when transferring `govern` address	Informational	SPEARBIT/33
6.6.30	`OperatorRegistry.removeValidators` is resetting the `limit` (approved validators) even when not needed	Informational	SPEARBIT/22
6.6.31	Consider renaming `transferOwnership` to better reflect the function's logic	Informational	SPEARBIT/11
6.6.32	Wrong return name used	Informational	SPEARBIT/30
6.6.33	Discrepancy between architecture and code	Informational	SPEARBIT/27
6.6.34	Consider replacing the remaining `require` with custom errors	Informational	SPEARBIT/23
6.6.35	Both `wlsETH` and `lsETH transferFrom` implementation allow the owner of the token to use `transferFrom` like if it was a "normal" transfer	Informational	SPEARBIT/9
6.6.36	Both `wlsETH` and `lsETH` tokens are reducing the allowance when the allowed amount is `type(uint256).max`	Informational	SPEARBIT/9
6.6.37	Missing, confusing or wrong natspec comments	Informational	SPEARBIT/33
6.6.38	Remove unused imports from code	Informational	SPEARBIT/32
6.6.39	Missing event emission in critical functions, init functions and setters	Informational	SPEARBIT/31

6 Findings

6.1 Critical Risk

6.1.1 An attacker can freeze all incoming deposits and brick the oracle members' reporting system with only `1 wei`

Severity: *Critical Risk*

Context: SharesManager.1.sol#L195-L206

Description: An attacker can brick/lock all deposited user funds and also prevent oracle members to come to a quorum when there is an earning to be distributed as rewards. Consider the following scenario:

1. The attacker force sends `1 wei` to the `River` contract using, e.g., `selfdestruct`. The attacker has to make sure to perform this transaction before any other users deposit their funds in the contract. The attacker can look at the mempool and also front-run the initial user deposit. Now the `b = _assetBalance() > 0`, is at least `1 wei`.
2. Now an allowed user tries to deposit funds into the `River` protocol. The call eventually ends up in `_mintShares(o, x)` and in the 1st line `oldTotalAssetBalance = _assetBalance() - x`, `_assetBalance()` represents the updated `River` balance after taking into account the `x` deposit as well by the user. So `_assetBalance()` is now `b + x + ...` and `oldTotalAssetBalance = b + ...` where the `...` includes beacon balance sum, deposited amounts for validators in queue, ... (which is probably `0` by now). Therefore, `oldTotalAssetBalance > 0` means that the following `if` block is skipped:

```
if (oldTotalAssetBalance == 0) {
    _mintRawShares(_owner, _underlyingAssetValue);
    return _underlyingAssetValue;
}
```

And goes directly to the `else` block for the 1st allowed user deposit:

```
else {
    uint256 sharesToMint = (_underlyingAssetValue * _totalSupply()) / oldTotalAssetBalance;
    _mintRawShares(_owner, sharesToMint);
    return sharesToMint;
}
```

But since shares have not been minted yet `_totalSupply() == 0`, and therefore `sharesToMint == 0`. So we mint `0` shares for the user and return `0`. Note that `_assetBalance()` keeps increasing, but `_totalSupply()` or `Shares.get()` remains `0`.

3. Now the next allowed users deposit funds and just like step 2. `River` would mint them `0` shares, `_assetBalance()` increases but `_totalSupply()` or `Shares.get()` remains `0`.

Note that `_totalSupply()` or `Shares.get()` remains `0` until the oracle members come to a quorum for the beacon balance sum and number of validators for a voting frame. Then the last oracle member who calls the `reportBeacon` to trigger the quorum causes a call to `_pushToRiver` which in turn calls `river.setBeaconData`. Now in `setBeaconData` if we have accumulated interest then `_onEarnings` is called. The 1st few lines of `_onEarnings` are:

```
uint256 currentTotalSupply = _totalSupply();
if (currentTotalSupply == 0) {
    revert ZeroMintedShares();
}
```

But `_totalSupply()` is still `0` so `revert ZeroMintedShares()` is called and the revert bubbles up the stack of call to `reportBeacon`, which means that the last oracle member that can trigger a quorum will have its call to `reportBeacon` reverted. Therefore, no quorums will ever be made which has some earnings and the `River` protocol will stay unaware of its deposited validators on the beacon change. Any possible path to `_mintRawShares` (which could cause `Shares.get()` to increase) is also blocked and `_totalSupply()` would stay at `0`.

So even after validators, oracle members, etc.. become active, when an allowed user deposits they would receive `0` shares.

Note, that an attacker can turn this into a DoS attack for the `River` Protocol, since redeployment alone would not solve this issue. The attacker can monitor the mempool and always try to be 1st person to force deposit `1 wei` into the `River` deployed contract.

Alluvial: If we change the condition that checks if the old underlying asset balance is zero to checking if the total shares is under a minimum amount (so we would mint 1:1 as long as we haven't reached that value) would it solve the issue? This minimum value can be `DEPOSIT_SIZE` as the price per share should be 1:1 anyway as no revenue was generated.

Spearbit: Yes, this would solve this issue. Alluvial can also as part of an atomic deployment send `1 wei` and mint the corresponding share using a call to the internal function `_mintRawShares`.

Also note, we should remove the check for `oldTotalAssetBalance == 0` as `oldTotalAssetBalance` is used as the denominator for `sharesToMint`. There could be some edge scenarios where the `_totalSupply()` is positive but `oldTotalAssetBalance` is `0`. So if extra checks are introduced for `_totalSupply()`, we should still keep the check or a modified version for `oldTotalAssetBalance`.

Spearbit: In PR:

- [SPEARBIT/4] Add a ethToDeposit storage var that accounts for incoming ETH

Alluvial introduces `BalanceToDeposit` storage variable. This variable is basically replacing `address(this).balance` for `River` in multiple places including `_assetBalance()` function. The `BalanceToDeposit` can only be modified when:

1. An allowed user deposits into `River` which will increase `BalanceToDeposit` and also total minted shares
2. An entity (in later commits only the `admin` can call this endpoint) calls `depositToConsensusLayer` which might reduce the `BalanceToDeposit` amount but the net effect on `_assetBalance()` would be zero. That would also mean `BalanceToDeposit` should have been non-zero to begin with.
3. A call to `setConsensusLayerData` by the Oracle (originated by a call to `reportConsensusLayerData` by an oracle member) which will pull fees from `ELFeeRecipientAddress` (in later commits it will only pull fees if needed and up to a max amount) and would increase `BalanceToDeposit`.

Note the attack in this issue works by making sure `_assetBalance()` is non-zero while `_totalSupply()` is zero. That means we cannot afford a call to end up at `_onEarnings` for this scheme to work. Since if `_totalSupply() == 0`, `_onEarnings` would revert. That means even with a malicious group of oracle members reporting wrong data, if a call ends up at `setConsensusLayerData` with `_validatorCount = 0` and `_validatorTotalBalance > 0`, `_onEarnings` would trip. Also if the oracle members are not malicious and just report `_validatorCount = 0` and `_validatorTotalBalance = 0`, but an attacker force sends `1 wei` to `ELFeeRecipientAddress`, the same thing happens again and `_onEarnings` would revert since no shares are minted yet, That means all the possible ways to have a net positive effect on `BalanceToDeposit` (and thus `_assetBalance()`) while keeping `_totalSupply()` zero is blocked. But

```
_assetBalance() = (
   BalanceToDeposit.get() +
   CLValidatorTotalBalance.get() +
   (DepositedValidatorCount.get() - CLValidatorCount.get()) *
↪  ConsensusLayerDepositManagerV1.DEPOSIT_SIZE
);
```

or

$$B = D + B_v + 32(C_d - C_v)$$

where:

- B is `_assetBalance()`

- D is `BalanceToDeposit.get()`
- B_v is `CLValidatorTotalBalance.get()`
- C_d is `DepositedValidatorCount.get()`
- C_v is `CLValidatorCount.get()`

Also $C_v \leq C_d$ is an invariant. B_v, C_v are only set in `setConsensusLayerData` and thus can only be changed by a quorum of oracle members. C_d only increases and is only set in `depositToConsensusLayer` and requires a positive D. After a call to `depositToConsensusLayer`, $\Delta D = -32\Delta C_d$ ($\Delta B = 0$).

Thus putting all this info together, all the possible points of attack to make sure B will be positive while keeping `_totalSupply()` zero are blocked.

A note for the future: when users are able to withdraw their investment and burn their shares if all users withdraw and due to some rounding errors or other causes B stays positive, then the next user to deposit and mint a share would receive zero shares.

6.1.2 `Operators._hasFundableKeys` returns `true` for operators that do not have fundable keys

Severity: *Critical Risk*

Context: Operators.sol#L149-L154

Description: Because `_hasFundableKeys` uses `operator.stopped` in the check, an operator without fundable keys be validated and return `true`.

Scenario: Op1 has

- `keys = 10`
- `limit = 10`
- `funded = 10`
- `stopped = 10`

This means that all the keys got funded, but also "exited". Because of how `_hasFundableKeys` is made, when you call `_hasFundableKeys(op1)` it will return `true` even if the operator does not have keys available to be funded.

By returning `true`, the operator gets wrongly included in `getAllFundable` returned array. That function is critical because it is the one used by `pickNextValidators` that picks the next validator to be selected and stake delegate user ETH.

Because of this issue in `_hasFundableKeys` also the issue *`OperatorsRegistry._getNextValidatorsFromActive-Operators` can DOS Alluvial staking if there's an operator with `funded==stopped` and `funded == min(limit, keys)`* can happen DOSing the contract that will always make `pickNextValidators` return empty.

Check *Appendix* for a test case to reproduce this issue.

Recommendation: Alluvial should reimplement the logic of `Operators. _hasFundableKeys` that should return `true` if and only if the operator is active and has fundable keys. The attribute `stopped` should not be used.

Alluvial: Recommendations implemented in PR SPEARBIT/3.

Spearbit: Acknowledged.

6.1.3 `OperatorsRegistry._getNextValidatorsFromActiveOperators` can DOS Alluvial staking if there's an operator with `funded==stopped` and `funded == min(limit, keys)`

Severity: *Critical Risk*

Context: OperatorsRegistry.1.sol#L403-L454

Description: This issue is also related to *`OperatorsRegistry._getNextValidatorsFromActiveOperators` should not consider `stopped` when picking a validator*.

Consider a scenario where we have

```
Op at index 0
name op1
active true
limit 10
funded 10
stopped 10
keys 10

Op at index 1
name op2
active true
limit 10
funded 0
stopped 0
keys 10
```

In this case,

- Op1 got all 10 keys funded and exited. Because it has `keys=10` and `limit=10` it means that it has no more keys to get funded again.
- Op2 instead has still 10 approved keys to be funded.

Because of how the selection of the picked validator works

```
uint256 selectedOperatorIndex = 0;
for (uint256 idx = 1; idx < operators.length;) {
    if (
        operators[idx].funded - operators[idx].stopped
            < operators[selectedOperatorIndex].funded - operators[selectedOperatorIndex].stopped
    ) {
        selectedOperatorIndex = idx;
    }
    unchecked {
        ++idx;
    }
}
```

When the function finds an operator with `funded == stopped` it will pick that operator because `0 < operators[selectedOperatorIndex].funded - operators[selectedOperatorIndex].stopped`.

After the loop ends, `selectedOperatorIndex` will be the index of an operator that has no more validators to be funded (for this scenario). Because of this, the following code

```
uint256 selectedOperatorAvailableKeys = Uint256Lib.min(
            operators[selectedOperatorIndex].keys,
            operators[selectedOperatorIndex].limit
        ) - operators[selectedOperatorIndex].funded;
```

when executed on Op1 it will set `selectedOperatorAvailableKeys = 0` and as a result, the function will return `return (new bytes[](0), new bytes[](0));`.

In this scenario when `stopped==funded` and there are no keys available to be funded (`funded == min(limit, keys)`) the function will **always** return an empty result, breaking the `pickNextValidators` mechanism that won't be able to stake user's deposited ETH anymore even if there are operators with fundable validators.

Check the *Appendix* for a test case to reproduce this issue.

Recommendation: Alluvial should

- reimplement the logic of `Operators._hasFundableKeys` that should select only active operators with fundable keys without using the `stopped` attribute.
- reimplement the logic inside the `OperatorsRegistry._getNextValidatorsFromActiveOperators` loop to correctly pick the active operator with the higher number of fundable keys without using the `stopped` attribute.

Alluvial: Recommendation implemented in SPEARBIT/3.

Spearbit: Acknowledged.

6.2 High Risk

6.2.1 `Oracle.removeMember` could, in the same epoch, allow members to vote multiple times and other members to not vote at all

Severity: *High Risk*

Context: Oracle.1.sol#L213-L222

Description: The current implementation of `removeMember` is introducing an exploit that allows an oracle member to vote again and again (in the same epoch) and an oracle that has never voted is prevented from voting (in the same epoch).

Because of how `OracleMembers.deleteItem` is implemented, it will swap the last item of the array with the one that will be deleted and pop the last element.

Let's make an example:

1) At T0 `m0` to the list of members → `members[0] = m0`.
2) At T1 `m1` to the list of members → `members[1] = m1`.
3) At T3 `m0` call `reportBeacon(...)`. By doing that, `ReportsPositions.register(uint256(0));` will be called, registering that the member at `index 0` has registered the vote.
4) At T4, the oracle admin call `removeMember(m0)`. This operation, as we said, will swap the member's address from the last position of the array of members with the position of the member that will be deleted. After doing that will pop the last position of the array. The state changes from `members[0] = m0; members[1] = m1` to `members[0] = m1;`.

At this point, the oracle member `m1` will not be able to vote during this epoch because when he/she will call `reportBeacon(...)` the function will enter inside the check.

```
 if (ReportsPositions.get(uint256(memberIndex))) {
    revert AlreadyReported(_epochId, msg.sender);
}
```

This is because `int256 memberIndex = OracleMembers.indexOf(msg.sender);` will return `0` (the position of the m0 member that have already voted) and `ReportsPositions.get(uint256(0))` will return true.

At this point, if for whatever reason an admin of the contract add again the deleted oracle, it would be added to the position 1 of the array of the members, allowing the same member that have already voted, to vote again.

Note: while the scenario where a removed member can vote multiple time would involve a corrupted admin (that would re-add the same member) the second scenario that prevent a user to vote would be more common.

Check the *Appendix* for a test case to reproduce this issue.

Recommendation: After removing a member from `OracleMembers` remove also the information relative to the removed member from both `ReportsPositions` and `ReportsVariants`.

If the single removal is not possible, clear the entire state of both `ReportsPositions` and `ReportsVariants`. The consequence would be that all the active members would need to vote again for the same epoch.

Alluvial: Recommendation implemented in SPEARBIT/2.

Spearbit: Acknowledged.

Note: After removing an oracle member, all the "valid" members that have already voted for the epoch must vote again (everything is erased). We suggest documenting this behavior to clear any confusion for active oracle members.

6.2.2 Order of calls to `removeValidators` can affect the resulting validator keys set

Severity: *High Risk*

Context: OperatorsRegistry.1.sol#L310

Description: If two entities *A* and *B* (which can be either the `admin` or the operator *O* with the index *I*) send a call to `removeValidators` with 2 different set of parameters:

- $T_1 : (I, R_1)$
- $T_2 : (I, R_2)$

Then depending on the order of transactions, the resulting set of validators for this operator might be different. And since either party might not know a priori if any other transaction is going to be included on the blockchain after they submit their transaction, they don't have a 100 percent guarantee that their intended set of validator keys are going to be removed.

This also opens an opportunity for either party to DoS the other party's transaction by frontrunning it with a call to remove enough validator keys to trigger the `InvalidIndexOutOfBounds` error:

OperatorsRegistry.1.sol#L324-L326:

```
if (keyIndex >= operator.keys) {
    revert InvalidIndexOutOfBounds();
}
```

Recommendation: We can send a snapshot block parameter to `removeValidators` and compare it to a stored field for the operator and make sure there have not been any changes to the validator key set since that snapshot block. Alluvial has introduced such a mechanism for `setOperatorLimits` in 030b52feb5af2dd2ad23da0d512c5b0e55eb8259. A similar technique can be used here.

Alluvial: Don't think this is really an issue.

On a regular basis, the admin would not remove the keys but would request the Node Operator to remove the keys (because a key is unhealthy for example). In case a Node Operator refuses to remove the key (which is unexpected because this is would be against terms and conditions) then the admin could deactivate the operator and then remove the key without being exposed to the front run attack.

This is not as sensitive as the front run we had on `setOperatorLimit` because in this case, we are not making any keys eligible for funding. So the consequences are not this bad. Worst case the admin deactivates the node operator and there is no issue anymore.

Spearbit: We think the issue still needs to be documented both for the `admin` and also for the operators. Because in the scenario above both *A* and *B* can be the operator *O*. And *O* might send two transactions T_1, T_2 thinking T_1 would be applied to the state before T_2 (this might be unintentional, or intentional maybe because of something like out-of-gas issues). But it is possible that the order would be reversed and the end result would not be what the operator had expected. And if the operator would not check this, the issue can go unnoticed.

6.2.3 `_hasFundableKeys` marks operators that have no more fundable validators as fundable.

Severity: *High Risk*

Context: Operators.sol#L151-L152

Description: Since `operator.keys >= operator.limit` (based on the checks when `operator.keys` has been set), we can simplify `_hasFundableKeys`'s `return` expression to:

```
operator.active && operator.limit > (operator.funded - operator.stopped)
```

Also based on the assumption at *Non-zero `operator.limit` should always be greater than or equal to `operator.funded`* , if true:

```
operator.limit >= operator.funded
```

This means an active `operator` that has at least one `stopped` validator would pass the test (`_hasFundableKeys(operator) == true`)

```
(stop, funded, limit, keys) = (s, F, L, K)  // where s > 0
```

Even operators that have their `funded` equal to their `limit`:

```
(stop, funded, limit, keys) = (s, F, F, K) // where s > 0
```

Although they are maxed out for further funding. For these cases `_hasFundableKeys` returns `true`. So based on these findings it would make sense to have this function return:

```
operator.active && ( operator.limit > operator.funded )
```

Unless some other changes are applied to `OperatorResgistry.1.sol`, specially its `_getNextValidatorsFromActiveOperators` function.

Also, note that `funded`, `limit`, and `keys` are not only counters for the `operator`'s struct, but also they define ranges in `ValidatorKeys` for the `operator` (based on the way that they have been used in `OperatorResgistry.1.sol`). And this is one of the main differences between these 3 fields and the `stopped` field. The `stopped` field is only used as a counter.

Alluvial: Exactly! We shouldn't take `stopped` into account for the `_hasFundableKeys`, but we should keep it in the optimal operator search as we want to favor operators with the lowest count of running validators.

Spearbit: So basically `stopped` is used for maybe bookkeeping off-chain and also for the optimal search algorithm for picking the next validators.

Alluvial: Yes, currently it's more like a manual feature because exits should be rare but it will be a core part of the process once withdrawals are here and operators will often have to exit validators for users to redeem `ETH`. Recommendation implemented in SPEARBIT/3.

Spearbit: Acknowledged.

6.2.4 Non-zero `operator.limit` should always be greater than or equal to `operator.funded`

Severity: *High Risk*

Context: OperatorsRegistry.1.sol#L241, OperatorsRegistry.1.sol#L428-L430

Description: For the subtraction operation in OperatorsRegistry.1.sol#L428-L430 to not underflow and revert, there should be an assumption that

```
operators[selectedOperatorIndex].limit >= operators[selectedOperatorIndex].funded
```

Perhaps this is a general assumption, but it is not enforced when `setOperatorLimits` is called with a new set of limits.

Recommendation: Add a check in `setOperatorLimits` to enforce the new limits for the operators to be either `0` or in the range `[limit, keys]`.

If these assumptions are not correct, what would having `0 < limit < funded` signify? Also, what would setting the `limit` to `0` when `funded` is positive signify?

Alluvial: Implemented in SPEARBIT/3.

Spearbit: Acknowledged.

6.3 Medium Risk

6.3.1 Decrementing the quorum in `Oracle` in some scenarios can open up a frontrunning/backrunning opportunity for some oracle members

Severity: *Medium Risk*

Context:

- Oracle.1.sol#L338-L370
- Oracle.1.sol#L260
- Oracle.1.sol#L156 @ 030b52feb5af2dd2ad23da0d512c5b0e55eb8259

Description: Assume there are 2 groups of oracle members A, B where they have voted for report variants V_a and V_b respectively. Let's also assume the count for these variants C_a and C_b are equal and are the highest variant vote counts among all possible variants. If the Oracle `admin` changes the `quorum` to a number less than or equal to $C_a + 1 = C_b + 1$, any oracle member can backrun this transaction by the admin to decide which report variant V_a or V_b gets pushed to the `River`. This is because when a lower `quorum` is submitted by the `admin` and there exist two variants that have the highest number of votes, in the `_getQuorumReport` function the returned `isQuorum` parameter would be `false` since `repeat == 0` is `false`:

Oracle.1.sol#L369:

```
return (maxval >= _quorum && repeat == 0, variants[maxind]);
```

Note that this issue also exists in the commit hash `030b52feb5af2dd2ad23da0d512c5b0e55eb8259` and can be triggered by the admin either by calling `setQuorum` or `addMember` when the abovementioned conditions are met.

Also, note that the free oracle member agent can frontrun the `admin` transaction to decide the `quorum` earlier in the scenario above. Thus this way `_getQuorumReport` would actually return that it is a `quorum`.

Recommendation: This issue is similar to *The `reportBeacon` is prone to front-running attacks by oracle members*.

We recommend always checking that after the modifications are done to the set of oracle members, setting the new `quorum` would respect the $Q > \frac{C+1}{2}$ invariant. Here Q is the `quorum` and C is the number of all oracle members. Note that the invariant provided here is slightly stronger than the one in issues/54 since we are assuming the last oracle member to vote can go either way and doesn't need to belong to a particular group. issues/54 could also have been worded that way and the invariant would have been $Q > \frac{C+1}{2}$ (also note this only applies when $C1$).

Alluvial: The solution provided on liquid-collective/liquid-collective-protocol#151 should be solving the issue (clearing report data whenever member is added or removed + quorum is changed). Not sure how the new proposed quorum invariant solves the issue.

Spearbit: After a quick test (located in *Appendix*) and think that with your changes the contract will push to river the report of a removed member.

Scenario:

- member1 added `oracle.addMember(om1, 1);`.
- member2 added `oracle.addMember(om1, 2);`.
- member2 call `reportConsensusLayerData`, `_pushToRiver` is not called because we have a quorum of 2.
- member2 is removed because it's a malicious oracle member `oracle.removeMember(om2, 1);` this action will trigger a `_pushToRiver` because reports are cleared at the end of `_setQuorumAndClearReports`.
- as a consequence, the vote of a malicious oracle has triggered `_pushToRiver` with probably a malicious report.

Alluvial: After discussing with the team, we decided to always clear the reports whenever we add a member, remove a member or change the quorum. We removed the logic that would check if a report could be forwarded to River from the _setQuorum internal method (that has been renamed to _clearReportsAndSetQuorum). So from now on we expect oracle operators to resubmit their votes whenever on of the three actions happens.

Spearbit:

> The solution provided on liquid-collective/liquid-collective-protocol#151 should be solving the issue (clearing report data whenever member is added or removed + quorum is changed). Not sure how the new proposed quorum invariant solves the issue.

The proposed inequality check would have solved the issue and also would have fixed:

- *The `reportBeacon` is prone to front-running attacks by oracle members*

But with the current implementation of

- PR #151

This issue is solved, since in the `_setQuorum` we don't check for `isQuorum` and also when adding/removing an oracle member or setting the `quorum` we also clear the oracle member reports.

The acceptable range for Q the `quorum` is $Q \in \frac{C+1}{2}$, C to fix *The `reportBeacon` is prone to front-running attacks by oracle members*. The current invariant check:

```
(_newQuorum == 0 && memberCount > 0) || _newQuorum > memberCount
```

can be changed to:

```
(
   memberCount > 1 && !(2 * _newQuorum > (memberCount + 1))
) ||
(
   memberCount > 0 && _newQuorum == 0
) ||
(
   memberCount < _newQuorum
)
```

Again, if this new inequality is not going to be checked on-chain, it should be documented and also checked off-chain.

Alluvial: We are fine not constraining the quorum, We added a comment about it in the natspec.

Spearbit: Acknowledged.

6.3.2 `_getNextValidatorsFromActiveOperators` can be tweaked to find an operator with a better validator pool

Severity: *Medium Risk*

Context: OperatorsRegistry.1.sol#L417-L420

Description: Assume for an operator:

```
(A, B) = (funded - stopped, limit - funded)
```

The current algorithm finds the first index in the cached operators array with the minimum value for `A` and tries to gather as many `publicKey`s and `signature`s from this operator's validators up to a max of `_requestedAmount`. But there is also the `B` cap for this amount. And if `B` is zero, the function returns early with empty arrays. Even though there could be other approved and non-funded validators from other operators.

Related: *`OperatorsRegistry._getNextValidatorsFromActiveOperators` should not consider `stopped` when picking a validator*, *`OperatorsRegistry._getNextValidatorsFromActiveOperators` can DOS Alluvial staking if there's an operator with `funded==stopped` and `funded == min(limit, keys)`*, *`_hasFundableKeys` marks operators that have no more fundable validators as fundable.*

Recommendation: A better search algorithm would be to try to find an operator by minimizing `A` but also maximizing `B`. But the only linear cost function that would avoid this shortfall above is `-B`. If the minimum of `-B` all operators is `0`, then we can conclude that for all operators `B == 0` and thus `limit == funded` for all operators. So no more approved fundable validators left to select. Note, that we can also look at non-linear cost functions or try to change the picking algorithm in a different direction.

Alluvial: It's ok to pick the operator with the least running validators as the best one even if he doesn't have a lot of fundable keys. The check for fundable should be edited to not take stopped into account, that way we can be sure that the cached operator list contains operators with at least 1 fundable key and focus entirely on finding the operator with the lowest active validator count. The end goal of the system is to even the validator count of all operators. Of course limits will be set to similar values but new operators with low validator numbers should be prioritised to catch up with the rest of the operators.

Spearbit: We can also tweak the current algorithm to not just pick the 1st found operator with the lowest non-stopped funded number of validators, but pick one amongst those that also have the highest approved non-funded validators. Basically with my notations from the previous comment, among the operators with the lowest `A`, pick the one with the highest `B`.

We can tweak the search algorithm to favor/pick an `operator` with the highest number of allowed non-funded validators amongst the operators with the lowest number of non-stopped funded validators (As a side effect, this change also has negative net gas on all tests, gas is more saved).

```
uint256 selectedFunded = operators[selectedOperatorIndex].funded;
uint256 currentFunded = operators[idx].funded;

uint256 selectedNonStoppedFundedValidators = (
    selectedFunded -
    operators[selectedOperatorIndex].stopped
);
uint256 currerntNonStoppedFundedValidators = (
    currentFunded -
    operators[idx].stopped
);

bool equalNonStoppedFundedValidators = (
    currerntNonStoppedFundedValidators ==
    selectedNonStoppedFundedValidators
);

bool hasLessNonStoppedFundedValidators = (
    currerntNonStoppedFundedValidators <
    selectedNonStoppedFundedValidators
);

bool hasMoreAllowedNonFundedValidators = (
    operators[idx].limit - currentFunded >
    operators[selectedOperatorIndex].limit - selectedFunded
);

if (
    hasLessNonStoppedFundedValidators ||
    (
        equalNonStoppedFundedValidators &&
        hasMoreAllowedNonFundedValidators
    )
) {
    selectedOperatorIndex = idx;
}
```

Spearbit: The picking algorithm has been changed in PR SPEARBIT/3 slightly by adding a `MAX_VALIDATOR_ATTRIBUTION_PER_ROUND` per round and keeping track of the `picked` number of validators for an operator. Thus the recommendation in this issue is not fully implemented, but the early return issue has been fixed.

So the new operator picking algorithm is changed to (below the subscripts f refers to `funded`, p `picked`, s `stopped`, l `limit`):

$$o^* = \underset{o \in \text{Operators}}{\operatorname{argmin}} \{o_f + o_p - o_s \mid o_a \wedge (o_l > o_f + o_p)\}$$

And once the operator o is selected, we pick the number of validation keys based on:

$$o^* = \underset{o \in \text{Operators}}{\operatorname{argmin}} \{o_f + o_p - o_s \mid o_a \wedge (o_l > o_f + o_p)\}$$

That means for each round we pick maximum `MAX_VALIDATOR_ATTRIBUTION_PER_ROUND = 5` validator keys. There could be also scenarios where each operator $o_l - (o_f + o_p) = 1$, which means at each round we pick exactly 1 key. Now, if `count` is a really big number, we might run into out-of-gas issues.

One external call that triggers `_pickNextValidatorsFromActiveOperators` is `ConsensusLayerDepositManagerV1.depositToConsensusLayer` which currently does not have any access control modifier. So anyone can call into it. But from the test files, it seems like it might be behind a firewall that only an executor might have

permission to call into. Is that true that `depositToConsensusLayer` will always be behind a firewall? In that case, we could move the picking/distribution algorithm off-chain and modify the signature of `depositToConsensusLayer` to look like:

```
function depositToConsensusLayer(
   uint256[] calldata operatorIndexes,
   uint256[] calldata validatorCounts
) external
```

So we provide `depositToConsensusLayer` with an array of operators that we want to fund and also per operator the number of new validator keys that we would like to fund. Note that this would also help with the out-of-gas issue mentioned before. Why is the picking/distribution algorithm currently on-chain? Is it because it is kind of transparent for the operators how their validators get picked?

Alluvial: we would like to keep things that should be transparent for end users inside the contract. Knowing that the validators are properly split between all operators is important as operator diversification is one of the advertised benefits of liquid staking.

6.3.3 Dust might be trapped in `WlsETH` when burning one's balance.

Severity: *Medium Risk*

Context: WLSETH.1.sol#L140

Description: It is not possible to burn the exact amount of minted/deposited `lsETH` back because of the `_value` provided to `burn` is in `ETH`. Assume we've called `mint(r,v)` with our address `r`, then to get the `v lsETH` back to our address, we need to find an x where $v = \lfloor x\frac{S}{B} \rfloor$ and call `burn(r, x)` (Here S represents the total share of `lsETH` and B the total underlying value.).

It's not always possible to find the exact `x`. So there will always be an amount locked in this contract ($v - \lfloor x\frac{S}{B} \rfloor$). These dust amounts can accumulate from different users and turn into a big number. To get the full amount back, the user needs to mint more `wlsETH` tokens so that we can find an exact solution to $v = \lfloor x\frac{S}{B} \rfloor$. The extra amount to get the locked-up fees back can be engineered.

The same problem exists for `transfer` and `transferFrom`.

Also note, if you have minted x amount of shares, the `balanceOf` would tell you that you own $b = \lfloor \frac{xB}{S} \rfloor$ `wlsETH`. Internally `wlsETH` keeps track of the shares x. So users think they can only `burn` b amount, plug that in for the `_value` and in this case, the number of shares burnt would be

$$\left\lfloor \frac{\lfloor \frac{xB}{S} \rfloor S}{B} \right\rfloor$$

which has even more rounding errors. `wlsETH` could internally track the underlying but that would not appropriate value like `lsETH`, which would basically be kind of `wETH`. We think the issue of not being able to `transfer` your full amount of shares is not as serious as not being able to `burn` back your shares into `lsETH`.

On the same note, we think it would be beneficial to expose the `wlsETH` share amount to the end user:

```
function sharesBalanceOf(address _owner) external view returns (uint256 shares) {
    return BalanceOf.get(_owner);
}
```

Recommendation: Allow the `burn` function to use the share amount as the unit of `_value` instead of avoiding locking up these dust amounts.

Alluvial: Recommendation implemented in PR SPEARBIT/5.

Spearbit: Acknowledged.

6.3.4 `BytesLib.concat` can potentailly return results with dirty byte paddings.

Severity: *Medium Risk*

Context: BytesLib.sol#L23

Description: `concat` does not clean the potential dirty bytes that might have been copied from `_postBytes` (nor does it clean the padding). The dirty bytes from `_postBytes` are carried over to the padding for `tempBytes`.

Recommendation: Consider modifiying the code to clean the byte padding or if the gas cost is not an issue you can replace the usage of `BytesLib.concat` in the codebase by `bytes.concat`.

Alluvial: Resolved in SPEARBIT/6.

Spearbit: Acknowledged.

6.3.5 The `reportBeacon` is prone to front-running attacks by oracle members

Severity: *Medium Risk*

Context: Oracle.1.sol#L289

Description: There could be a situation where the oracle members are segmented into 2 groups A and B , and members of the group A have voted for the report variant V_a and the group B for V_b . Also, let's assume these two variants are `1` vote short of `quorum`. Then either group can try to front-run the other to push their submitted variant to `river`.

Recommendation: Assume that

- Q is the `quorum`
- C is the total number of all oracle members
- C_a is the number of oracle members you have voted for V_a
- C_b is the number of oracle members you have voted for V_b

then in the scenario described we have:

$$C_a = C_b = Q - 1$$

Since for this frontrunning race to work we need one more oracle member in each group, we would have

$$2Q = (C_a + 1) + (C_b + 1) \leq C \Rightarrow Q \leq \frac{C}{2}$$

So to mitigate this issue we would need to make sure $Q\frac{C}{2}$. This invariant can be checked every time we modify the `quorum` or add a new oracle member. If we look at the last oracle member to vote differently as a free agent that can vote either way (so that we don't need two members to race against each other) then like the issue:

- *Decrementing the quorum in `Oracle` in some scenarios can open up a frontrunning/backrunning opportunity for some oracle members*

A better invariant would be:

$$Q\frac{C+1}{2}$$

Alluvial: Yes, this is a risk from the oracle that we could mitigate by having a large oracle member count + a high quorum to make collusion as hard as possible. Also, the sanity checks inside the oracle prevent very unusual updates of the validator balance sum so even if a collusion occurs they are bound to alter the balance up to what the sanity checks are allowing them, reducing the economical incentive behind such a move (is the max allowed delta worth it if you get removed from the system afterward?)

6.3.6 Avoid multiple divisions when calculating `operatorRewards`

Severity: *Medium Risk*

Context: River.1.sol#L277

Description/Recommendation: In _onEarnings, we calculate the `sharesToMint` and `operatorRewards` by diving 2 numbers. We can reduce the number of divisions to `1` and also delegate this division to `_rewardOperators`. This would further avoid the rounding errors that we would get when we divide two numbers in `EVM`. So in:

```
uint256 globalFee = GlobalFee.get();
uint256 numerator = _amount * currentTotalSupply * globalFee;
uint256 denominator = (_assetBalance() * BASE) - (_amount * globalFee);
uint256 sharesToMint = denominator == 0 ? 0 : (numerator / denominator);

uint256 operatorRewards = (sharesToMint * OperatorRewardsShare.get()) / BASE;

uint256 mintedRewards = _rewardOperators(operatorRewards);
```

Instead of passing `operatorRewards` we can pass 2 values, one for the numerator and one for the denominator. This way we can avoid extra rounding errors introduced in `_rewardOperators`. `_rewardOperators` also need to be changed slightly to account for these 2 new values.

```
uint256 globalFee = GlobalFee.get();
uint256 numerator = _amount * currentTotalSupply * globalFee * OperatorRewardsShare.get();
uint256 denominator = ((_assetBalance() * BASE) - (_amount * globalFee)) * BASE;

uint256 mintedRewards;

if(denominator != 0) { // note: this was added to avoid calling `_rewardOperators` if `denominator == 0`
   mintedRewards = _rewardOperators(numerator, denominator);
}
```

Without this correction, the rounding errors are in favor of the general users/stakers and the `treasury` of the `River` protocol (not the operators).

Alluvial: The whole operator rewarding system has been removed in SPEARBIT/8.

Spearbit: Acknowledged.

6.3.7 Shares distributed to operators suffer from rounding error

Severity: *Medium Risk*

Context: River.1.sol#L238

Description: `_rewardOperators` distribute a portion of the overall shares distributed to operators based on the number of active and funded validators that each operator has.

The current number of shares distributed to a validator is calculated by the following code

```
_mintRawShares(operators[idx].feeRecipient, validatorCounts[idx] * rewardsPerActiveValidator);
```

where `rewardsPerActiveValidator` is calculated as

```
uint256 rewardsPerActiveValidator = _reward / totalActiveValidators;
```

This means that in reality each operator receives `validatorCounts[idx] * (_reward / totalActiveValidators)` shares. Such share calculation suffers from a rounding error caused by division before multiplication.

Recommendation: Consider re-writing the number of shares distributed to each operator:

```
// removed ---> uint256 rewardsPerActiveValidator = _reward / totalActiveValidators;

for (uint256 idx = 0; idx < validatorCounts.length;) {
    _mintRawShares(
       operators[idx].feeRecipient,
       (validatorCounts[idx] * _reward) / totalActiveValidators;
    );
    ...
```

Note that this will reduce the rounding error, but it adds 1 `DIV` gas cost (`5` gas) per iteration. Also, the rounding errors favors the users/depositors.

Alluvial: The whole operator rewarding system has been removed in SPEARBIT/8.

Spearbit: Acknowledged.

6.3.8 `OperatorsRegistry._getNextValidatorsFromActiveOperators` should not consider `stopped` when picking a validator

Severity: *Medium Risk*

Context: OperatorsRegistry.1.sol#L403-L454

Description: Note that

- `limited` → number of validators (already pushed by op) that have been approved by Alluvial and can be selected to be funded.
- `funded` → number of validators funded.
- `stopped` → number of validators exited (so that were funded at some point but for any reason they have exited the staking).

The implementation of the function should favor operators that have the highest number of available validators to be funded. Nevertheless functions favor validators that have `stopped` value near the `funded` value.

Consider the following example:

```
Op at index 0
name op1
active true
limit 10
funded 5
stopped 5
keys 10

Op at index 1
name op2
active true
limit 10
funded 0
stopped 0
keys 10
```

1) op1 and op2 have 10 validators whitelisted.
2) op1 at time1 get 5 validators funded.
3) op1 at time2 get those 5 validators exited, this mean that `op.stopped == 5`.

In this scenario, those 5 validators would not be used because they are "blacklisted".

At this point

- op1 have 5 validators that can be funded.

- op2 have 10 validators that can be funded.

`pickNextValidators` logic should favor operators that have the higher number of available keys (not funded but approved) to be funded.

If we run `operatorsRegistry.pickNextValidators(5);` the result is this

```
Op at index 0
name op1
active true
limit 10
funded 10
stopped 5
keys 10

Op at index 1
name op2
active true
limit 10
funded 0
stopped 0
keys 10
```

Op1 gets all the remaining 5 validators funded, the function (from the specification of the logic) should instead have picked Op2.

Check the *Appendix* for a test case to reproduce this issue.

Recommendation: Alluvial should:

- reimplement the logic of `Operators. _hasFundableKeys` that should select only active operators with fundable keys without using the `stopped` attribute.
- reimplement the logic inside the `OperatorsRegistry. _getNextValidatorsFromActiveOperators` loop to correctly pick the active operator with the higher number of fundable keys without using the `stopped` attribute.

Alluvial: Recommendation implemented in SPEARBIT/3. While `stopped` is not used anymore to gather the list of active and fundable operators, it's still used in the sorting algorithm. As a result, it could happen that operators with `stopped > 0` get picked before operators that have fundable keys but `stopped === 0`.

Spearbit: Acknowledged.

6.3.9 `approve()` function can be front-ran resulting in token theft

Severity: *Medium Risk*

Context: SharesManager.1.sol#L143, WLSETH.1.sol#L116-L120

Description: The `approve()` function has a known race condition that can lead to token theft. If a user calls the approve function a second time on a spender that was already allowed, the spender can front-run the transaction and call `transferFrom()` to transfer the previous value and still receive the authorization to transfer the new value.

Recommendation: Consider implementing functionality that allows a user to increase and decrease their allowance similar to Lido's implementation. This will help prevent users losing funds from front-running attacks.

Alluvial: Recommendation implemented in SPEARBIT/9.

Spearbit: Acknowledged. Note: if you want to follow the same logic of OpenZeppelin ERC20 implementation, the `_spendAllowance` in both `SharesManager` and `WLSETH` should execute `emit Approval(owner, spender, amount);`.

Alluvial: Fixed in PR 151.

6.3.10 Add missing input validation on constructor/initializer/setters

Severity: *Medium Risk*

Description: `Allowlist.1.sol`

- initAllowlistV1 should require the `_admin` parameter to be not equal to `address(0)`. This check is not needed if issue *`LibOwnable._setAdmin` allows setting `address(0)` as the admin of the contract* is implemented directly at `LibOwnable._setAdmin` level.
- allow should check `_accounts[i]` to be not equal to `address(0)`.

`Firewall.sol`

- constructor should check that: `governor_ != address(0). executor_ != address(0). destination_ != address(0).`
- setGovernor should check that `newGovernor != address(0).`
- setExecutor should check that `newExecutor != address(0).`

`OperatorsRegistry.1.sol`

- initOperatorsRegistryV1 should require the `_admin` parameter to be not equal to `address(0)`. This check is not needed if issue *`LibOwnable._setAdmin` allows setting `address(0)` as the admin of the contract* is implemented directly at `LibOwnable._setAdmin` level.
- addOperator should check: `_name` to not be an empty string. `_operator` to not be `address(0)`. `_feeRecipient` to not be `address(0)`.
- setOperatorAddress should check that `_newOperatorAddress` is not `address(0)`.
- setOperatorFeeRecipientAddress should check that `_newOperatorFeeRecipientAddress` is not `address(0)`.
- setOperatorName should check that `_newName` is not an empty string.

`Oracle.1.sol`

- initOracleV1 should require the `_admin` parameter to be not equal to `address(0)`. This check is not needed if issue *`LibOwnable._setAdmin` allows setting `address(0)` as the admin of the contract* is implemented directly at `LibOwnable._setAdmin` level. Consider also adding some min and max limit to the values of `_annualAprUpperBound` and `_relativeLowerBound` and be sure that `_epochsPerFrame`, `_slotsPerEpoch`, `_secondsPerSlot` and `_genesisTime` matches the values expected.
- addMember should check that `_newOracleMember` is not `address(0)`.
- setBeaconBounds: Consider adding min/max value that `_annualAprUpperBound` and `_relativeLowerBound` should respect.

`River.1.sol`

- initRiverV1:

`_globalFee` should follow the same validation done in `setGlobalFee`. Note that client said that **0** is a valid `_globalFee` value

> "The revenue redistributuon would be computed off-chain and paid by the treasury in that case. It's still an on-going discussion they're having at Alluvial."

`_operatorRewardsShare` should follow the same validation done in `setOperatorRewardsShare`. Note that client said that **0** is a valid `_operatorRewardsShare` value

> "The revenue redistributuon would be computed off-chain and paid by the treasury in that case. It's still an on-going discussion they're having at Alluvial."

`ConsensusLayerDepositManager.1.sol`

- initConsensusLayerDepositManagerV1: `_withdrawalCredentials` should not be empty and follow the requirements expressed in the following official Consensus Specs document.

Recommendation: Consider implementing all the checks suggested above.

Alluvial: Recommendation implemented in SPEARBIT/10. Some validation checks like on the `admin` not being `address(0)` will be addressed in another PR.

Spearbit: Only empty check is performed on `_withdrawalCredentials`. `_annualAprUpperBound` and `_relativeLowerBound` are still not checked in both`initOracleV1` and `setReportBounds`.

6.3.11 `LibOwnable._setAdmin` allows setting `address(0)` as the admin of the contract

Severity: *Medium Risk*

Context: LibOwnable.sol#L8-L10

Description: While other contracts like `RiverAddress` (for example) do not allow `address(0)` to be used as `set` input parameter, there is no similar check inside `LibOwnable._setAdmin`.

Because of this, contracts that call `LibOwnable._setAdmin` with `address(0)` will not revert and functions that should be callable by an admin cannot be called anymore.

This is the list of contracts that import and use the `LibOwnable` library

- `AllowlistV1`
- `OperatorsRegistryV1`
- `OracleV1`
- `RiverV1`

Recommendation: Consider adding a check inside `LibOwnable._setAdmin` to prevent setting `address(0)` as the admin, or move that specific check in each contract that import and use `LibOwnable`.

Alluvial: Recommendation implemented in SPEARBIT/11.

Spearbit: Note 1: Still missing (client said it will be implemented in other PRs)

- `Administrable` miss all natspec comments
- Event for `_setAdmin` are still missing but will be added to `Initializable` event in another PR

Note 2: Client has acknowledged that all the contracts that inherit from `Administrable` have the ability to transfer ownership, even contracts like `AllowlistV1` that didn't have the ability before this PR.

Alluvial: Issues in Note 1 addressed in SPEARBIT/33.

Spearbit: Acknowledged.

6.3.12 `OracleV1.getMemberReportStatus` returns true for non existing oracles

Severity: *Medium Risk*

Context: Oracle.1.sol#L115-L118

Description: `memberIndex` will be equal to `-1` for non-existing oracles, which will cause the `mask` to be equal to `0`, which will cause the function to return `true` for non-existing oracles.

Recommendation: Consider changing the function to return false for `memberIndex = -1` but bear in mind that if this function is used directly inside some part of the logic it could allow a not existing member to vote. In this case, the best solution is to always check if the member does not exist by checking if `memberIndex >= 0`.

The function could otherwise `revert` if the member does not exist, and return `true/false` if it does exist and has voted/not voted. If this solution is chosen, remember that if integrated directly into the code could create a DOS scenario if not handled correctly.

For all these reasons, consider properly documenting the behavior of the function and the possible side effects in the natspec comment.

Alluvial: Fixed in SPEARBIT/12 by returning `false` for non-existing oracles.

Spearbit: Acknowledged.

6.3.13 Operators might add the same validator more than once

Severity: *Medium Risk*

Context: OperatorsRegistry.1.sol#L270

Description: Operators can use `OperatorsRegistryV1.addValidators` to add the same validator more than once. Depositors' funds will be directed to these duplicated addresses, which in turn, will end up having more than 32 eth. This act will damage the capital efficiency of the entire deposit pool and thus will potentially impact the pool's APY.

Recommendation: Consider adding a de-duplication mechanism.

Alluvial: Acknowledged.

6.3.14 `OracleManager.setBeaconData` possible front running attacks

Severity: *Medium Risk*

Context: River.1.sol#L27

Description: The system is designed in a way that depositors receive shares (`lsETH`) in return for their eth deposit. A share represents a fraction of the total eth balance of the system in a given time. Investors can claim their staking profits by withdrawing once withdrawals are active in the system. Profits are being pulled from `ELFeeRecipient` to the `River` contract when the oracle is calling `OracleManager.setBeaconData`. `setBeaconData` updates `BeaconValidatorBalanceSum` which might be increased or decreased (as a result of slashing for instance).

Investors have the ability to time their position in two main ways:

- Investors might time their deposit just before profits are being distributed, thus harvesting profits made by others.
- Investors might time their withdrawal / sell `lsETH` on secondary markets just before the loss is realized. By doing this, they will effectively avoid the loss, escaping the intended mechanism of socializing losses.

Recommendation: As for the first issue, we recommend considering replacing the accounting logic with a different model that takes into account the timing of the deposit. In this case, as communicated with the team, oracles are supposed to call `OracleManager.setBeaconData` on a daily basis, which will limit the impact of such a strategy. However, we do recommend the off-chain monitoring of oracles' activity to make sure that the impact of this issue stays limited.

The second issue will have to be tackled once there is more certainty about the withdrawal process.

Alluvial: Resolved in SPEARBIT/13

Spearbit: It will solve the issue but will introduce further reliance on oracles, and will make it harder to track their performance in general.

Alluvial: It indeed adds more opacity around operators but all the keys are inside the contract and anyone could map the keys to the operators and track their performance. For now, we decided to move the operator rewards outside of River and we will use an approach that is similar to what we were doing on-chain at first, but we will work on adding more granularity to the rewards by taking into account how efficient their validators are. Adding this on-chain would make everything a lot more complicated, the oracle reports would need to perform a huge ton of extra work (we prefer having simple oracles that operators and integrators could easily run so we increase the quorum as the protocol grows than requiring bigger hardware requirements and making it a pain to operate)

Spearbit: Acknowledged.

6.3.15 `SharesManager._mintShares` - Depositors may receive zero shares due to front-running

Severity: *Medium Risk*

Context: SharesManager.1.sol#L202

Description: The number of shares minted to a depositor is determined by `(_underlyingAssetValue * _totalSupply()) / oldTotalAssetBalance`. Potential attackers can spot a call to `UserDepositManagerV1._deposit` and front-run it with a transaction that sends wei to the contract (by self-destructing another contract and sending the funds to it), causing the victim to receive fewer shares than what he expected. More specifically, In case `oldTotalAssetBalance()` is greater than `_underlyingAssetValue * _totalSupply()`, then the number of shares the depositor receives will be 0, although `_underlyingAssetValue` will be still pulled from the depositor's balance.

An attacker with access to enough liquidity and to the mem-pool data can spot a call to `UserDepositManagerV1._deposit` and front-run it by sending at least `totalSupplyBefore * (_underlyingAssetValue - 1) + 1` wei to the contract . This way, the victim will get 0 shares, but `_underlyingAssetValue` will still be pulled from its account balance. In this case, the attacker does not necessarily have to be a whitelisted user, and it is important to mention that the funds that were sent by him can not be directly claimed back, rather, they will increase the price of the share.

The attack vector mentioned above is the general front runner case, the most profitable attack vector will be the case where the attacker is able to determine the share price (for instance if the attacker mints the first share). In this scenario, the attacker will need to send at least `attackerShares * (_underlyingAssetValue - 1) + 1` to the contract, (`attackerShares` is completely controlled by the attacker, and thus can be `1`). In our case, depositors are whitelisted, which makes this attack harder for a foreign attacker.

Recommendation: Consider adding a validation check enforcing that `sharesToMint > 0`.

Spearbit: The fix presented in issue *An attacker can freeze all incoming deposits and brick the oracle members' reporting system with only `1 wei`* also fixes this vulnerability.

6.4 Low Risk

6.4.1 Orphaned (index, values) in `SlotOperator` storage slots in `operatorsRegistry`

Severity: *Low Risk*

Context: Operators.sol#L261-L263

Description: If `!opExists` corresponds to an operator which has `OperatorResolution.active` set to `false`, the line below can leave some orphaned (index, values) in `SlotOperator` storage slots:

```
_setOperatorIndex(name, newValue.active, r.value.length - 1);
```

Recommendation: In the case `!opExists` corresponds to an operator which has `OperatorResolution.active` set to `false`, we can clear the storage for those operators. Unless, Alluvial would like to keep a record of those operators on-chain.

Alluvial: Issue does not exist anymore given that the referenced code has been removed in SPEARBIT/14.

Spearbit: Acknowledged.

6.4.2 A malicious operator can purposefully mismanage its validators to benefit from `lsETH` market price movements.

Severity: *Low Risk*

Context: operatorsRegistry/Operators.sol#L11

Description: At some point, an `operator` might find it beneficial to let its validators get slashed or just turn them off. Meanwhile short on a `lsETH / WlsETH` position on a market. The effect of this operator's validators getting slashed can move the market for those coins. If the operator would also be able to calculate and time this market position move precisely, they can take flash-loan to even gain a higher leverage/reward.

Related: LID-02, 5.4 Malicious Node Operators

Alluvial: Acknowledged.

6.4.3 Warn the `admin` and the `operator` that `setOperatorName` has other side-effects

Severity: *Low Risk*

Context: OperatorsRegistry.1.sol#L196

Description: Need to warn the `admin` and the `operator` that `setOperatorName` sets the `OperatorResolution.active` to `true` for this `name` as a side-effect. This warning is more important for the `admin` since only active operators can call into this function.

Alluvial: Resolved in SPEARBIT/14. by removing this function.

Spearbit: Acknowledged.

6.4.4 `OperatorsRegistry.setOperatorName` Possible front running attacks

Severity: *Low*

Context: OperatorsRegistry.1.sol#L196-L204

Description:

1. `setOperatorName` reverts for an already used name, which means that a call to `setOperatorName` might be front-ran using the same name. The front-runner can launch the same attack again and again thus causing a DoS for the original caller.
2. `setOperatorName` can be called either by an operator (to edit his own name) or by the admin. `setOperatorName` will revert for an already used `_newName`. `setOperatorName` caller might be front-ran by the identical transaction transmitted by someone else, which will lead to failure for his transaction, where in practice this failure is a "false failure" since the desired change was already made.

Recommendation:

1. Consider changing the current mechanism to be based on a commit-reveal scheme. The idea is to add a `commitName` function that will store `hash(msg.sender, _newName)` for the `msg.sender`. This function will have to be called before the call to `setOperatorName`, which will implement the "reveal" side, validating that `hash(msg.sender, _newName)` exists, and if so, change the previous name to the new name, and clears the value previously stored in `commitName`.
2. The off-chain code that calls `setOperatorName` should handle this case by querying the name of the operator even when the transaction fails, if it is the desired name, then it should be considered successful.

Alluvial: Fixed in SPEARBIT/14.

Spearbit: The provided PR resolves the issues above, but it allows duplicated names. Are you ok with that?

Alluvial: yes, we completely removed the name unicity and are now working entirely with indexes as unique identifiers. Names are considered as informational off chain data now. it also makes things a lot easier to read imo.

Spearbit: Acknowledged.

6.4.5 Prevent users from burning token via `lsETH/wlsETH` `transfer` or `transferFrom` functions

Severity: *Low Risk*

Context: SharesManager.1.sol#L158-L165, WLSETH.1.sol#L157-L165

Description: The current implementation of both `lsETH` (`SharesManager` component of `River` contract) and `wlsETH` allow the user to "burn" tokens, sending them directly to the `address(0)` via the `transfer` and `transferFrom` function.

By doing that, it would bypass the logic of the existing `burn` functions present right now (or in the future when withdrawals will be enabled in River) in the protocol.

Recommendation: Add a check to both `_transfer` functions to prevent the user to transfer tokens to the `address(0)` Consider also to adopt the same checks on `address(0)` done by the OpenZeppelin ERC20 implementation, where needed and possible.

Alluvial: Recommendation implemented in SPEARBIT/9.

Spearbit: Acknowledged.

6.5 Gas Optimization

6.5.1 In `addOperator` when emitting an event use stack variables instead of reading from memory again

Severity: *Gas Optimization*

Context: OperatorsRegistry.1.sol#L161

Description: In `OperatorsRegistry`'s `addOperator` function when emitting the `AddedOperator` event we read from memory all the event parameters except `operatorIndex`.

```
emit AddedOperator(operatorIndex, newOperator.name, newOperator.operator, newOperator.feeRecipient);
```

We can avoid reading from memory to save gas.

Recommendation: Use the stack variables already defined in the scope.

```
emit AddedOperator(operatorIndex, _name, _operator, _feeRecipient);
```

This would also be consistent with other events emitted in the functions defined in `OperatorsRegistry`.

Alluvial: Implemented in PR 151. Implementation is slightly different since `_feeRecipient` has been removed from the `Operators.Operator` struct.

Spearbit: Acknowledged.

6.5.2 Avoid slicing and concatenating by passing a different pattern of bytes to `addValidators`

Severity: *Gas Optimization*

Context: OperatorsRegistry.1.sol#L270, OperatorsRegistry.1.sol#L289-L292

Description/Recommendation:

We can pass `bytes calldata _publicKeys, bytes calldata _signatures` also concatenated like

```
<PUBKEY_0> <SIG_0> <PUBKEY_1> <SIG_1> ...
```

And call that `bytes calldata _publicKeysAndSignatures`. This way we would only slice once in the `for` loop. Also, we can modify the `ValidatorKeys.set` to accept a concatenated `publicKey` and `signatures`. That way we can also avoid the concatenation that happens again in `ValidatorKeys.set`.

ValidatorKeys.sol#L64-L65

```
function set(uint256 operatorIndex, uint256 idx, bytes memory publicKey, bytes memory signature)
↪  internal {
    bytes memory concatenatedKeys = BytesLib.concat(publicKey, signature);
```

Alluvial: Implemented in SPEARBIT/24.

Spearbit: Acknowledged.

6.5.3 Cache `_indexes[_indexes.length - 1]` in `removeValidators` to avoid reading from storage multiple times.

Severity: *Gas Optimization*

Context: OperatorsRegistry.1.sol#L344-L345

Description/Recommendation: Cache `_indexes[_indexes.length - 1]` into a new variable to read it only once.

```
testExecutorCanSetOperatorLimit() (gas: -1 (-0.000%))
testGovernorCanSetOperatorLimit() (gas: -1 (-0.000%))
Overall gas change: -2 (-0.000%)
```

```
uint256 lastIndex = _indexes[_indexes.length - 1];
if (lastIndex < operator.limit) {
    operator.limit = lastIndex;
}
```

Alluvial: Recommendation implemented in PR SPEARBIT/24.

Spearbit: Acknowledged.

6.5.4 Avoid updating `operator.keys` storage slot inside of a loop in `removeValidators`

Severity: *Gas Optimization*

Context: OperatorsRegistry.1.sol#L310-L347

Description/Recommendation: Along with other minor gas optimizations, we can take `operator.keys` updates outside of the `for` loop to avoid updating this storage slot multiple times. The update can happen outside of the loop.

```
testBurnWrappedTokensWithRebase(uint256,uint32) (gas: -2 (-0.000%))
testBurnWrappedTokensInvalidTransfer(uint256,uint32) (gas: 2 (0.000%))
testBurnWrappedTokens(uint256,uint32) (gas: -2 (-0.000%))
testTransferFrom(uint256,uint256,uint256,uint32) (gas: 4 (0.000%))
testBalanceOfEdits(uint256,uint32) (gas: -3 (-0.000%))
testRemoveMember(uint256) (gas: 2 (0.000%))
testTransfer(uint256,uint256,uint32) (gas: -5 (-0.000%))
testBalanceOfEditsMultiBurnsAndRebase(uint256) (gas: -7 (-0.000%))
testTotalSupplyEditsMultiBurnsAndRebase(uint256) (gas: 7 (0.000%))
testRemoveValidatorsKeyOutOfBounds(bytes32,uint256,uint256) (gas: -169 (-0.000%))
testRemoveValidatorsAndRetrieveAsAdmin(bytes32,uint256,uint256) (gas: -170 (-0.000%))
testRemoveValidatorsFundedKeyRemovalAttempt(bytes32,uint256,uint256) (gas: 215 (0.000%))
testRemoveValidatorsAsOperator(bytes32,uint256,uint256) (gas: -5325 (-0.002%))
testRemoveValidatorsAsAdmin(bytes32,uint256,uint256) (gas: -5351 (-0.002%))
testRemoveValidatorsUnsortedIndexes(bytes32,uint256,uint256) (gas: -20864 (-0.014%))
testTotalSupply(uint256,uint128,uint128) (gas: -6700 (-0.058%))
testBreakingLowerBoundLimit(uint64,uint64,uint32) (gas: -19196 (-0.078%))
testSetBeaconBounds(uint256,uint64) (gas: -4800 (-0.171%))
Overall gas change: -62361 (-0.326%)
```

An example of a different implementation:

```
function removeValidators(uint256 _index, uint256[] calldata _indexes) external operatorOrAdmin(_index)
↪   {
    uint256 indexesLength = _indexes.length;
    if (indexesLength == 0) {
        revert InvalidKeyCount();
    }

    Operators.Operator storage operator = Operators.getByIndex(_index);

    uint256 totalKeys = operator.keys;

    if (!(_indexes[0] < totalKeys)) {
        revert InvalidIndexOutOfBounds();
    }

    uint256 lastIndex = _indexes[_indexes.length - 1];

    if (lastIndex < operator.funded) {
        revert InvalidFundedKeyDeletionAttempt();
    }

    if (lastIndex < operator.limit) {
        operator.limit = lastIndex;
    }

    operator.keys = totalKeys - indexesLength;

    uint256 idx;
    for (; idx < indexesLength;) {
        uint256 keyIndex = _indexes[idx];

        if (idx > 0 && !(keyIndex < _indexes[idx - 1])) {
            revert InvalidUnsortedIndexes();
        }

        unchecked {
            ++idx;
        }

        uint256 lastKeyIndex = totalKeys - idx;

        (bytes memory removedPublicKey,) = ValidatorKeys.get(_index, keyIndex);
        (bytes memory lastPublicKey, bytes memory lastSignature) = ValidatorKeys.get(_index,
↪   lastKeyIndex);

        ValidatorKeys.set(_index, keyIndex, lastPublicKey, lastSignature);
        ValidatorKeys.set(_index, lastKeyIndex, new bytes(0), new bytes(0));

        emit RemovedValidatorKey(_index, removedPublicKey);
    }
}
```

Alluvial: Implemented in SPEARBIT/24.

Spearbit: Acknowledged.

6.5.5 Cache storage related variables in `WLSETH.1.sol::burn` to save gas.

Severity: *Gas Optimization*

Context: WLSETH.1.sol#L140-L151

Description/Recommendation: We can cache `IRiverV1(payable(RiverAddress.get()))` and `BalanceOf.get(msg.sender)` to avoid reading from storage mutiple times.

```
testRemoveValidatorsAsAdmin(bytes32,uint256,uint256) (gas: -2 (-0.000%))
testRemoveValidatorsAsOperator(bytes32,uint256,uint256) (gas: 10 (0.000%))
testRemoveMember(uint256) (gas: 2 (0.000%))
testTransferFrom(uint256,uint256,uint256,uint32) (gas: -580 (-0.002%))
testTransfer(uint256,uint256,uint32) (gas: -611 (-0.002%))
testTotalSupplyEdits(uint256,uint32) (gas: -587 (-0.003%))
testBalanceOfEdits(uint256,uint32) (gas: -590 (-0.003%))
testBurnWrappedTokensWithRebase(uint256,uint32) (gas: -592 (-0.003%))
testBurnWrappedTokens(uint256,uint32) (gas: -587 (-0.003%))
testBurnWrappedTokensInvalidTransfer(uint256,uint32) (gas: -593 (-0.003%))
testBalanceOfEditsMultiBurnsMultiUserAndRebase(uint256,uint256) (gas: -3550 (-0.010%))
testBalanceOfEditsMultiBurnsAndRebase(uint256) (gas: -3557 (-0.014%))
testTotalSupplyEditsMultiBurnsAndRebase(uint256) (gas: -3552 (-0.014%))
testSetBeaconBounds(uint256,uint64) (gas: -4800 (-0.171%))
Overall gas change: -19589 (-0.227%)
```

```
function burn(address _recipient, uint256 _value) external nonReentrant {
    IRiverV1 river = IRiverV1(payable(RiverAddress.get()));
    uint256 balance = BalanceOf.get(msg.sender);

    uint256 callerUnderlyingBalance =
        river.underlyingBalanceFromShares(balance);
    if (_value > callerUnderlyingBalance) {
        revert BalanceTooLow();
    }
    uint256 sharesAmount = river.sharesFromUnderlyingBalance(_value);
    BalanceOf.set(msg.sender, balance - sharesAmount);
    if (!river.transfer(_recipient, sharesAmount)) {
        revert TokenTransferError();
    }
}
```

Alluvial: The final implementation is a bit different but the caching suggestions have been applied in PRs:

- [SPEARBIT/5] Remove dust on WLSETH burn
- [SPEARBIT/33] Documentation and natspec

Spearbit: Acknowledged.

6.5.6 Replace `pad64` with `abi.encodePacked`

Severity: *Gas Optimization*

Context: ConsensusLayerDepositManager.1.sol#L109, ConsensusLayerDepositManager.1.sol#L113

Description/Recommendation: Using the native `abi.encodePacked` just like Ethereum 2.0 deposit contract instead of `pad64` would save a considerable amount of gas.

And since these are the only 2 places that `BytesLib.pad64` has been used, we can remove this helper function from the codebase, unless the team is planning to use it elsewhere.

```
testInitWithZeroAddressValue() (gas: -24260 (-0.009%))
testUserDepositsOperatorWithStoppedValiadtors() (gas: -39038 (-0.018%))
testUserDepositsForAnotherUser() (gas: -44258 (-0.019%))
testDeniedUser() (gas: -44258 (-0.019%))
testELFeeRecipientPullFunds() (gas: -44258 (-0.019%))
testUserDeposits() (gas: -44258 (-0.019%))
testNoELFeeRecipient() (gas: -44258 (-0.019%))
testUserDepositsTenPercentFee() (gas: -44258 (-0.019%))
testUserDepositsFullAllowance() (gas: -44258 (-0.019%))
testUserDepositsUnconventionalDeposits() (gas: -44740 (-0.019%))
testValidatorsPenaltiesEqualToExecLayerFees() (gas: -44258 (-0.020%))
testValidatorsPenalties() (gas: -44258 (-0.020%))
testDepositTwentyValidators() (gas: -12938 (-0.023%))
testDepositTenValidators() (gas: -12938 (-0.023%))
Overall gas change: -532236 (-0.265%)
```

```
bytes32 pubkeyRoot = sha256(abi.encodePacked(_publicKey, bytes16(0)));
bytes32 signatureRoot = sha256(
    abi.encodePacked(
        sha256(BytesLib.slice(_signature, 0, 64)),
        sha256(abi.encodePacked(BytesLib.slice(_signature, 64, SIGNATURE_LENGTH - 64), bytes32(0)))
    )
);
```

Alluvial: Resolved in SPEARBIT/6.

Spearbit: Acknowledged.

6.5.7 Unroll loops with a fixed number of iterations to avoid extra gas costs.

Severity: *Gas Optimization*

Context: Uint256Lib.sol#L8-L13

Description/Recommendation: Since the number of loops is known at compile time, we can unroll it to avoid the extra variable in the stack (`i`) and also avoid the conditional `JUMPI` and `JUMPDEST` opcodes. Here is what the unrolling would look like:

```
result = temp_value & 0xFF;
temp_value >>= 8;

result = (result << 8) | (temp_value & 0xFF);
temp_value >>= 8;

result = (result << 8) | (temp_value & 0xFF);
temp_value >>= 8;

result = (result << 8) | (temp_value & 0xFF);
temp_value >>= 8;

result = (result << 8) | (temp_value & 0xFF);
temp_value >>= 8;

result = (result << 8) | (temp_value & 0xFF);
temp_value >>= 8;

result = (result << 8) | (temp_value & 0xFF);
temp_value >>= 8;

result = (result << 8) | (temp_value & 0xFF);
temp_value >>= 8;
```

and the overall gas saving:

```
testInitWithZeroAddressValue() (gas: 9430 (0.004%))
testUserDepositsOperatorWithStoppedValiadtors() (gas: -14790 (-0.007%))
testUserDepositsForAnotherUser() (gas: -16762 (-0.007%))
testDeniedUser() (gas: -16762 (-0.007%))
testELFeeRecipientPullFunds() (gas: -16762 (-0.007%))
testUserDepositsUnconventionalDeposits() (gas: -16762 (-0.007%))
testUserDeposits() (gas: -16762 (-0.007%))
testNoELFeeRecipient() (gas: -16762 (-0.007%))
testUserDepositsTenPercentFee() (gas: -16762 (-0.007%))
testUserDepositsFullAllowance() (gas: -16762 (-0.007%))
testValidatorsPenaltiesEqualToExecLayerFees() (gas: -16762 (-0.007%))
testValidatorsPenalties() (gas: -16762 (-0.008%))
testDepositTwentyValidators() (gas: -4930 (-0.009%))
testDepositTenValidators() (gas: -4930 (-0.009%))
Overall gas change: -182840 (-0.093%)
```

It would be best to define a constant for `0xFF` as well.

Note the deposit contract uses a similar method as the unrolled loop here.

Alluvial: Recommendation implemented in SPEARBIT/25.

Spearbit: Acknowledged.

6.5.8 Rewrite `pad64` so that it doesn't use `BytesLib.concat` and `BytesLib.slice` to save gas

Severity: *Gas Optimization*

Context: BytesLib.sol#L5-L21

Description: We can avoid using the `BytesLib.concat` and `BytesLib.slice` and write `pad64` mostly in assembly. Since the current implementation adds more memory expansion than needed (also not highly optimized).

Recommendation: Here is an implementation mostly in assembly:

```
function pad64(bytes memory _b) internal pure returns (bytes memory res) {
    assert(_b.length >= 32 && _b.length <= 64);
    if (64 == _b.length) {
        return _b;
    }

    // SPDX-License-Identifier: MIT
    assembly {
        // load free memory pointer
        let freeMemPtr := mload(0x40)

        res := freeMemPtr

        // store length of the new array as 0x40 = 64
        mstore(freeMemPtr, 0x40)

        // advance free memory pointer
        freeMemPtr := add(freeMemPtr, 0x20)

        // store the 1st 32 bytes of `_b` in the next memory slot
        mstore(freeMemPtr, mload(add(_b, 0x20)))

        // advance free memory pointer
        freeMemPtr := add(freeMemPtr, 0x20)

        // _b[32:64)
        let endChunk := mload(add(_b, 0x40))

        // shift value equals the amount needed to pad `_b` to a multiple of 0x20
        let shiftValue := sub(0x40, mload(_b))

        // mask the padded part
        endChunk := shl(shiftValue, shr(shiftValue, endChunk))

        // store padded end of `_b`
        mstore(freeMemPtr, endChunk)

        // advance free memory pointer
        freeMemPtr := add(freeMemPtr, 0x20)

        // update the value at the free memory pointer slot
        mstore(0x40, freeMemPtr)
    }
}
```

And a snapshot of gas savings:

```
testInitWithZeroAddressValue() (gas: -33876 (-0.013%))
testUserDepositsOperatorWithStoppedValiadtors() (gas: -45518 (-0.021%))
testUserDepositsForAnotherUser() (gas: -51602 (-0.022%))
testDeniedUser() (gas: -51602 (-0.022%))
testELFeeRecipientPullFunds() (gas: -51602 (-0.022%))
testUserDeposits() (gas: -51602 (-0.022%))
testNoELFeeRecipient() (gas: -51602 (-0.022%))
testUserDepositsTenPercentFee() (gas: -51602 (-0.022%))
testUserDepositsUnconventionalDeposits() (gas: -52084 (-0.022%))
testUserDepositsFullAllowance() (gas: -51602 (-0.022%))
testValidatorsPenaltiesEqualToExecLayerFees() (gas: -51602 (-0.023%))
testValidatorsPenalties() (gas: -51602 (-0.023%))
testDepositTwentyValidators() (gas: -15178 (-0.027%))
testDepositTenValidators() (gas: -15178 (-0.027%))
Overall gas change: -626252 (-0.311%)
```

The `assert` statement can also be changed to include strict inequalities to further save gas:

```
-assert(_b.length >= 32 && _b.length <= 64)
+assert(_b.length > 31 && _b.length < 65)
```

It would be best to define constants for the literals in the implementation for better readability.

Alluvial: Resolved by SPEARBIT/6 since it has been removed from the code base.

Spearbit: Acknowledged.

6.5.9 Cache `r.value.length` used in a loop condition to avoid reading from the storage multiple times.

Severity: *Gas Optimization*

Context: Operators.sol#L167, Operators.sol#L181, Operators.sol#L207, Operators.sol#L221

Description: In a loop like the one below, consider caching `r.value.length` value to avoid reading from storage on every round of the loop.

```
for (uint256 idx = 0; idx < r.value.length;) {
```

Recommendation: Consider implementing the code below.

```
uint256 idx;
uint256 length = r.value.length;
for (; idx < length;) {
```

Alluvial: Resolved in SPEARBIT/14.

Spearbit: Acknowledged.

6.5.10 Cache the `r.value.length - 1` value to avoid reading from the storage multiple times.

Severity: *Gas Optimization*

Context: Operators.sol#L261-L264

Description/Recommendation: We can cache the `r.value.length - 1` value to avoid reading from the storage twice and also doing the same arithmetic operation twice.

```
testExecutorCanSetOperatorLimit() (gas: -156 (-0.000%))
testGovernorCanSetOperatorLimit() (gas: -156 (-0.000%))
testMakingFunctionGovernorOnly() (gas: -156 (-0.001%))
testRandomCallerCannotSetOperatorLimit() (gas: -156 (-0.001%))
testRandomCallerCannotSetOperatorStatus() (gas: -156 (-0.001%))
testRandomCallerCannotSetOperatorStoppedValidatorCount() (gas: -156 (-0.001%))
testExecutorCanSetOperatorStoppedValidatorCount() (gas: -156 (-0.001%))
testGovernorCanSetOperatorStatus() (gas: -156 (-0.001%))
testGovernorCanSetOperatorStoppedValidatorCount() (gas: -156 (-0.001%))
testGovernorCanAddOperator() (gas: -156 (-0.001%))
testExecutorCanSetOperatorStatus() (gas: -156 (-0.001%))
Overall gas change: -1716 (-0.007%)
```

```
if (!opExists) {
    r.value.push(newValue);
    uint256 index = r.value.length - 1;
    _setOperatorIndex(name, newValue.active, index);
    return index;
```

Alluvial: Issue does not exist anymore given that the referenced code has been removed in SPEARBIT/14.

Spearbit: Acknowledged.

6.5.11 Caching `activeCount` in `Operators.sol::getAllActive/getAllFundable` to storage to save gas

Severity: *Gas Optimization*

Context: Operators.sol#L165-L176, Operators.sol#L205-L216

Description/Recommendation: `activeCount` can be made into a storage variable that gets updated when an operator becomes active or inactive. This way we can avoid the expensive loops(1, 2) to calculate `activeCount` on each call to `getAllActive()` or `getAllFundable()`. Note that for the loop for `getAllFundable()`, we need to store a slightly modified version in the storage and call that `activeFundableCount`.

These extra loops are only needed since we need to plugin the count to define the variable below:

```
// getAllActive()
Operator[] memory activeOperators = new Operator[](activeCount);

// getAllFundable()
CachedOperator[] memory activeOperators = new CachedOperator[](activeCount);
```

Alluvial: Acknowledged.

6.5.12 Rewrite the for loop in `ValidatorKeys.sol::getKeys` to save gas

Severity: *Gas Optimization*

Context: ValidatorKeys.sol#L54-L60

Description: Recommendation: We can rewrite the for loop in `ValidatorKeys.sol::getKeys` to move/cache some variables around to save gas.

```
testUserDepositsOperatorWithStoppedValiadtors() (gas: -3614 (-0.002%))
testUserDepositsForAnotherUser() (gas: -4078 (-0.002%))
testDeniedUser() (gas: -4078 (-0.002%))
testELFeeRecipientPullFunds() (gas: -4078 (-0.002%))
testUserDeposits() (gas: -4078 (-0.002%))
testNoELFeeRecipient() (gas: -4078 (-0.002%))
testUserDepositsTenPercentFee() (gas: -4078 (-0.002%))
testUserDepositsFullAllowance() (gas: -4078 (-0.002%))
testUserDepositsUnconventionalDeposits() (gas: -4145 (-0.002%))
testValidatorsPenaltiesEqualToExecLayerFees() (gas: -4078 (-0.002%))
testValidatorsPenalties() (gas: -4078 (-0.002%))
Overall gas change: -44461 (-0.019%)
```

```
uint256 idx;
for (; idx < amount;) {
    bytes memory rawCredentials = r.value[operatorIndex][idx + startIdx];
    publicKey[idx] = BytesLib.slice(rawCredentials, 0, PUBLIC_KEY_LENGTH);
    signatures[idx] = BytesLib.slice(rawCredentials, PUBLIC_KEY_LENGTH, SIGNATURE_LENGTH);
    unchecked {
        ++idx;
    }
}
```

Alluvial: Resolved in SPEARBIT/24.

Spearbit: Acknowledged.

6.5.13 `Operators.get` in `_getNextValidatorsFromActiveOperators` can be replaced by `Operators.getByIndex` to avoid extra operations/gas.

Severity: *Gas Optimization*

Context: OperatorsRegistry.1.sol#L436

Description: `Operators.get` in `_getNextValidatorsFromActiveOperators` performs multiple checks that have been done before when `Operators.getAllFundable()` was called. This includes finding the `index`, and checking if `OperatorResolution.active` is set. These are all not necessary.

Recommendation: Replace Line 436 in `_getNextValidatorsFromActiveOperators` with:

```
Operators.Operator storage operator = Operators.getByIndex(operators[selectedOperatorIndex].index);
```

```
testRemoveValidatorsAsOperator(bytes32,uint256,uint256) (gas: -2 (-0.000%))
testRemoveValidatorsAsAdmin(bytes32,uint256,uint256) (gas: -7 (-0.000%))
testBalanceOfEditsMultiBurnsMultiUserAndRebase(uint256,uint256) (gas: 2 (0.000%))
testBalanceOfEdits(uint256,uint32) (gas: 2 (0.000%))
testTotalSupplyEdits(uint256,uint32) (gas: 2 (0.000%))
testBurnWrappedTokensInvalidTransfer(uint256,uint32) (gas: 2 (0.000%))
testTotalSupplyEditsMultiBurnsAndRebase(uint256) (gas: -5 (-0.000%))
testBurnWrappedTokens(uint256,uint32) (gas: 8 (0.000%))
testTransferFrom(uint256,uint256,uint256,uint32) (gas: 29 (0.000%))
testTransfer(uint256,uint256,uint32) (gas: -24 (-0.000%))
testGetKeysAsRiver(bytes32,uint256,uint256) (gas: -922 (-0.001%))
testGetKeysAsRiverLimitTest(bytes32,uint256,uint256) (gas: -922 (-0.001%))
testUserDepositsForAnotherUser() (gas: -1872 (-0.001%))
testDeniedUser() (gas: -1872 (-0.001%))
testELFeeRecipientPullFunds() (gas: -1872 (-0.001%))
testUserDeposits() (gas: -1872 (-0.001%))
testNoELFeeRecipient() (gas: -1872 (-0.001%))
testUserDepositsTenPercentFee() (gas: -1872 (-0.001%))
testUserDepositsFullAllowance() (gas: -1872 (-0.001%))
testValidatorsPenaltiesEqualToExecLayerFees() (gas: -1872 (-0.001%))
testValidatorsPenalties() (gas: -1872 (-0.001%))
testRiverFuzzing(uint96,uint96,uint32) (gas: -1872 (-0.001%))
testUserDepositsOperatorWithStoppedValiadtors() (gas: -1872 (-0.001%))
testUserDepositsUnconventionalDeposits() (gas: -2808 (-0.001%))
testBreakingLowerBoundLimit(uint64,uint64,uint32) (gas: -19196 (-0.078%))
Overall gas change: -44433 (-0.090%)
```

Also note, if we apply this change, we can go further and cache `operators[selectedOperatorIndex].index`'s value since it has also been used in the`if/else` block immediately after it.

Alluvial: Recommendation implemented in SPEARBIT/14.

Spearbit: Acknowledged.

6.5.14 Avoid unnecessary equality checks with `true` in `if` statements

Severity: *Gas Optimization*

Context: OperatorsRegistry.1.sol#L197

Description: Statements of the type `if( condition == true)` can be replaced with `if(condition)`. The extra comparison with `true` is redundant.

Recommendation: So Line 197 in OperatorsRegistry can be changed to:

```
if (Operators.exists(_newName)) {
```

Alluvial: We completely removed the name unicity and are now working entirely with indexes as unique identifiers. Names are considered as informational off chain data now. Check has been removed in SPEARBIT/14.

Spearbit: Acknowledged.

6.5.15 Rewrite `OperatorRegistry.getOperatorDetails` to save gas

Severity: *Gas Optimization*

Context: OperatorsRegistry.1.sol#L130

Description: In `getOperatorDetails` the 1st line is:

```
_index = Operators.indexOf(_name);
```

Since we already have the `_index` from this line we can use that along with `getByIndex` to retrieve the `_operatorAddress`. This would reduce the gas cost significantly, since `Operators.get(_name)` calls `Operators._getOperatorIndex(name)` to find the `_index` again.

```
testExecutorCanSetOperatorLimit() (gas: -1086 (-0.001%))
testGovernorCanSetOperatorLimit() (gas: -1086 (-0.001%))
testUserDepositsForAnotherUser() (gas: -2172 (-0.001%))
testDeniedUser() (gas: -2172 (-0.001%))
testELFeeRecipientPullFunds() (gas: -2172 (-0.001%))
testUserDepositsUnconventionalDeposits() (gas: -2172 (-0.001%))
testUserDeposits() (gas: -2172 (-0.001%))
testNoELFeeRecipient() (gas: -2172 (-0.001%))
testUserDepositsTenPercentFee() (gas: -2172 (-0.001%))
testUserDepositsFullAllowance() (gas: -2172 (-0.001%))
testValidatorsPenaltiesEqualToExecLayerFees() (gas: -2172 (-0.001%))
testValidatorsPenalties() (gas: -2172 (-0.001%))
testUserDepositsOperatorWithStoppedValiadtors() (gas: -3258 (-0.002%))
testMakingFunctionGovernorOnly() (gas: -1086 (-0.005%))
testRandomCallerCannotSetOperatorLimit() (gas: -1086 (-0.005%))
testRandomCallerCannotSetOperatorStatus() (gas: -1086 (-0.005%))
testRandomCallerCannotSetOperatorStoppedValidatorCount() (gas: -1086 (-0.005%))
testExecutorCanSetOperatorStoppedValidatorCount() (gas: -1086 (-0.006%))
testGovernorCanSetOperatorStatus() (gas: -1086 (-0.006%))
testGovernorCanSetOperatorStoppedValidatorCount() (gas: -1086 (-0.006%))
testGovernorCanAddOperator() (gas: -1086 (-0.006%))
testExecutorCanSetOperatorStatus() (gas: -1086 (-0.006%))
Overall gas change: -36924 (-0.062%)
```

Also note, when the operator is not `OperatorResolution.active`, `_index` becomes `-1` in both cases. With the change suggested if `_index` is `-1`, `uint256(_index) == type(uint256).max` which would cause `getByIndex` to revert with `OperatorNotFoundAtIndex(index)`. But with the current code, it will revert with an index out-of-bound type of error.

```
_operatorAddress = Operators.getByIndex(uint256(_index)).operator;
```

Recommendation: Consider implementing the following change.

```
- _operatorAddress = Operators.get(_name).operator;
+ _operatorAddress = Operators.getByIndex(uint256(_index)).operator;
```

Alluvial: Issue does not exist anymore given that the referenced code has been removed by PR SPEARBIT/14.

Spearbit: Acknowledged.

6.5.16 Rewrite/simplify `OracleV1.isMember` to save gas.

Severity: *Gas Optimization*

Context: Oracle.1.sol#L189-L200

Description: `OracleV1.isMember` can be simplified to save gas.

Recommendation: Rewrite `isMember`.

```
function isMember(address _memberAddress) external view returns (bool) {
    return OracleMembers.indexOf(_memberAddress) >= 0
}
```

```
testRemoveValidatorsAsOperator(bytes32,uint256,uint256) (gas: -21 (-0.000%))
testRemoveValidatorsAsAdmin(bytes32,uint256,uint256) (gas: -22 (-0.000%))
testBalanceOfEdits(uint256,uint32) (gas: 2 (0.000%))
testBurnWrappedTokensWithRebase(uint256,uint32) (gas: -2 (-0.000%))
testTotalSupplyEdits(uint256,uint32) (gas: 2 (0.000%))
testBurnWrappedTokensInvalidTransfer(uint256,uint32) (gas: -2 (-0.000%))
testBurnWrappedTokens(uint256,uint32) (gas: 3 (0.000%))
testTransfer(uint256,uint256,uint32) (gas: -5 (-0.000%))
testTransferFrom(uint256,uint256,uint256,uint32) (gas: 12 (0.000%))
testBalanceOfEditsMultiBurnsMultiUserAndRebase(uint256,uint256) (gas: 21 (0.000%))
testBalanceOfEditsMultiBurnsAndRebase(uint256) (gas: -24 (-0.000%))
testTotalSupplyEditsMultiBurnsAndRebase(uint256) (gas: -36 (-0.000%))
testRandomCallerCannotRemoveMember() (gas: -121 (-0.001%))
testExecutorCanAddMember() (gas: -121 (-0.002%))
testGovernorCanAddMember() (gas: -121 (-0.002%))
testRemoveMemberUnauthorized(uint256) (gas: -233 (-0.002%))
testAddMember(uint256) (gas: -233 (-0.002%))
testRemoveMember(uint256) (gas: -288 (-0.003%))
testExecutorCanRemoveMember() (gas: -187 (-0.003%))
testGovernorCanRemoveMember() (gas: -187 (-0.003%))
Overall gas change: -1563 (-0.017%)
```

Alluvial: Recommendation implemented in SPEARBIT/20.

Spearbit: Acknowledged.

6.5.17 Cache `beaconSpec.secondsPerSlot * beaconSpec.slotsPerEpoch` multiplication in to save gas.

Severity: *Gas Optimization*

Context: Oracle.1.sol#L167-L168

Description: The calculation for `_startTime` and `_endTime` uses more multiplication than is necessary.

Recommendation: Cache the multiplcation in `beaconSpec.secondsPerSlot * beaconSpec.slotsPerEpoch` to save gas.

```
uint256 secondsPerEpoch = beaconSpec.secondsPerSlot * beaconSpec.slotsPerEpoch;
_startTime = beaconSpec.genesisTime + _startEpochId * secondsPerEpoch;
_endTime = _startTime + secondsPerEpoch * beaconSpec.epochsPerFrame - 1;
```

```
testRemoveValidatorsAsOperator(bytes32,uint256,uint256) (gas: -2 (-0.000%))
testBalanceOfEdits(uint256,uint32) (gas: 2 (0.000%))
testBurnWrappedTokensWithRebase(uint256,uint32) (gas: -2 (-0.000%))
testTotalSupplyEdits(uint256,uint32) (gas: 2 (0.000%))
testBurnWrappedTokensInvalidTransfer(uint256,uint32) (gas: -2 (-0.000%))
testBurnWrappedTokens(uint256,uint32) (gas: 3 (0.000%))
testBalanceOfEditsMultiBurnsAndRebase(uint256) (gas: -5 (-0.000%))
testTransferFrom(uint256,uint256,uint256,uint32) (gas: 10 (0.000%))
testRemoveMember(uint256) (gas: -7 (-0.000%))
testTotalSupplyEditsMultiBurnsAndRebase(uint256) (gas: -17 (-0.000%))
testUserDepositsForAnotherUser() (gas: -166 (-0.000%))
testDeniedUser() (gas: -166 (-0.000%))
testELFeeRecipientPullFunds() (gas: -166 (-0.000%))
testUserDepositsUnconventionalDeposits() (gas: -166 (-0.000%))
testUserDeposits() (gas: -166 (-0.000%))
testNoELFeeRecipient() (gas: -166 (-0.000%))
testUserDepositsTenPercentFee() (gas: -166 (-0.000%))
testUserDepositsFullAllowance() (gas: -166 (-0.000%))
testValidatorsPenaltiesEqualToExecLayerFees() (gas: -166 (-0.000%))
testValidatorsPenalties() (gas: -166 (-0.000%))
testRiverFuzzing(uint96,uint96,uint32) (gas: -166 (-0.000%))
testUserDepositsOperatorWithStoppedValiadtors() (gas: -166 (-0.000%))
testTransfer(uint256,uint256,uint32) (gas: -24 (-0.000%))
Overall gas change: -2034 (-0.001%)
```

Alluvial: Recommendation implemented in SPEARBIT/20.

Spearbit: Acknowledged.

6.5.18 Avoid to waste gas distributing rewards when the number of shares to be distributed is zero

Severity: *Gas Optimization*

Context: River.1.sol#L263-L280

Description: `_onEarnings` calculate and distribute shares to both operators and treasury. During the Audit, the Client stated that both `GlobalFee` and `OperatorRewardsShare`, used to calculate the number of shares to be distributed to operators and treasury, could range from `0` to `BASE`.

This mean that there are scenarios where:

- `sharesToMint` could be `0` (total number of shares to be distributed to operators and treasury).
- `operatorRewards` could be `0` (number of shares to be distributed to operators).
- `sharesToMint - mintedRewards` could be `0` (number of shares to be distributed to treasury).

In those scenarios, all the gas spent in calculation and event emitted by calling `_mintRawShares` could be avoided.

Note: if `GlobalFee` or `OperatorRewardsShare` is `0` the number of shares distributed to the operators as a reward for providing the validators infrastructure would be zero.

Alluvial: The whole operator rewarding system has been removed in SPEARBIT/8. Now shares are distributed to the `treasury` only if `sharesToMint > 0`.

Spearbit: Acknowledged.

6.5.19 `_rewardOperators` could save gas by skipping operators with no active and funded validators

Severity: *Gas Optimization*

Context: River.1.sol#L219-L248

Description: `_rewardOperators` is the River function that distribute the earning rewards to each active operator based on the amount of active validators.

The function iterate over the list of active operators returned by `OperatorsRegistryV1.listActiveOperators` calculating the total amount of active and funded validators (`funded-stopped`) and the number of active and funded validators (`funded-stopped`) for each operator.

Because of current code, the final temporary array `validatorCounts` could have some item that contains `0` if the operator in the index position had no more active validators.

This mean that:

1) gas has been wasted during the loop
2) gas will be wasted in the second loop, distributing **0 shares** to an operator without active and funded validators
3) `_mintRawShares` will be executed without minting any shares but emitting a `Transfer` event

Recommendation: Consider making `listActiveOperators` return only operators that have at least one active and funded validator (`funded-stopped>0`). Consider also preventing to call `_mintRawShares` if the number of shares to be minted is `0`.

Alluvial: The whole operator rewarding system has been removed in SPEARBIT/8.

Spearbit: Acknowledged.

6.6 Informational

6.6.1 Consider adding a strict check to prevent `Oracle` admin to add more than 256 members

Severity: *Informational*

Context: Oracle.1.sol#L202-L211, OracleMembers.sol#L23-L33

Description: At the time of writing this issue in the latest commit at `030b52feb5af2dd2ad23da0d512c5b0e55eb8259`, in the natspec docs of `OracleMembers` there is a `@dev` comment that says

> @dev There can only be up to 256 oracle members. This is due to how report statuses are stored in Reports Positions

If we look at `ReportsPositions.sol` the natspec docs explains that

> Each bit in the stored uint256 value tells if the member at a given index has reported

But both `Oracle.addMember` and `OracleMembers.push` do not prevent the admin to add more than 256 items to the list of oracle members.

If we look at the result of the test (located in *Appendix*), we can see that:

- It's possible to add more than 256 oracle members.
- The result of `oracle.getMemberReportStatus(oracleMember257)` return `true` even if the oracle member has not reported yet.
- Because of that, `oracle.reportConsensusLayerData` (executed by `oracleMember257`) reverts correctly.
- If we remove a member from the list (for example oracle member with index `1`) the `oracleMember257` will be swapped with the removed member and at this point it will be able to vote because `oracle.getMemberReportStatus(oracleMember257)` return `false`.

Recommendation: Consider adding a strict check inside `Oracle.addMember` that prevents the Oracle admin to add more than 256 oracle members.

Alluvial: This issue has been acknowledged as we would never have such a big oracle member count and if we ever need it we would update the Oracle contract to make the report voting scalable above 256.

6.6.2 `ApprovalsPerOwner.set` does not check if `owner` or `spender` is `address(0)`.

Severity: *Informational*

Context: ApprovalsPerOwner.sol#L24-L34

Description: When `ApprovalsPerOwner` value is set for an `owner` and a `spender`, the addresses of the `owner` and the `spender` are not checked against `address(0)`.

Recommendation: Add checks for `owner` and `spender` to make sure they are not passed as the zero address by mistake. Other libraries like OpenZepplin have a similar check.

Alluvial: The following PR has introduced checks for the mentioned addresses that are fed to `ApprovalsPerOwner`. So although the checks are not inside the library, they have been applied before using the library's setter function.

Spearbit: Acknowledged.

6.6.3 Quorum could be higher than the number of oracles, DOSing the Oracle contract

Severity: *Informational*

Context: Oracle.1.sol#L257-L282

Description: The current implementation of `Oracle.setQuorum` only checks if the `_newQuorum` input parameter is not `0` or equal to the current quorum value.

By setting a quorum higher than the number of oracle members, no quorum could be reached for the current or future slots.

Recommendation: Consider adding a check that revert the transaction if the value of `_newQuorum` is greater than the number of oracle members.

Alluvial: Recommendation implemented in PR SPEARBIT/15.

Spearbit: Acknowledged. One important thing to remember is that when the other PR about `removeMember` is merged with this, the part that resets everything about report for the epoch must be done before `_setQuorum` otherwise the function will push a quorum that includes the report from the user that will be deleted.

Alluvial: Acknowledged.

6.6.4 `ConsensusLayerDepositManager.depositToConsensusLayer` should be called only after a quorum has been reached to avoid rewarding validators that have not performed during the frame

Severity: *Informational*

Context: ConsensusLayerDepositManager.1.sol#L50

Description: Alluvial is not tracking timestamps or additional information of some actions that happen on-chain like

- when operator validator is funded on the beacon chain.
- when an operator is added.
- when validators are added or removed.
- when a quorum is reached.
- when rewards/penalties/slashes happen and which validator is involved.
- and so on...

By not having these enriched informations it could happen that validators that have not contributed to a frame will still get rewards and this could be not fair to other validators that have contributed to the overall balance by working and bringing rewards.

Let's make an example: we have 10 operators with 1k validators each at the start of a frame. At some point during the very end of the frame `validato_10` get approved 9k validators and all of them get funded. Those validators only participated a small fraction in the production of the rewards. But because there's no way to track these timing and because oracles do not know anything about these (they just need to report the balance and the number of validators during the frame) they will report and arrive to a quorum of `reportBeacon(correctEpoch, correctAmountOfBalance, 21_000)` that will trigger the `OracleManagerV1.setBeaconData`.

The contract check that `21_000 > DepositedValidatorCount.get()` will pass and `_onEarnings` is called.

Let's not consider the math involved in the process of calculating the number of shares to be distributed based on the staked balance delta, let's say that because of all the increase in capital Alluvial will call `_rewardOperators(1_000_000)`; distributing `1_000_000` shares to operators based on the number of validators that produced that reward.

Because as we said we do not know how much each validator has contributed, those shares will be contributed in the same way to operators that could have not contributed at all to the epoch.

This is true for both scenarios where validators that have joined or exited the beacon chain not at the start of the epoch where the last quorum was set.

Recommendation: If adding these informations and the logic to better track these behaviors on chain is problematic, consider at least documenting all these possible scenarios to let each actor in Alluvial be aware of them.

Alluvial: The whole operator rewarding system has been removed in PR SPEARBIT/8. There's going to be a contract receiving the rewards (treasury) from which we'll pay the operators. To begin with, we'll probably not do a performance-based distribution to keep it simple but this is something we can add in the future as the rewards are computed off-chain.

Spearbit: Acknowledged.

6.6.5 Document the decision to include `executionLayerFees` in the logic to trigger `_onEarnings` to distribute rewards to Operators and Treasury

Severity: *Informational*

Context: OracleManager.1.sol#L40-L68

Description: The `setBeaconData` function from `OracleManager` contract is called when oracle members have reached a quorum. The function after checking that the report data respects some integrity check performs a check to distribute rewards to operators and treasury if needed:

```
        uint256 executionLayerFees = _pullELFees();

        if (previousValidatorBalanceSum < _validatorBalanceSum + executionLayerFees) {
            _onEarnings((_validatorBalanceSum + executionLayerFees) - previousValidatorBalanceSum);
        }
```

The delta between `_validatorBalanceSum` and `previousValidatorBalanceSum` is the sum of all the rewards, penalties and slashes that validators have accumulated during the validation work of one or multiple frames.

By adding `executionLayerFees` to the check, it means that even if the validators have performed poorly (the sum of rewards is less than the sum of penalties+slash) they could still get rewards if `executionLayerFees` is greater than the negative delta of `newSum-prevSum`.

If we look at the natspec of the `_onEarnings` it seems that only the validator's balance (without fees) should be used in the `if` check.

```
    /// @notice Handler called if the delta between the last and new validator balance sum is positive
    /// @dev Must be overriden
    /// @param _profits The positive increase in the validator balance sum (staking rewards)
    function _onEarnings(uint256 _profits) internal virtual;
```

Recommendation: If the current logic is correct, consider updating the natspec comments and further explain the logic and the decisions made to remove any possible doubts.

Alluvial: The whole operator rewarding system has been removed by PR SPEARBIT/8.

Alluvial statement regarding using `executionLayerFees` in the `_onEarnings` trigger check: The whole process has been revamped by adding a maximum amount that we can pull. Basically, the Oracle will compute the maximum allowed increase in balance and give that to the River contract that will be able to pull funds up to the allowed limit. The flow will be documented in the doc PR, but basically we consider el fees as a core revenue stream on which operators and the treasury can take a fee.

Spearbit: Acknowledged.

6.6.6 Consider documenting how and if funds from the execution layer fee recipient are considered inside the `annualAprUpperBound` and `relativeLowerBound` boundaries.

Severity: *Informational*

Context: BeaconReportBounds.sol#L6-L9, Oracle.1.sol#L423-L451, Oracle.1.sol#L468-L475

Description: When oracle members reach a quorum, the `_pushToRiver` function is called. Inside the function, Alluvial is performing some sanity check to prevent malicious oracle member to report malicious beacon data.

```
uint256 prevTotalEth = IRiverV1(payable(address(riverAddress))).totalUnderlyingSupply();
riverAddress.setBeaconData(_validatorCount, _balanceSum, bytes32(_epochId));
uint256 postTotalEth = IRiverV1(payable(address(riverAddress))).totalUnderlyingSupply();
uint256 timeElapsed = (_epochId - LastEpochId.get()) * _beaconSpec.slotsPerEpoch *
↪  _beaconSpec.secondsPerSlot;
_sanityChecks(postTotalEth, prevTotalEth, timeElapsed);
```

```
function _sanityChecks(uint256 _postTotalEth, uint256 _prevTotalEth, uint256 _timeElapsed) internal
↪  view {
  if (_postTotalEth >= _prevTotalEth) {
    uint256 annualAprUpperBound = BeaconReportBounds.get().annualAprUpperBound;
    if (
        uint256(10000 * 365 days) * (_postTotalEth - _prevTotalEth)
            > annualAprUpperBound * _prevTotalEth * _timeElapsed
    ) {
        revert BeaconBalanceIncreaseOutOfBounds(_prevTotalEth, _postTotalEth, _timeElapsed,
↪  annualAprUpperBound);
    }
  } else {
    uint256 relativeLowerBound = BeaconReportBounds.get().relativeLowerBound;
    if (uint256(10000) * (_prevTotalEth - _postTotalEth) > relativeLowerBound * _prevTotalEth) {
        revert BeaconBalanceDecreaseOutOfBounds(_prevTotalEth, _postTotalEth, _timeElapsed,
↪  relativeLowerBound);
    }
  }
}
```

Both `prevTotalEth` and `postTotalEth` call `SharesManager.totalUnderlyingSupply()` that returns the value from `River._assetBalance()`. Inside those balance is also included the amount of fees that are pulled from the `ELFeeRecipient` (Execution Layer Fee Recipient).

Alluvial should document how and if funds from the execution layer fee recipient are also considered inside the `annualAprUpperBound` and `relativeLowerBound` boundaries.

Recommendation: Consider documenting how and if funds from the execution layer fee recipient are considered inside the `annualAprUpperBound` and `relativeLowerBound` boundaries.

Alluvial: Acknowledged.

6.6.7 `Allowlist.allow` allows arbitrary values for `_statuses` input

Severity: *Informational*

Context: Allowlist.1.sol#L46-L70

Description: The current implementation of `allow` does not check if the value inside each `_statuses` item is a valid value or not. The function can be called by both the administrator or the *allower* (roles authorized to manage the user permissions) that can specify arbitrary values to be assigned to the corresponding `_accounts` item.

The user's permissions handled by `Allowlist` are then used by the `River` contract in different parts of the code. Those permissions inside the River contracts are a limited set of permissions that could not match what the *allower*/admin of the `Allowlist` has used to update a user's permission when the `allow` function was called.

Recommendation: Consider documenting which pool of values can be assigned to `_statuses` items based on the checks done by the contracts that use `Allowlist` contract.

Otherwise, consider implementing a way to store "allowed" permission masks to be assigned to `_statuses` and enforce an equality check inside the `allow` loop.

Alluvial: Acknowledged.

6.6.8 Consider exploring a way to update the `withdrawal credentials` and document all the possible scenarios

Severity: *Informational*

Context: Description: The `withdrawal credentials` is currently set when River.initRiverV1 is called. The function will internally call ConsensusLayerDepositManager.initConsensusLayerDepositManagerV1 that will perform `WithdrawalCredentials.set(_withdrawalCredentials);`

After initializing the `withdrawal credentials`, there's no way to update it and change it. The `withdrawal credentials` is a key part of the whole protocol and everything that concern it should be well documented including all the worst-case scenario

- What if the `withdrawal credentials` is lost?
- What if the `withdrawal credentials` is compromised?
- What if the `withdrawal credentials` must be changed (lost, compromised or simply the wrong one has been submitted)? What should be implemented inside the Alluvial logic to use the new `withdrawal credentials` for the operator's validators that have not been funded yet (the old `withdrawal credentials` has not been sent to the Deposit contract)?

Note that currently there's seem to be no way to update the `withdrawal credentials` for a validator already submitted to the Deposit contract.

Recommendation:

- Consider documenting how Alluvial would handle all the scenarios where the `withdrawal credentials` have to be updated.
- Consider documenting how Alluvial plan to safeguard the `withdrawal credentials`.
- Consider implementing a logic that allow Alluvial to update the `withdrawal credentials` on a time basis or after X amount of ETH has been staked to reduce the funds lost in case the key is lost or key/contract is compromised.

Alluvial: Acknowledged.

6.6.9 `Oracle` contract allows members to skip frames and report them (even if they are past) one by one or all at once

Severity: *Informational*

Context: Oracle.1.sol#L284-L334

Description: The current implementation of `reportBeacon` allows oracle members to skip frames (255 epochs) and report them (even if they are past) one by one or all at once.

Let's assume that members arrived to a quorum for `epochId_X`. When quorum is reached, `_pushToRiver` is called, and it will update the following properties:

- clean all the storage used for member reporting.
- set `ExpectedEpochId` to `epochId_X + 255`.
- set `LastEpochId` to `epochId_X`.

With this context, let's assume that members decide to wait 30 frames (30 days) or that for 30 days they cannot arrive at quorum. At the new time, the new epoch would be `epochId_X + 255 * 30`

The following scenarios can happen:

- 1) Report at once all the missed epochs

Instead of reporting only the current epoch (`epochId_X + 255 * 30`), they will report all the previous "skipped" epochs that are in the past.

In this scenario, `ExpectedEpochId` contains the number of the expected next epoch assigned 30 days ago from the previous call to `_pushToRiver`. In `reportBeacon` if the `_epochId` is what the system expect (equal to `ExpectedEpochId`) the report can go on.

So to be able to report all the missing reports of the "skipped" frames the member just need to call in a sequence `reportBeacon(epochId_X + 255, ...)`, `reportBeacon(epochId_X + 255 + 255, ...)` + ... + `reportBeacon(epochId_X + 255 * 30, ...)`

- 2) Report only the last epoch

In this scenario, they would call directly `reportBeacon(epochId_X + 255 * 30, ...)`. `_pushToRiver` call `_sanityChecks` to perform some checks as do not allow changes in the amount of staked ether that are below or above some bounds.

The call that would be made is `_sanityChecks(oracleReportedStakedBalance, prevTotalEth, timeElapsed)` where `timeElapsed` is calculated as `uint256 timeElapsed = (_epochId - LastEpochId.get()) * _beaconSpec.slotsPerEpoch * _beaconSpec.secondsPerSlot;`

So, time elapsed is the number of seconds between the reported epoch and the `LastEpochId`. But in this scenario, `LastEpochId` has the old value from the previous call to `_pushToRiver` made 30 days ago that will be `epochId_X`.

Because of this, the check made inside `_sanityChecks` for the upper bound would be more relaxed, allowing a wider spread between `oracleReportedStakedBalance` and `prevTotalEth`

Recommendation: Consider documenting these possible behaviors or implement a logic that prevents oracle members to call `reportBeacon` for past epochs (relative to the real-time beacon epoch).

Alluvial: Acknowledged.

6.6.10 Consider renaming `OperatorResolution.active` to a more meaningful name

Severity: *Informational*

Context: Operators.sol#L35

Description: The name `active` in the struct `OperatorResolution` could be misleading because it can be confused with the fact that an operator (the struct containing the real operator information is `Operator`) is active or not.

The value of `OperatorResolution.active` does not represent if an operator is active, but is used to know if the `index` associated to the struct's item (`OperatorResolution.index`) is used or not.

Recommendation: Consider renaming to something more meaningful for the usage done in the code and to distinguish it from the operator's active state.

Alluvial: Issue does not exist anymore given that the referenced code has been removed in PR SPEARBIT/14.

Spearbit: Acknowledged.

6.6.11 `lsETH` and `WlsETH`'s `name()` functions return inconsistent name.

Severity: *Informational*

Context: WLSETH.1.sol#L49, SharesManager.1.sol#L53

Description: `lsETH.name()` is `River Ether`, while `WlsETH.name()` is `Wrapped Alluvial Ether`.

Recommendation: Perhaps either `River` or `Alluvial` would need to be replaced by the other.

Alluvial: Recommendation implemented in SPEARBIT/18.

Spearbit: Acknowledged.

6.6.12 Rename modifiers to have consistent naming and patterns `only<ROLE>`.

Severity: *Informational*

Context: Firewall.sol#L42-L48, Firewall.sol#L50-L56

Description: The modifiers `ifGovernor` and `ifGovernorOrExecutor` in `Firewall.sol` have a different naming conventions and also logical patterns.

Recommendation: We can rename and also modify their logic to follow the rest of the similar modifiers in the codebase:

```
modifier onlyGovernor() { // <--- ifGovernor
    if (msg.sender != governor) {
        revert Errors.Unauthorized(msg.sender);
    }
    _;
}

modifier onlyGovernorOrExecutor() { // <--- ifGovernorOrExecutor
    if (msg.sender != governor && msg.sender != executor) {
        revert Errors.Unauthorized(msg.sender);
    }
    _;
}
```

Alluvial: Recommendation implemented in SPEARBIT/30.

Spearbit: Acknowledged.

6.6.13 `OperatorResolution.active` might be a redundant struct field which can be removed.

Severity: *Informational*

Context: Operators.sol#L35

Description: The value of `active` stays `true` once it has been set `true` for a given `index`. This is especially true since the only call to `Operators.set` is from `OperatorsRegistryV1.addOperator` which does not override values for already registered names.

Recommendation: The updates to this value is almost synced with the same value as the updates to `Operator.active` except at `Operators.setOperatorName` (which the unsynched nature at `setOperatorName` might be a mistake). So maybe these 2 parameters are meant to be the same. If so, perhaps we should remove it from either here or from `Operator` struct.

Alluvial: This active value is not named properly, it's just here to signal that the index is used. This struct should be replaced by an `uint256` that stores `index + 1` so we know that if `value > 0` then the name is assigned, and we can return `value - 1` to retrieve the operator index.

Alluvial: Issue does not exist anymore given that the referenced code has been removed in SPEARBIT/14.

Spearbit: Acknowledged.

6.6.14 Inline the known value of the boolean `opExists` with its value.

Severity: *Informational*

Context: Operators.sol#L261-L270

Description/Recommendation: In the `else` branch on Line 269 we know `opExists == true`. So we can simplify the lines 268-269 a bit:

```
if (!newValue.active) {
    _setOperatorIndex(name, false, index);
```

Alluvial: Issue does not exist anymore given that the referenced code has been removed in SPEARBIT/14.

Spearbit: Acknowledged.

6.6.15 The expression for `selectedOperatorAvailableKeys` in `OperatorsRegistry` can be simplified.

Severity: *Informational*

Context: OperatorsRegistry.1.sol#L428-L430

Description: `operators[selectedOperatorIndex].limit` is always less than or equal to `operators[selectedOperatorIndex].keys`. Since the places that the `limit` has been set with a value other than `0` has checks against going above `keys` bound:

OperatorsRegistry.1.sol#L250-L252

```
if (_newLimits[idx] > operator.keys) {
    revert OperatorLimitTooHigh(_newLimits[idx], operator.keys);
}
```

OperatorsRegistry.1.sol#L324-L326

```
if (keyIndex >= operator.keys) {
    revert InvalidIndexOutOfBounds();
}
```

OperatorsRegistry.1.sol#L344-L346

```
if (_indexes[_indexes.length - 1] < operator.limit) {
    operator.limit = _indexes[_indexes.length - 1];
}
```

Recommendation: The usage of the`Uint256Lib.min` utility function is not necessary and the value of `selectedOperatorAvailableKeys` can be computed as:

```
uint256 selectedOperatorAvailableKeys = (
        operators[selectedOperatorIndex].limit -
        operators[selectedOperatorIndex].funded
);
```

Spearbit: It is kind of implemented. Although the minimizer/cost function has been modified in SPEARBIT/3.

6.6.16 The unused constant `DELTA_BASE` can be removed

Severity: *Informational*

Context: BeaconReportBounds.sol#L11

Description: The constant `DELTA_BASE` in `BeaconReportBounds` is never used.

Recommendation: If you are not planning to use it, it would be best to remove it from the codebase.

Alluvial: Recommendation implemented in SPEARBIT/30. Note: We changed the name of `BeaconReportBounds` to `ReportBounds`.

Spearbit: Acknowledged.

6.6.17 Remove unused modifiers

Severity: *Informational*

Context: OperatorsRegistry.1.sol#L115

Description: The `modifier active(uint256 _index)` is not used in the project.

Recommendation: If you are not planning to use this modifier, consider removing it from the codebase.

Alluvial: Recommendation implemented in SPEARBIT/30.

Spearbit: Acknowledged.

6.6.18 Modifier names do not follow the same naming patterns

Severity: *Informational*

Context: OperatorsRegistry.1.sol#L45, OperatorsRegistry.1.sol#L62

Description: The modifier names do not follow the same naming patterns in `OperatorsRegistry`.

Recommendation: Consider renaming the following modifiers:

```
modifier operatorFeeRecipientOrAdmin(uint256 _index)
modifier operatorOrAdmin(uint256 _index)
```

To be consistent with other `modifier` names, it might be best to call `operatorFeeRecipientOrAdmin` `onlyActiveOperatorFeeRecipientOrAdmin` (and its input could be called `_operatorIndex`). And also we can rename `operatorOrAdmin` to `onlyActiveOperatorOrAdmin` (and its input could be called `_operatorIndex`).

Alluvial: Recommendation implemented in SPEARBIT/30.

Spearbit: Acknowledged.

6.6.19 In `AllowlistV1.allow` the input variable `_statuses` can be renamed to better represent that values it holds

Severity: *Informational*

Context: Allowlist.1.sol#L49

Description: In `AllowlistV1.allow` the input variable `_statuses` can be renamed to better represent the values it holds. `_statuses` is a bitmap where each bit represents a particular action that a user can take.

Recommendation: Rename `_statuses` to better represent what it is used for. Basically, each `status` here is representing 256 permission roles. Maybe rename it to `_userPermissions` to relate to the use cases in the other functions in `AllowlistV1`.

Alluvial: Changed `_statuses` to `_permissions` in SPEARBIT/30.

Spearbit: Acknowledged.

6.6.20 `riverAddress` can be renamed to `river` and we can avoid extra interface casting

Severity: *Informational*

Context: Oracle.1.sol#L468-L471, Oracle.1.sol#L479

Description: `riverAddress`'s name suggest that it is only an address. Although it is an address with the `IRiverV1` attached to it. Also, we can avoid unnecessary casting of interfaces.

Recommendation: `riverAddress` can be renamed to `river`, since it's not just an address as the name suggest. Also, the extra interface castings can be removed.

```
IRiverV1 river = IRiverV1(payable(RiverAddress.get()));
uint256 prevTotalEth = river.totalUnderlyingSupply();
river.setBeaconData(_validatorCount, _balanceSum, bytes32(_epochId));
uint256 postTotalEth = river.totalUnderlyingSupply();
...
emit PostTotalShares(
   postTotalEth, prevTotalEth, timeElapsed, river.totalSupply()
);
```

Alluvial: Recommendation implemented in SPEARBIT/30.

Spearbit: Acknowledged.

6.6.21 Define named constants for numeric literals

Severity: *Informational*

Context: Oracle.1.sol#L436, Oracle.1.sol#L447

Description: In `_sanitychecks` there 2 numeric literals `10000` and `365 days` used:

```
uint256(10000 * 365 days) * (_postTotalEth - _prevTotalEth)
...
if (uint256(10000) * (_prevTotalEth - _postTotalEth) > relativeLowerBound * _prevTotalEth) {
```

Recommendation: It would be best to define and replace those with named constants.

Alluvial: Recommendation implemented in SPEARBIT/19.

Spearbit: Acknowledged.

6.6.22 Move `memberIndex` and `ReportsPositions` checks at the beginning of the `OracleV1.reportBeacon` function.

Severity: *Informational*

Context: Oracle.1.sol#L289, Oracle.1.sol#L307-L313

Description: The checks for `memberIndex == -1` and `ReportsPositions.get(uint256(memberIndex))` happen in the middle of `reportBeacon` after quite a few calculations are done.

Recommendation: Move these checks to the beginning of the function to `revert` early if it would ever have to `revert` because of these 2 checks.

```
function reportBeacon(uint256 _epochId, uint64 _beaconBalance, uint32 _beaconValidators) external {
   int256 memberIndex = OracleMembers.indexOf(msg.sender);
   if (memberIndex == -1) {
       revert Errors.Unauthorized(msg.sender);
   }
   ...
   if (ReportsPositions.get(uint256(memberIndex))) {
       revert AlreadyReported(_epochId, msg.sender);
  }
  ...
```

Alluvial: Recommendation implemented in SPEARBIT/20.

Spearbit: Acknowledged.

6.6.23 Document what incentivizes the operators to run their validators when `globalFee` is zero

Severity: *Informational*

Context: River.1.sol#L270

Description: If `GlobalFee` could be `0`, then neither the treasury nor the operators earn rewards. What factor would motivate the operators to keep their validators running?

Recommendation: It would be best to document if there are other incentives for the operators to keep their validators running when `globalFee` is zero.

Alluvial: Acknowledged.

6.6.24 Document how Alluvial plans to prevent institutional investors and operators get into business directly and bypass using the `River` protocol.

Severity: *Informational*

Description: Since the list of operators and also depositors can be looked up from the information on-chain, what would prevent Institutional investors (users) and the operators to do business outside of `River`? Is there going to be an off-chain legal contract between Alluvial and these other entities to prevent this scenario?

Recommendation: Document how Alluvial plans to prevent institutional investors and operators get into business directly and bypass using the `River` protocol.

Alluvial: Acknowledged.

6.6.25 Document how operator rewards will be distributed if `OperatorRewardsShare` is zero

Severity: *Informational*

Context: River.1.sol#L275

Description: If `OperatorRewardsShare` could be `0`, then the operators won't earn rewards. What factor would motivate the operators to keep their validators running?

Sidenote: Other incentives for the operators to keep their validators running (if their reward share portion is `0`) would be some sort of MEV or block proposal/attestation bribes.

Related: *Avoid to waste gas distributing rewards when the number of shares to be distributed is zero*

Recommendation: It would be best to document this issue for the operators so they would know how they can access the rewards.

Alluvial: The whole operator rewarding system has been removed in SPEARBIT/8. The revenue redistribution would be computed off-chain and will be detailed in the documentation PR.

Spearbit: Acknowledged.

6.6.26 Current operator reward distribution does not favor more performant operators

Severity: *Informational*

Context: River.1.sol#L238

Description: Reward shares are distributed based on the fraction of the active funded non-stopped validators owned by an operator. This distribution of shares does not promote the honest operation of validators to the fullest extent.

Since the oracle members don't report the delta in the balance of each validator, it is not possible to reward operators/validators that have been performing better than the rest.

Also if a high-performing operator or operators were the main source of the beacon balance sum and if they had enough `ETH` to initially deposit into the `ETH2.0` deposit contract on their own, they could have made more profit that way versus joining as an operator in the `River` protocol.

Recommendation: Oracle members can report more information per voting frame. For example, they can report the delta of balances during a voting frame per all validators of an operator. Then `River` can incorporate these specialized deltas in the reward distribution share minting process.

Alluvial: Tracking the delta in performance on validators is too complex to achieve on-chain + the revenue delta between excellent and good operators is not massive. So the hassle to track all that on-chain is too big and it could be resolved by off-chain monitoring and discussions with operators.

Also I don't get the point of operators that could directly deposit because they're not the ones bringing the funds in the system, they are simply paid to run validators on behalf of the system, and the goal of the system is to transparently expose the list of operators and their validators so anyone can audit the performance or the oracle reports.

Spearbit: It's kind of a cost/reward analysis that depends on the overall earnings, `GlobalFee`, and `OperatorRewardsShare`. The operators who run the validators which may not have the funds to stake have at least 2 options:

- Join `River` protocol so others would pay for staking `32 ETH` and earn rewards based on the parameters above.
- Take loans (with some interest rates I assume) to fully or partially fund their validators and earn interests and eventually payout those loans.
- Join other liquid staking protocols with a different set of parameters.

Has Alluvial analyzed/compared scenarios like the above to come up with parameters that would attract potential operators into joining their system?

Alluvial: The whole operator rewarding system has been removed in SPEARBIT/8.

6.6.27 `TRANSFER_MASK == 0` which causes a no-op.

Severity: *Informational*

Context: River.1.sol#L36-L37, River.1.sol#L189-L190, River.1.sol#L198-L204, Allowlist.1.sol#L82, Allowlist.1.sol#L96, Allowlist.1.sol#L110

Description: `TRANSFER_MASK` is a named constant defined as `0` (River.1.sol#L37). Like the other masks `DEPOSIT_MASK` and `DENY_MASK` which supposed to represent a bitmask, on the first look, you would think `TRANSFER_MASK` would need to also represent a bitmask. But if you take a look at `_onTransfer`:

```
function _onTransfer(address _from, address _to) internal view override {
    IAllowlistV1(AllowlistAddress.get()).onlyAllowed(_from, TRANSFER_MASK); // this call reverts if
↪  unauthorized or denied
    IAllowlistV1(AllowlistAddress.get()).onlyAllowed(_to, TRANSFER_MASK); // this call reverts if
↪  unauthorized or denied
}
```

This would translate into calling `onlyAllowed` with the:

```
IAllowlistV1(AllowlistAddress.get()).onlyAllowed(x, 0);
```

Now if we look at the `onlyAllowed` function with these parameters:

```
function onlyAllowed(x, 0) external view {
    uint256 userPermissions = Allowlist.get(x);
    if (userPermissions & DENY_MASK == DENY_MASK) {
        revert Denied(_account);
    }
    if (userPermissions & 0 != 0) { // <---  ( x & 0 != 0 ) == false
        revert Unauthorized(_account);
    }
}
```

Thus if the `_from`, `_to` addresses don't have their `DENY_MASK` set to `1` they would not trigger a `revert` since we would never step into the 2nd `if` block above when `TRANSFER_MASK` is passed to these functions.

The `TRANSFER_MASK` is also used in `_onDeposit`:

```
IAllowlistV1(AllowlistAddress.get()).onlyAllowed(_depositor, DEPOSIT_MASK + TRANSFER_MASK); //
↪  DEPOSIT_MASK + TRANSFER_MASK == DEPOSIT_MASK
IAllowlistV1(AllowlistAddress.get()).onlyAllowed(_recipient, TRANSFER_MASK); // like above in
↪  `_onTransfer`
```

Recommendation: At first, we thought `TRANSFER_MASK` was not assigned the correct bitmask value. So we recommended setting those values correctly. But after discussing with the Alluvial team, it became clear that they are not planning to use this parameter currently and that is why it is assigned a `no-op` type of value.

Alluvial: They have been asked to leave it so they can change its value if they ever need that feature (imo won't ever happen else the project becomes unusable)

Spearbit: If `TRANSFER_MASK` is not being used, it would be best to remove it from the project including all the code that relies on it. Right now, the only thing that it does is a `no-op`. Also, if its value ever needed to be changed the whole contract would need to be redeployed (since it's a constant). So in this case, I would change the above lines in `_onTransfer` to:

```
IAllowlistV1 allowList = IAllowlistV1(AllowlistAddress.get());

if( allowList.isDenied(_from) ) {
   // Denied custom error needs to be imported or created or
   // we can create a new function for IAllowlist called `onlyNotDenied` to
   // replace this whole `if` block
   revert Denied(_from);
}

if( allowList.isDenied(_to) ) {
   revert Denied(_to);
}
```

and in `_onDeposit` to:

```
IAllowlistV1 allowList = IAllowlistV1(AllowlistAddress.get());

allowList.onlyAllowed(_depositor, DEPOSIT_MASK);

if( allowList.isDenied(_recipient) ) {
   revert Denied(_recipient);
}
```

Alluvial: Recommendation implemented in SPEARBIT/21.

Spearbit: Acknowledged.

6.6.28 Reformat numeric literals with many digits for better readability.

Severity: *Informational*

Context: River.1.sol#L35, Oracle.1.sol#L436, Oracle.1.sol#L447, ConsensusLayerDepositManager.1.sol#L107

Description: Reformat numeric literals with many digits into a more readable form.

Recommendation: Use _ or scientific notation to either partition the digits or shorten the form.

River.1.sol#L35

```
- uint256 public constant BASE = 100000;
+ uint256 public constant BASE = 100_000; // or 1e5
```

Oracle.1.sol#L436, Oracle.1.sol#L447

```
- 10000
+ 100_00 // or 1e4
```

ConsensusLayerDepositManager.1.sol#L107

```
- uint256 depositAmount = value / 1000000000 wei;
+ uint256 depositAmount = value / 1 gwei; // or 1e9 wei or 1_000_000_000 wei
```

Alluvial: Recommendation implemented in SPEARBIT/19.

Spearbit: Acknowledged.

6.6.29 `Firewall` should follow the two-step approach present in `River` when transferring `govern` address

Severity: *Informational*

Context: Firewall.sol#L59-L61

Description: Both `River` and `OperatorsRegistry` follow a two-step approach to transfer the ownership of the contract.

1) Propose a new owner storing the `address` in a `pendingAdmin` variable
2) The pending admins accept the new role by actively calling `acceptOwnership`

This approach makes this crucial action much safer because

1) Prevent the admin to transfer ownership to `address(0)` given that `address(0)` cannot call `acceptOwnership`
2) Prevent the admin to transfer ownership to an address that cannot "admin" the contract if they cannot call `acceptOwnership`. For example, a contract do not have the implementation to at least call `acceptOwnership`.
3) Allow the current admin to stop the process by calling `transferOwnership(address(0))` if the pending admin has not called `acceptOwnership` yet

The current implementation does not follow this safe approach, allowing the governor to directly transfer the governor role to a new address.

Recommendation: Consider implementing a two-step approach transfer of the governor role, similarly implemented in the `River` and `OperatorsRegistry` contracts.

Alluvial: Recommendation implemented in SPEARBIT/11.

Spearbit: Note(1); Still missing (client said it will be implemented in other PRs)

- `Administrable` miss all natspec comments.
- Event for `_setAdmin` are still missing but will be added to `Initializable` event in another PR.

Note(2); Client has acknowledged that all the contracts that inherit from `Administrable` have the ability to transfer ownership, even contracts like `AllowlistV1` that didn't have the ability before this PR.

Alluvial: Issues in Note 1 addressed in SPEARBIT/33.

Spearbit: Acknowledged.

6.6.30 `OperatorRegistry.removeValidators` is resetting the `limit` (approved validators) even when not needed

Severity: *Informational*

Context: OperatorsRegistry.1.sol#L304-L347

Description: The current implementation of `removeValidators` allow an admin or node operator to remove validators, passing to the function the list of validator's index to be removed.

Note that the list of indexes must be ordered DESC.

At the end of the function, we can see these checks

```
if (_indexes[_indexes.length - 1] < operator.limit) {
    operator.limit = _indexes[_indexes.length - 1];
}
```

That reset the operator's `limit` to the lower index value (this to prevent that a not approved key get swapped to a position inside the `limit`). The issue with this implementation is that it is not considering the case where all the operator's validators are already approved by Alluvial. In this case, if an operator removes the validator with the lower index, all the other validators get de-approved because the limit will be set to the lower limit.

Consider this scenario:

```
op.limit = 10
op.keys = 10
op.funded = 0
```

This mean that all the validators added by the operator have been approved by Alluvial and are safe (`keys == limit`).

If the operator or Alluvial call `removeValidators([validatorIndex], [0])` removing the validator at index `0` this will

- swap the `validator_10` with `validator_0`.
- set the limit to `0` because `0 < 10` (`_indexes[_indexes.length - 1] < operator.limit`).

The consequence is that even if all the validators present before calling `removeValidators` were "safe" (because approved by Alluvial) the limit is now `0` meaning that all the validators are not "safe" anymore and cannot be selected by `pickNextValidators`.

Recommendation: Consider updating the logic of `removeValidators` to handle this edge case.

Alluvial: Recommendation implemented in SPEARBIT/22.

Spearbit: Acknowledged.

6.6.31 Consider renaming `transferOwnership` to better reflect the function's logic

Severity: *Informational*

Context: River.1.sol#L142-L146, OperatorsRegistry.1.sol#L88-L92

Description: The current implementation of `transferOwnership` is not really transferring the ownership from the current admin to the new one. The function is setting the value of the **Pending Admin** that must subsequently call `acceptOwnership` to accept the role and confirm the transfer of the ownership.

Recommendation: Consider renaming the `transferOwnership` function name to something that better reflects the function logic.

Alluvial: Recommendation implemented in SPEARBIT/11.

Spearbit: Acknowledged.

6.6.32 Wrong return name used

Severity: *Informational*

Context: Uint256Lib.sol#L20

Description: The `min` function returns the minimum of the 2 inputs, but the return name used is `max`.

Recommendation: Rename or remove return variable name.

```
-function min(uint256 a, uint256 b) internal pure returns (uint256 max) {
+function min(uint256 a, uint256 b) internal pure returns (uint256) {
```

Alluvial: Recommendation implemented in SPEARBIT/30.

Spearbit: Acknowledged.

6.6.33 Discrepancy between architecture and code

Severity: *Informational*

Context: ConsensusLayerDepositManager.1.sol#L50

Description: The architecture diagram states that admin triggers deposits on the Consensus Layer Deposit Manager, but the `depositToConsensusLayer()` function allows anyone to trigger such deposits.

Recommendation: Implement necessary changes to reflect the intended functionality.

Alluvial: `depositToConsensusLayer()` function is now protected by `onlyAdmin-CDMV1` modifier in SPEARBIT/27.

Spearbit: Acknowledged.

6.6.34 Consider replacing the remaining `require` with custom errors

Severity: *Informational*

Context: ConsensusLayerDepositManager.1.sol#L129, BytesLib.sol#L94, BytesLib.sol#L95

Description: In the vast majority of the project contracts have defined and already use Custom Errors that provide a better UX, DX and gas saving compared to `require` statements.

There are still some instances of `require` usage in `ConsensusLayerDepositManager` and `BytesLib` contracts that could be replaced with custom errors.

Recommendation: Consider replacing the remaining `require` statements with custom errors to follow the best practice already adopted by the project.

Alluvial: Recommendation implemented in SPEARBIT/23.

Spearbit: Acknowledged.

6.6.35 Both `wlsETH` and `lsETH` `transferFrom` implementation allow the owner of the token to use `transferFrom` like if it was a "normal" transfer

Severity: *Informational*

Context: WLSETH.1.sol#L103-L109, SharesManager.1.sol#L129-L135

Description: The current implementation of `transferFrom` allow the `msg.sender` to use the function like if it was a "normal" `transfer`. In this case, the allowance is checked only if the `msg.sender` is not equal to `_from`

```
if (_from != msg.sender) {
    uint256 currentAllowance = ApprovalsPerOwner.get(_from, msg.sender);
    if (currentAllowance < _value) {
        revert AllowanceTooLow(_from, msg.sender, currentAllowance, _value);
    }
    ApprovalsPerOwner.set(_from, msg.sender, currentAllowance - _value);
}
```

This implementation diverge from what is usually implemented in both Solmate and OpenZeppelin.

Recommendation: For clarity, remove the check and allow only the `msg.sender` to transfer tokens from his/her balance only via the `transfer` function. Removing the check would also make the function cost less gas.

Alluvial: Recommendation implemented in SPEARBIT/9.

Spearbit: Acknowledged.

6.6.36 Both `wlsETH` and `lsETH` tokens are reducing the allowance when the allowed amount is `type(uint256).max`

Severity: *Informational*

Context: WLSETH.1.sol#L103-L109, SharesManager.1.sol#L129-L135

Description: The current implementation of the function `transferFrom` in both `SharesManager.1.sol` and `WLSETH.1.sol` is not taking into consideration the scenario where a user has approved a spender the maximum possible allowance `type(uint256).max`.

The Alluvial `transferFrom` acts differently from standard ERC20 implementations like the one from Solmate and OpenZeppelin. In their implementation, they check and reduce the spender allowance if and only if the allowance is different from `type(uint256).max`.

Recommendation: Consider following the standard ERC20 implementation from Solmate or OpenZeppelin to prevent reducing an allowance that has set to infinite (`type(uint256).max`).

Alluvial: Recommendation implemented in SPEARBIT/9.

Spearbit: Acknowledged.

6.6.37 Missing, confusing or wrong natspec comments

Severity: *Informational*

Context:

- Allowlist.1.sol
- Firewall.sol
- Initializable.sol
- OperatorsRegistry.1.sol
- Oracle.1.sol
- River.1.sol
- TUPProxy.sol
- WLSETH.1.sol
- Withdraw.1.sol
- ConsensusLayerDepositManager.1.sol
- OracleManager.1.sol
- SharesManager.1.sol
- UserDepositManager.1.sol
- all interfaces inside the `interfaces` folder consider the usage of `@inheritdoc`
- BytesLib.sol
- Errors.sol
- LibOwnable.sol
- Uint256Lib.sol
- UnstructuredStorage.sol
- all the contracts/libraries inside the `state` folder

Description: In the current implementation not all the constructors, functions, events, custom errors, variables or struct are covered by natspec comments. Some of them are only partially covered (missing `@param`, `@return` and so on).

Note that the contracts listed in the context section of the issue have inside of them complete or partial missing natspec.

- Natspec Fixes / Typos:

River.1.sol#L38-L39

Swap the empty line with the NatSpec `@notice`

```
- /// @notice Prevents unauthorized calls
-
+
+ /// @notice Prevents unauthorized calls
```

OperatorsRegistry.1.sol#L44, OperatorsRegistry.1.sol#L61, OperatorsRegistry.1.sol#L114

Replace `name` with `index`.

```
- /// @param _index The name identifying the operator
+ /// @param _index The index identifying the operator
```

OperatorsRegistry.1.sol#L218

Replace `cound` with `count`.

```
- /// @notice Changes the operator stopped validator cound
+ /// @notice Changes the operator stopped validator count
```

- Expand the natspec explanation:

We also suggest expanding some function's logic inside the natspec

OperatorsRegistry.1.sol#L355-L358

Expand the natspec documentation and add a `@return` natspec comment clarifying that the returned value is the number of total operator and not the active/fundable one.

ReportsVariants.sol#L5

Add a comment that explains the `COUNT_OUTMASK`'s assignment. This will mask `beaconValidators` and `beaconBalance` in the designed packing.

```
xx...xx <beaconBalance> <beaconValidators> xxxx & COUNT_OUTMASK  ==
00...00 <beaconBalance> <beaconValidators> 0000
```

ReportsVariants.sol

`ReportsVariants` should have a documentation regarding the packing used for `ReportsVariants` in an `uint256`:

```
[ 0,  16) : <voteCount>          oracle member's total vote count for the numbers below (uint16, 2
↪  bytes)
[16,  48) : <beaconValidators>   total number of beacon validators (uint32, 4 bytes)
[48, 112) : <beaconBalance>      total balance of all the beacon validators (uint64, 6 bytes)
```

OracleMembers.sol

Leave a comment/warning that only there could a maximum of `256` oracle members. This is due to the `ReportsPosition` setup where in an `uint256`, 1 bit is reserved for each oracle member's index.

ReportsPositions.sol

Leave a comment/warning for the `ReportsPosition` setup that the `i`th bit in the `uint256` represents whether or not there has been a beacon report by the `i`th oracle member.

Oracle.1.sol#L202-L205

Leave a comment/warning that only there could a maximum of `256` oracle members. This is due to the `ReportsPosition` setup where in an `uint256`, 1 bit is reserved for each oracle member's index.

Allowlist.1.sol#L46-L49

Leave a comment, warning that the permission bitmaps will be overwritten instead of them getting updated.

OracleManager.1.sol#L44

Add more comment for `_roundId` to mention that when the `setBeaconData` is called by `Oracle.1.sol:_pushToRiver` and that the value passed to it for this parameter is always the **1st epoch of a frame**.

OperatorsRegistry.1.sol#L304-L310

`_indexes` parameter, mentioning that this array:

1) needs to be duplicate-free and sorted (`DESC`)
2) each element in the array needs to be in a specific range, namely `operator.[funded, keys)`.

OperatorsRegistry.1.sol#L60-L62

Better rephrase the natspec comment to avoid further confusion.

Oracle.1.sol#L284-L289

Update the `reportBeacon` natspec documentation about the `_beaconValidators` parameter to avoid further confusion.

Client answer to the PR comment

> The docs should be updated to also reflect our plans for the Shanghai fork. Basically we can't just have the same behavior for a negative delta in validator count than with a positive delta (where we just assume that each validator that was in the queue only had 32 eth). Now when we exit validator we need to know how much was exited in order to compute the proper revenue value for the treasury and operator fee. This probably means that there will be an extra arg with the oracle to keep track of the exited eth value. But as long as the spec is not final, we'll stick to the validator count always growing. We should definitely add a custom error to explain that in case a report provides a smaller validator count.

Recommendation: Consider the missing natspec to the suggested constructors, functions, events, custom errors, variables or struct.

Alluvial: Addressed in [SPEARBIT/33] Documentation and natspec.

Spearbit: Acknowledged.

6.6.38 Remove unused imports from code

Severity: *Informational*

Context: ELFeeRecipient.1.sol#L6, Oracle.1.sol#L9

Description: The codebase has unused imports across the code base. If they are not used inside the contract, it would be better to remove them to avoid confusion.

Recommendation: Remove imports if not used by the contracts.

Alluvial: Recommendation implemented in SPEARBIT/32.

Spearbit: Acknowledged.

6.6.39 Missing event emission in critical functions, init functions and setters

Severity: *Informational*

Context:

- Allowlist.1.sol#L22
- Allowlist.1.sol#L37
- OperatorsRegistry.1.sol#L23
- OperatorsRegistry.1.sol#L84
- OperatorsRegistry.1.sol#L90
- OperatorsRegistry.1.sol#L95
- ELFeeRecipient.1.sol#L18
- Firewall.sol#L25-L30
- Firewall.sol#L59
- Firewall.sol#L64
- Firewall.sol#L69
- Oracle.1.sol#L53-L62
- Oracle.1.sol#L205
- Oracle.1.sol#L216
- Oracle.1.sol#L230
- Oracle.1.sol#L248
- River.1.sol#L57-L68
- River.1.sol#L92
- River.1.sol#L107
- River.1.sol#L122
- River.1.sol#L133
- River.1.sol#L144
- River.1.sol#L149
- River.1.sol#L169
- TUPProxy.sol#L27
- TUPProxy.sol#L32
- WLSETH.1.sol#L43
- WLSETH.1.sol#L129
- WLSETH.1.sol#L140
- OracleManager.1.sol#L77

Description: Some critical functions like contract's `constructor`, contract's `init*(...)`function (upgradable contracts) and some setter or in general critical functions are missing event emission.

Event emissions are very useful for external web3 applications, but also for monitoring the usage and security of your protocol when paired with external monitoring tools.

Note: in the `init*(...)`/`constructor` function, consider if adding a general broad event like `ContractInitialized` or split it in more specific events like `QuorumUpdated+OwnerChanged+...`

Note: in general, consider adding an event emission to all the `init*(...)` functions used to initialize the upgradable contracts, passing to the event the relevant args in addition to the version of the upgrade.

Recommendation: Consider adding event emission to the contracts that are lacking them.

Alluvial: Recommendation implemented in SPEARBIT/31.

Spearbit: Acknowledged.

7 Appendix

7.1 Test Cases

7.1.1 `Oracle.removeMember` could, in the same epoch, allow members to vote multiple times and other members to not vote at all test

to run it `forge test --match-contract SpearReportBeaconTest -vv`

```
//SPDX-License-Identifier: MIT

pragma solidity 0.8.10;

import "forge-std/Test.sol";
import "../src/Oracle.1.sol";
import "../src/libraries/Errors.sol";
import "../src/Withdraw.1.sol";
import "./utils/River.setup1.sol";
import "./mocks/RiverMock.sol";
import "../src/interfaces/IRiver.1.sol";

contract SpearReportBeaconTest is Test {
    OracleV1 internal oracle;

    IRiverV1 internal oracleInput;

    address internal admin = address(0xEA674fdDe714fd979de3EdF0F56AA9716B898ec8);

    address internal oracleOne = address(0x7fe52bbF4D779cA115231b604637d5f80bab2C40);
    address internal oracleTwo = address(0xb479DE67E0827Cc72bf5c1727e3bf6fe15007554);

    uint64 internal constant EPOCHS_PER_FRAME = 225;
    uint64 internal constant SLOTS_PER_EPOCH = 32;
    uint64 internal constant SECONDS_PER_SLOT = 12;
    uint64 internal constant GENESIS_TIME = 1606824023;

    uint256 internal constant UPPER_BOUND = 1000;
    uint256 internal constant LOWER_BOUND = 500;

    function setUp() public {
        oracleInput = IRiverV1(payable(address(new RiverMock())));
        oracle = new OracleV1();
        oracle.initOracleV1(
            address(oracleInput),
            admin,
            EPOCHS_PER_FRAME,
            SLOTS_PER_EPOCH,
            SECONDS_PER_SLOT,
            GENESIS_TIME,
            UPPER_BOUND,
            LOWER_BOUND
        );

        vm.startPrank(admin);
        oracle.setQuorum(2);
        vm.stopPrank();
    }

    function testRemoveMemberExploit() public {
        uint64 timeFromGenesis = 1;
        uint64 balanceSum = 1;
        uint32 validatorCount = 1;
```

```
        RiverMock(address(oracleInput)).sudoSetTotalShares(1e9 * uint256(balanceSum));
        RiverMock(address(oracleInput)).sudoSetTotalSupply(1e9 * uint256(balanceSum));

        vm.warp(uint256(GENESIS_TIME) + uint256(timeFromGenesis));
        vm.startPrank(admin);
        oracle.setQuorum(100);
        // add both members
        oracle.addMember(oracleOne);
        oracle.addMember(oracleTwo);
        vm.stopPrank();

        uint256 epochId = oracle.getCurrentEpochId();
        uint256 frameFirstEpochId = oracle.getFrameFirstEpochId(epochId);

        // Register oracle one report
        vm.prank(oracleOne);
        oracle.reportBeacon(frameFirstEpochId, balanceSum, validatorCount);

        assertEq(oracle.getMemberReportStatus(oracleOne), true);
        assertEq(oracle.getMemberReportStatus(oracleTwo), false);

        // Try to register again with oracle 1, it should revert
        vm.prank(oracleOne);
        vm.expectRevert(abi.encodeWithSignature("AlreadyReported(uint256,address)", frameFirstEpochId,
↪   oracleOne));
        oracle.reportBeacon(frameFirstEpochId, balanceSum, validatorCount);

        // Now we remove the oracle one member, the `removeMember` function call
↪   `OracleMembers.deleteItem(uint256(memberIdx))`
        // that swap the last array item with the index of the member to be removed then pop the array
        // the problem by doing that is that now `oracleTwo` cannot report because the system think
↪   that oracleTwo is now oracleNow
        // if they register again oracleOne it will be able (in the same epoch) to vote again
        vm.prank(admin);
        oracle.removeMember(oracleOne);

        // This should be true if it was tracked correctly
        // It will STILL return true just because `getMemberReportStatus` do not revert when passed the
↪   address of a not existing member
        // it's try to get `ReportsPositions.get(uint256(-1));` and `uint256(-1)` is
↪   `115792089237316195423570985008687907853269984665640564039457584007913129639935`
        // that is type(uint256).max
        assertEq(oracle.getMemberReportStatus(oracleOne), true);
        // This should be false if tracked correctly because has never reported
        assertEq(oracle.getMemberReportStatus(oracleTwo), true);

        // This SHOULD not revert because oracleTwo has never reported
        vm.prank(oracleTwo);
        vm.expectRevert(abi.encodeWithSignature("AlreadyReported(uint256,address)", frameFirstEpochId,
↪   oracleTwo));
        oracle.reportBeacon(frameFirstEpochId, balanceSum, validatorCount);

        // If we register again oracleOne
        // he will be able to vote even if he/she has already voted
        vm.prank(admin);
        oracle.addMember(oracleOne);

        vm.prank(oracleOne);
        // normally it should revert because oracleOne has already voted
        oracle.reportBeacon(frameFirstEpochId, balanceSum, validatorCount);
```

```
        assertEq(oracle.getMemberReportStatus(oracleOne), true);
        // This should be false if tracked correctly because has never reported
        assertEq(oracle.getMemberReportStatus(oracleTwo), true);
    }
}
```

7.1.2 `Operators._hasFundableKeys` returns `true` for operators that do not have fundable keys test

Here to run the test `forge test --match-test test_getAllFundableDoNotReturnZeroBecauseStopped`

```
// SPDX-License-Identifier: Unlicense
pragma solidity ^0.8.10;

import "forge-std/Test.sol";

import "../src/OperatorsRegistry.1.sol";
import "../src/Allowlist.1.sol";
import "../src/libraries/LibOwnable.sol";
import "../src/state/operatorsRegistry/Operators.sol";

contract WOperatorsRegistryV1 is OperatorsRegistryV1 {
    function isOperatorActiveByName(string calldata _name) external view returns (bool) {
        return Operators.exists(_name);
    }

    function sudoUpdateOperatorData(
        string calldata name,
        uint256 funded,
        uint256 stopped
    ) external {
        Operators.Operator storage operator = Operators.get(name);
        operator.funded = funded;
        operator.stopped = stopped;
    }

    function getAllFundable() external view returns (Operators.CachedOperator[] memory) {
        return Operators.getAllFundable();
    }
}

contract SpearbitTest is Test {
    AllowlistV1 internal allowlist;
    WOperatorsRegistryV1 internal operatorsRegistry;

    address private admin = address(1);
    address private river = address(2);
    address private nodeOperatorFeeRecipient = address(3);

    bytes tenPublicKeys =
        hex"e18e2e0fff999e38c547fb921bd915a1fef6a46e07db2705a57d17542f0acc6be99fee18fb2c17a7f247fc05edd
↪   722b9e95f700adf12dd6f20d65c2e411a6a447cb6f4f7c84b8b6673099385d5dcf964e479a9167c802d977ba0a0a22df233
↪   6f83182e83bc198c34c301482f189475209fb93d362704e317709fec266f14b3383e7bac3e0e306b1ac8228482c31d7ffbc
↪   7e0f865bd4648c7c45c1361f9cd0f4ccab693068a55e6823a4edd8ef4ced9faa64343866ad7357d488bff347a36f2476cc7
↪   b4eaf397c832968548ea70e1de2b2691628399d9f59fdd21a62dd3da50d0255790ec1c15949850a869a37595f8f52cc9d68
↪   1f2b78748254ba36456f7e65d3c7ed2b64ef502cbd3c2f48d789924a99aedfbd097f4ff5829e787ef29b0b79c817337c08f
↪   f68c454a1ddc6359d09c0d48a9298a96691c3558812ca13239d9324f146a6b3bfb7bd3f2b1877929ceb74fa1ed590351801
↪   935d1e61c875c7772e74d037b26f972f2dfa0569ae9bf8cc32dee87e980497cd79492bf91edeafacfbd6ad32c642a6564ff
↪   c9a5d921d0cc35740314c88986547d8f94e60f5d3a2e0a0b1723d20546beaf064adfd78e5f63b600a6ee5b0bacf6f30bf45
↪   6afadef451a1c503e428fcd295b5d20ad9e065e1684dced5487fd33270bb68b783a6963eab250";
    bytes tenSignatures =
```

```
        hex"1d40e9c75a57997aa60105f21d0c68c3be0435f1f68f755c33015110e38ef97b4b5e2008096e75b906ae4d1f5ae
    ↪ c6ad505920c58758631a06ac8c7666b582c9b3fa70a67e39a5aec2a6a11756eb2c0dd74c33beafce3ce8396e86671e00528
    ↪ c0f4c6e9ccef87d852ac23e38f3a1585b710e4eeaa919b7040e49a55583b250801f4f54a7fc1d514ee4a89f2a0a31b50006
    ↪ 5ea383c4ff0d2981da1fc7e047239d1743c2ce0412a424fe7d32628d656c57d56a6f322d696b6099f44f5257e051cbfed14
    ↪ 97e89d637a05b2063a30f232ceba1677e571b08c3c8bb84979d5fcab1652ae40c0840e29a32725d9b45fea996edce566fff
    ↪ f5be6d38d056c6a1827fb7b0ba69f1923a0384e8da72dced43b0e83714d2eda54916252859962b659e0a081fcb4ea39240a
    ↪ fef5b8ae854eb4a159d1bdd9eed58aa368c690fe816052fd3f270215dd0ec0cc55e386ea43cb52849d00cc6a2efbec1d693
    ↪ 619f9dcf64bfbc9c1982d1acc877b898f48d8337b372a338907dc7af4d8d58e08e1302313721fef9d9c1697256b3dc113bf
    ↪ c7860bccf01dc4d28d3ef09e43db4ecdd78c41bdc89ed50130b1b87ba137c33928f9aff8481e204d8ccd86938da666357cb
    ↪ f4d7253ce56eed980dc7ec2d4a610f541d8f7c830592bb39b8048848203ce74db2f585a7b8cd1ae6b47192b1983efddfb42
    ↪ c1ad65a0c9ceae3548565829c1bcd1a06da7ed4e10698fd2e5b2715a3cbae861909d8364f622882e080f0a07e153fb29c7c
    ↪ 3abcc27f2011e76d09d013a5b511b0de51cbefc206c58cb9c09e405b5db80f1c98ea414ae2705de213715e01b81cda31c4e
    ↪ c814e50b341723dc4be0a81268cd04be2d306347540be98c2a3192a968dbad68ba2b5c066c684d18bdcb7345f1d2b643f2f
    ↪ f7bb8494c093eaaf31ee192c9a1486b387701c3bda0ed55af872665fcb0267e3e0b9e7d55e13555ec0f8742229898693ca2
    ↪ 6ddbb8cae390e9ec35fc3e07c21236fa8264974f9d3aea67a059d6e88111f48341ffdfa911ad89a2a2fbb2a790d69f3fc76
    ↪ 46c0071b0c1ca25567f8e2b09d3177abb3e0ec38e4ebe0da06dd6cd03551682fd7ab3a772f5fbc5cba2afbd17d1ed064cae
    ↪ 8db36b6bc6ab33b803a40fc7bc9fc7eb0327ee6486803672f653659cea6cc55212db194b9f7ecf1e0996117a489c5caf861
    ↪ 425743681184ed3dbe5f8de71b5e7479ce13b6e1b5f6368b187185e9e14967ec04bb67d3797e4b0b2f00ee0cc61a7e2b2fe
    ↪ ff0a5ed98d503de0e8e3f445e328e8e8d45dc4453e10f49a6d642cbd90c06d7e81e64cf8d8562f6ef708a5761f1503e221f
    ↪ fa15a5318a880e25f8e3a7e79ce4bd263fdc6f683fb2cd1";

    function setUp() public {
        allowlist = new AllowlistV1();

        operatorsRegistry = new WOperatorsRegistryV1();
        operatorsRegistry.initOperatorsRegistryV1(admin, river);
    }

    function test_getAllFundableDoNotReturnZeroBecauseStopped() public {
        address nop1 = address(4);

        // create operators
        _addOperator("op1", nop1, false);

        vm.startPrank(admin);

        // add 10 validators to op1
        operatorsRegistry.addValidators(0, 10, tenPublicKeys, tenSignatures);

        // raise limit to 10
        uint256[] memory opIdx = new uint256[](1);
        uint256[] memory opLimits = new uint256[](1);
        opIdx[0] = 0;
        opLimits[0] = 10;
        operatorsRegistry.setOperatorLimits(opIdx, opLimits);

        // simulate that op1 get 10 validators funded and then exited
        operatorsRegistry.sudoUpdateOperatorData("op1", 10, 10);

        // getAllFundable() should return an empty array because op1 has no fundable key
        // All keys approved got funded and there are no left keys to be funded
        // but it's still returned because of how it's implemented Operators._hasFundableKeys
        // that is using `keys` and `limit` against `operator.funded - operator.stopped`
        Operators.CachedOperator[] memory operators = operatorsRegistry.getAllFundable();

        // assert the operator select is 1 and should be instead 0
        assertEq(operators.length, 1);

        // assert that op1 should not be selected (no key available to be funded)
        Operators.Operator memory op1 = operatorsRegistry.getOperator(0);
        assertEq(op1.keys, 10);
        assertEq(op1.limit, 10);
```

```
        assertEq(op1.funded, 10);
        assertEq(op1.stopped, 10);
    }

    function _addOperator(
        string memory name,
        address nodeOperatorAddress,
        bool deactivate
    ) public returns (uint256) {
        vm.startPrank(admin);

        operatorsRegistry.addOperator(name, nodeOperatorAddress, nodeOperatorFeeRecipient);

        (int256 operatorIndex, ) = operatorsRegistry.getOperatorDetails(name);
        uint256 index = uint256(operatorIndex);

        if (deactivate) {
            operatorsRegistry.setOperatorStatus(index, false);
        }

        vm.stopPrank();

        return index;
    }
}
```

7.1.3 `OperatorsRegistry._getNextValidatorsFromActiveOperators` can DOS Alluvial staking if there's an operator with `funded==stopped` and `funded == min(limit, keys)` test

To run it `forge test --match-test test_pickNextValidatorReturnEmptyBecauseStopped`

```
// SPDX-License-Identifier: Unlicense
pragma solidity ^0.8.10;

import "forge-std/Test.sol";

import "../src/OperatorsRegistry.1.sol";
import "../src/Allowlist.1.sol";
import "../src/libraries/LibOwnable.sol";
import "../src/state/operatorsRegistry/Operators.sol";

contract WOperatorsRegistryV1 is OperatorsRegistryV1 {
    function isOperatorActiveByName(string calldata _name) external view returns (bool) {
        return Operators.exists(_name);
    }

    function sudoUpdateOperatorData(
        string calldata name,
        uint256 funded,
        uint256 stopped
    ) external {
        Operators.Operator storage operator = Operators.get(name);
        operator.funded = funded;
        operator.stopped = stopped;
    }
}

contract SpearbitTest is Test {
    AllowlistV1 internal allowlist;
    WOperatorsRegistryV1 internal operatorsRegistry;
```

```
    address private admin = address(1);
    address private river = address(2);
    address private nodeOperatorFeeRecipient = address(3);

    bytes tenPublicKeys =
        hex"e18e2e0fff999e38c547fb921bd915a1fef6a46e07db2705a57d17542f0acc6be99fee18fb2c17a7f247fc05edd
↪   722b9e95f700adf12dd6f20d65c2e411a6a447cb6f4f7c84b8b6673099385d5dcf964e479a9167c802d977ba0a0a22df233
↪   6f83182e83bc198c34c301482f189475209fb93d362704e317709fec266f14b3383e7bac3e0e306b1ac8228482c31d7ffbc
↪   7e0f865bd4648c7c45c1361f9cd0f4ccab693068a55e6823a4edd8ef4ced9faa64343866ad7357d488bff347a36f2476cc7
↪   b4eaf397c832968548ea70e1de2b2691628399d9f59fdd21a62dd3da50d0255790ec1c15949850a869a37595f8f52cc9d68
↪   1f2b78748254ba36456f7e65d3c7ed2b64ef502cbd3c2f48d789924a99aedfbd097f4ff5829e787ef29b0b79c817337c08f
↪   f68c454a1ddc6359d09c0d48a9298a96691c3558812ca13239d9324f146a6b3bfb7bd3f2b1877929ceb74fa1ed590351801
↪   935d1e61c875c7772e74d037b26f972f2dfa0569ae9bf8cc32dee87e980497cd79492bf91edeafacfbd6ad32c642a6564ff
↪   c9a5d921d0cc35740314c88986547d8f94e60f5d3a2e0a0b1723d20546beaf064adfd78e5f63b600a6ee5b0bacf6f30bf45
↪   6afadef451a1c503e428fcd295b5d20ad9e065e1684dced5487fd33270bb68b783a6963eab250";
    bytes tenSignatures =
        hex"1d40e9c75a57997aa60105f21d0c68c3be0435f1f68f755c33015110e38ef97b4b5e2008096e75b906ae4d1f5ae
↪   c6ad505920c58758631a06ac8c7666b582c9b3fa70a67e39a5aec2a6a11756eb2c0dd74c33beafce3ce8396e86671e00528
↪   c0f4c6e9ccef87d852ac23e38f3a1585b710e4eeaa919b7040e49a55583b250801f4f54a7fc1d514ee4a89f2a0a31b50006
↪   5ea383c4ff0d2981da1fc7e047239d1743c2ce0412a424fe7d32628d656c57d56a6f322d696b6099f44f5257e051cbfed14
↪   97e89d637a05b2063a30f232ceba1677e571b08c3c8bb84979d5fcab1652ae40c0840e29a32725d9b45fea996edce566fff
↪   f5be6d38d056c6a1827fb7b0ba69f1923a0384e8da72dced43b0e83714d2eda54916252859962b659e0a081fcb4ea39240a
↪   fef5b8ae854eb4a159d1bdd9eed58aa368c690fe816052fd3f270215dd0ec0cc55e386ea43cb52849d00cc6a2efbec1d693
↪   619f9dcf64bfbc9c1982d1acc877b898f48d8337b372a338907dc7af4d8d58e08e1302313721fef9d9c1697256b3dc113bf
↪   c7860bccf01dc4d28d3ef09e43db4ecdd78c41bdc89ed50130b1b87ba137c33928f9aff8481e204d8ccd86938da666357cb
↪   f4d7253ce56eed980dc7ec2d4a610f541d8f7c830592bb39b8048848203ce74db2f585a7b8cd1ae6b47192b1983efddfb42
↪   c1ad65a0c9ceae3548565829c1bcd1a06da7ed4e10698fd2e5b2715a3cbae861909d8364f622882e080f0a07e153fb29c7c
↪   3abcc27f2011e76d09d013a5b511b0de51cbefc206c58cb9c09e405b5db80f1c98ea414ae2705de213715e01b81cda31c4e
↪   c814e50b341723dc4be0a81268cd04be2d306347540be98c2a3192a968dbad68ba2b5c066c684d18bdcb7345f1d2b643f2f
↪   f7bb8494c093eaaf31ee192c9a1486b387701c3bda0ed55af872665fcb0267e3e0b9e7d55e13555ec0f8742229898693ca2
↪   6ddbb8cae390e9ec35fc3e07c21236fa8264974f9d3aea67a059d6e88111f48341ffdfa911ad89a2a2fbb2a790d69f3fc76
↪   46c0071b0c1ca25567f8e2b09d3177abb3e0ec38e4ebe0da06dd6cd03551682fd7ab3a772f5fbc5cba2afbd17d1ed064cae
↪   8db36b6bc6ab33b803a40fc7bc9fc7eb0327ee6486803672f653659cea6cc55212db194b9f7ecf1e0996117a489c5caf861
↪   425743681184ed3dbe5f8de71b5e7479ce13b6e1b5f6368b187185e9e14967ec04bb67d3797e4b0b2f00ee0cc61a7e2b2fe
↪   ff0a5ed98d503de0e8e3f445e328e8e8d45dc4453e10f49a6d642cbd90c06d7e81e64cf8d8562f6ef708a5761f1503e221f
↪   fa15a5318a880e25f8e3a7e79ce4bd263fdc6f683fb2cd1";

    function setUp() public {
        allowlist = new AllowlistV1();

        operatorsRegistry = new WOperatorsRegistryV1();
        operatorsRegistry.initOperatorsRegistryV1(admin, river);
    }
function test_pickNextValidatorReturnEmptyBecauseStopped() public {
        Operators.Operator memory op1;
        Operators.Operator memory op2;

        address nop1 = address(4);
        address nop2 = address(5);

        // create operators
        _addOperator("op1", nop1, false);
        _addOperator("op2", nop2, false);

        vm.startPrank(admin);

        // add 10 validators for each operator
        operatorsRegistry.addValidators(0, 10, tenPublicKeys, tenSignatures);
        operatorsRegistry.addValidators(1, 10, tenPublicKeys, tenSignatures);

        // raise their limit to 10
        uint256[] memory opIdx = new uint256[](2);
```

```
        uint256[] memory opLimits = new uint256[](2);
        opIdx[0] = 0;
        opIdx[1] = 1;
        opLimits[0] = 10;
        opLimits[1] = 10;
        operatorsRegistry.setOperatorLimits(opIdx, opLimits);

        // assert things are ok
        op1 = operatorsRegistry.getOperator(0);
        assertEq(op1.keys, 10);
        assertEq(op1.limit, 10);
        assertEq(op1.funded, 0);
        assertEq(op1.stopped, 0);

        op2 = operatorsRegistry.getOperator(1);
        assertEq(op2.keys, 10);
        assertEq(op2.limit, 10);
        assertEq(op2.funded, 0);
        assertEq(op2.stopped, 0);

        // simulate that op1 get 10 validators funded and then exited
        operatorsRegistry.sudoUpdateOperatorData("op1", 10, 10);

        // assert they
        op1 = operatorsRegistry.getOperator(0);
        assertEq(op1.keys, 10);
        assertEq(op1.limit, 10);
        assertEq(op1.funded, 10);
        assertEq(op1.stopped, 10);
        vm.stopPrank();

        // make river pick 5 validators. it should pick 5 from op2 because has lower funded value
        // but it will take op1 because of the stopped check in the for loop
        vm.prank(river);
        (bytes[] memory publicKeys, bytes[] memory signatures) =
↪   operatorsRegistry.pickNextValidators(5);
        assertEq(publicKeys.length, 0);
        assertEq(signatures.length, 0);

        op2 = operatorsRegistry.getOperator(1);
        assertEq(op2.keys, 10);
        assertEq(op2.limit, 10);
        assertEq(op2.funded, 0);
        assertEq(op2.stopped, 0);
    }

    function _addOperator(
        string memory name,
        address nodeOperatorAddress,
        bool deactivate
    ) public returns (uint256) {
        vm.startPrank(admin);

        operatorsRegistry.addOperator(name, nodeOperatorAddress, nodeOperatorFeeRecipient);

        (int256 operatorIndex, ) = operatorsRegistry.getOperatorDetails(name);
        uint256 index = uint256(operatorIndex);

        if (deactivate) {
            operatorsRegistry.setOperatorStatus(index, false);
        }
```

```
        vm.stopPrank();

        return index;
    }
}
```

7.1.4 `OperatorsRegistry._getNextValidatorsFromActiveOperators` should not consider `stopped` when picking a validator test

to run it `forge test --match-test test_pickNextValidatorPickOpWithHighStoppedValue`

```
// SPDX-License-Identifier: Unlicense
pragma solidity ^0.8.10;

import "forge-std/Test.sol";

import "../src/OperatorsRegistry.1.sol";
import "../src/Allowlist.1.sol";
import "../src/libraries/LibOwnable.sol";
import "../src/state/operatorsRegistry/Operators.sol";

contract WOperatorsRegistryV1 is OperatorsRegistryV1 {
    function isOperatorActiveByName(string calldata _name) external view returns (bool) {
        return Operators.exists(_name);
    }

    function sudoUpdateOperatorData(
        string calldata name,
        uint256 funded,
        uint256 stopped
    ) external {
        Operators.Operator storage operator = Operators.get(name);
        operator.funded = funded;
        operator.stopped = stopped;
    }
}

contract SpearbitTest is Test {
    AllowlistV1 internal allowlist;
    WOperatorsRegistryV1 internal operatorsRegistry;

    address private admin = address(1);
    address private river = address(2);
    address private nodeOperatorFeeRecipient = address(3);

    bytes tenPublicKeys =
        hex"e18e2e0fff999e38c547fb921bd915a1fef6a46e07db2705a57d17542f0acc6be99fee18fb2c17a7f247fc05edd
↪   722b9e95f700adf12dd6f20d65c2e411a6a447cb6f4f7c84b8b6673099385d5dcf964e479a9167c802d977ba0a0a22df233
↪   6f83182e83bc198c34c301482f189475209fb93d362704e317709fec266f14b3383e7bac3e0e306b1ac8228482c31d7ffbc
↪   7e0f865bd4648c7c45c1361f9cd0f4ccab693068a55e6823a4edd8ef4ced9faa64343866ad7357d488bff347a36f2476cc7
↪   b4eaf397c832968548ea70e1de2b2691628399d9f59fdd21a62dd3da50d0255790ec1c15949850a869a37595f8f52cc9d68
↪   1f2b78748254ba36456f7e65d3c7ed2b64ef502cbd3c2f48d789924a99aedfbd097f4ff5829e787ef29b0b79c817337c08f
↪   f68c454a1ddc6359d09c0d48a9298a96691c3558812ca13239d9324f146a6b3bfb7bd3f2b1877929ceb74fa1ed590351801
↪   935d1e61c875c7772e74d037b26f972f2dfa0569ae9bf8cc32dee87e980497cd79492bf91edeafacfbd6ad32c642a6564ff
↪   c9a5d921d0cc35740314c88986547d8f94e60f5d3a2e0a0b1723d20546beaf064adfd78e5f63b600a6ee5b0bacf6f30bf45
↪   6afadef451a1c503e428fcd295b5d20ad9e065e1684dced5487fd33270bb68b783a6963eab250";
    bytes tenSignatures =
```

```
        hex"1d40e9c75a57997aa60105f21d0c68c3be0435f1f68f755c33015110e38ef97b4b5e2008096e75b906ae4d1f5ae
    ↪ c6ad505920c58758631a06ac8c7666b582c9b3fa70a67e39a5aec2a6a11756eb2c0dd74c33beafce3ce8396e86671e00528
    ↪ c0f4c6e9ccef87d852ac23e38f3a1585b710e4eeaa919b7040e49a55583b250801f4f54a7fc1d514ee4a89f2a0a31b50006
    ↪ 5ea383c4ff0d2981da1fc7e047239d1743c2ce0412a424fe7d32628d656c57d56a6f322d696b6099f44f5257e051cbfed14
    ↪ 97e89d637a05b2063a30f232ceba1677e571b08c3c8bb84979d5fcab1652ae40c0840e29a32725d9b45fea996edce566fff
    ↪ f5be6d38d056c6a1827fb7b0ba69f1923a0384e8da72dced43b0e83714d2eda54916252859962b659e0a081fcb4ea39240a
    ↪ fef5b8ae854eb4a159d1bdd9eed58aa368c690fe816052fd3f270215dd0ec0cc55e386ea43cb52849d00cc6a2efbec1d693
    ↪ 619f9dcf64bfbc9c1982d1acc877b898f48d8337b372a338907dc7af4d8d58e08e1302313721fef9d9c1697256b3dc113bf
    ↪ c7860bccf01dc4d28d3ef09e43db4ecdd78c41bdc89ed50130b1b87ba137c33928f9aff8481e204d8ccd86938da666357cb
    ↪ f4d7253ce56eed980dc7ec2d4a610f541d8f7c830592bb39b8048848203ce74db2f585a7b8cd1ae6b47192b1983efddfb42
    ↪ c1ad65a0c9ceae3548565829c1bcd1a06da7ed4e10698fd2e5b2715a3cbae861909d8364f622882e080f0a07e153fb29c7c
    ↪ 3abcc27f2011e76d09d013a5b511b0de51cbefc206c58cb9c09e405b5db80f1c98ea414ae2705de213715e01b81cda31c4e
    ↪ c814e50b341723dc4be0a81268cd04be2d306347540be98c2a3192a968dbad68ba2b5c066c684d18bdcb7345f1d2b643f2f
    ↪ f7bb8494c093eaaf31ee192c9a1486b387701c3bda0ed55af872665fcb0267e3e0b9e7d55e13555ec0f8742229898693ca2
    ↪ 6ddbb8cae390e9ec35fc3e07c21236fa8264974f9d3aea67a059d6e88111f48341ffdfa911ad89a2a2fbb2a790d69f3fc76
    ↪ 46c0071b0c1ca25567f8e2b09d3177abb3e0ec38e4ebe0da06dd6cd03551682fd7ab3a772f5fbc5cba2afbd17d1ed064cae
    ↪ 8db36b6bc6ab33b803a40fc7bc9fc7eb0327ee6486803672f653659cea6cc55212db194b9f7ecf1e0996117a489c5caf861
    ↪ 425743681184ed3dbe5f8de71b5e7479ce13b6e1b5f6368b187185e9e14967ec04bb67d3797e4b0b2f00ee0cc61a7e2b2fe
    ↪ ff0a5ed98d503de0e8e3f445e328e8e8d45dc4453e10f49a6d642cbd90c06d7e81e64cf8d8562f6ef708a5761f1503e221f
    ↪ fa15a5318a880e25f8e3a7e79ce4bd263fdc6f683fb2cd1";

    function setUp() public {
        allowlist = new AllowlistV1();

        operatorsRegistry = new WOperatorsRegistryV1();
        operatorsRegistry.initOperatorsRegistryV1(admin, river);
    }

    function test_pickNextValidatorPickOpWithHighStoppedValue() public {
        Operators.Operator memory op1;
        Operators.Operator memory op2;

        address nop1 = address(4);
        address nop2 = address(5);

        // create operators
        _addOperator("op1", nop1, false);
        _addOperator("op2", nop2, false);

        vm.startPrank(admin);

        // add 10 validators for each operator
        operatorsRegistry.addValidators(0, 10, tenPublicKeys, tenSignatures);
        operatorsRegistry.addValidators(1, 10, tenPublicKeys, tenSignatures);

        // raise their limit to 10
        uint256[] memory opIdx = new uint256[](2);
        uint256[] memory opLimits = new uint256[](2);
        opIdx[0] = 0;
        opIdx[1] = 1;
        opLimits[0] = 10;
        opLimits[1] = 10;
        operatorsRegistry.setOperatorLimits(opIdx, opLimits);

        // assert things are ok
        op1 = operatorsRegistry.getOperator(0);
        assertEq(op1.keys, 10);
        assertEq(op1.limit, 10);
        assertEq(op1.funded, 0);
        assertEq(op1.stopped, 0);

        op2 = operatorsRegistry.getOperator(1);
```

```
        assertEq(op2.keys, 10);
        assertEq(op2.limit, 10);
        assertEq(op2.funded, 0);
        assertEq(op2.stopped, 0);

        // simulate that op1 get 5 validators funded and then exited
        operatorsRegistry.sudoUpdateOperatorData("op1", 5, 5);

        // assert they
        op1 = operatorsRegistry.getOperator(0);
        assertEq(op1.keys, 10);
        assertEq(op1.limit, 10);
        assertEq(op1.funded, 5);
        assertEq(op1.stopped, 5);
        vm.stopPrank();

        // make river pick 5 validators. it should pick 5 from op2 because has lower funded value
        // but it will take op1 because of the stopped check in the for loop
        vm.prank(river);
        operatorsRegistry.pickNextValidators(5);

        // assert the function behave INCORRECTLY
        op1 = operatorsRegistry.getOperator(0);
        assertEq(op1.keys, 10);
        assertEq(op1.limit, 10);
        assertEq(op1.funded, 10);
        assertEq(op1.stopped, 5);

        op2 = operatorsRegistry.getOperator(1);
        assertEq(op2.keys, 10);
        assertEq(op2.limit, 10);
        assertEq(op2.funded, 0);
        assertEq(op2.stopped, 0);
    }

    function _addOperator(
        string memory name,
        address nodeOperatorAddress,
        bool deactivate
    ) public returns (uint256) {
        vm.startPrank(admin);

        operatorsRegistry.addOperator(name, nodeOperatorAddress, nodeOperatorFeeRecipient);

        (int256 operatorIndex, ) = operatorsRegistry.getOperatorDetails(name);
        uint256 index = uint256(operatorIndex);

        if (deactivate) {
            operatorsRegistry.setOperatorStatus(index, false);
        }

        vm.stopPrank();

        return index;
    }
}
```

7.1.5 Consider adding a strict check to prevent `Oracle` admin to add more than 256 members test

The following test is run on the project's code at snapshot 2189a50a8ccdb4db471a66c5ad018cf660bf3896

```
//SPDX-License-Identifier: MIT

pragma solidity 0.8.10;

import "../src/Oracle.1.sol";
import "../src/Withdraw.1.sol";
import "./utils/UserFactory.sol";
import "./mocks/RiverMock.sol";
import "../src/interfaces/IRiver.1.sol";

import "forge-std/Test.sol";
import "forge-std/console2.sol";

contract WOracle is OracleV1 {
    function hasReported(address om) external returns (bool) {
        int256 memberIndex = OracleMembers.indexOf(om);
        if (memberIndex == -1) {
            revert LibErrors.Unauthorized(om);
        }

        return ReportsPositions.get(uint256(memberIndex));
    }

    function getMemberIndex(address om) external returns (int256) {
        return OracleMembers.indexOf(om);
    }
}

contract SpearOracleTest is Test {
    address[] oracleMemberList;
    WOracle internal oracle;

    IRiverV1 internal oracleInput;
    UserFactory internal uf = new UserFactory();

    address internal admin = address(0xEA674fdDe714fd979de3EdF0F56AA9716B898ec8);

    address internal oracleOne = address(0x7fe52bbF4D779cA115231b604637d5f80bab2C40);
    address internal oracleTwo = address(0xb479DE67E0827Cc72bf5c1727e3bf6fe15007554);

    uint64 internal constant EPOCHS_PER_FRAME = 225;
    uint64 internal constant SLOTS_PER_EPOCH = 32;
    uint64 internal constant SECONDS_PER_SLOT = 12;
    uint64 internal constant GENESIS_TIME = 1606824023;

    uint256 internal constant UPPER_BOUND = 1000;
    uint256 internal constant LOWER_BOUND = 500;

    function setUp() public {
        oracleInput = IRiverV1(payable(address(new RiverMock())));
        oracle = new WOracle();
        oracle.initOracleV1(
            address(oracleInput),
            admin,
            EPOCHS_PER_FRAME,
            SLOTS_PER_EPOCH,
            SECONDS_PER_SLOT,
            GENESIS_TIME,
            UPPER_BOUND,
            LOWER_BOUND
        );
```

```
    }

    function testAddingMoreThan256OracleMember() public {
        uint64 timeFromGenesis = 1;
        uint64 balanceSum = 1;
        uint32 validatorCount = 1;
        vm.warp(uint256(GENESIS_TIME) + uint256(timeFromGenesis));

        uint256 epochId = oracle.getCurrentEpochId();
        uint256 frameFirstEpochId = oracle.getFrameFirstEpochId(epochId);

        vm.startPrank(admin);

        // Add 256 oracle members (max supported by ReportsPosition)
        for (uint256 i = 0; i < 256; i++) {
            address om = uf._new(i);
            oracle.addMember(om, i + 1);
            oracleMemberList.push(om);
        }

        // Add an additional member (256th oracle member)
        // One more than the max number of supported by ReportsPosition.sol
        address oracleMember257 = uf._new(257);
        oracle.addMember(oracleMember257, 256 + 1);
        oracleMemberList.push(oracleMember257);

        vm.stopPrank();

        // We have more than 256 oracle members added to the list of members
        assertEq(oracle.getOracleMembers().length, 257);
        assertEq(oracle.getMemberIndex(oracleMember257), 256);

        // The `oracleMember257` is seen as a real member
        assertEq(oracle.isMember(oracleMember257), true);

        // The `oracleMember257` is a member but cannot report because
        assertEq(oracle.getMemberReportStatus(oracleMember257), true);

        // if we try to vote it will revert because it enter inside the if branch that result in
↪   `revert AlreadyReported(_epochId, msg.sender);`
        vm.prank(oracleMember257);
        vm.expectRevert(
            abi.encodeWithSignature("AlreadyReported(uint256,address)", frameFirstEpochId,
↪   oracleMember257)
        );
        oracle.reportConsensusLayerData(frameFirstEpochId, balanceSum, validatorCount);

        // Remove the second oracle from the list
        vm.prank(admin);
        oracle.removeMember(oracleMemberList[1], 256);

        // We now have correctly 256 members
        assertEq(oracle.getOracleMembers().length, 256);

        // The `oracleMember257` has been "swapped" with the removed member (index 1 in the list)
        assertEq(oracle.getMemberIndex(oracleMember257), 1);

        // The `oracleMember257` is still seen as a real member
        assertEq(oracle.isMember(oracleMember257), true);

        // The `oracleMember257` now is able to vote
        assertEq(oracle.getMemberReportStatus(oracleMember257), false);
```

```
        // it can successfully vote
        vm.prank(oracleMember257);
        oracle.reportConsensusLayerData(frameFirstEpochId, balanceSum, validatorCount);
    }
}
```

7.1.6 Decrementing the quorum in `Oracle` in some scenarios can open up a frontrunning/backrunning opportunity for some oracle members test

```
//SPDX-License-Identifier: MIT

pragma solidity 0.8.10;

import "../src/Oracle.1.sol";
import "../src/Withdraw.1.sol";
import "./utils/UserFactory.sol";
import "./mocks/RiverMock.sol";
import "../src/interfaces/IRiver.1.sol";

import "forge-std/Test.sol";

contract SpearOracleTest is Test {
    OracleV1 internal oracle;

    IRiverV1 internal oracleInput;
    UserFactory internal uf = new UserFactory();

    address internal admin = address(0xEA674fdDe714fd979de3EdF0F56AA9716B898ec8);

    address internal oracleOne = address(0x7fe52bbF4D779cA115231b604637d5f80bab2C40);
    address internal oracleTwo = address(0xb479DE67E0827Cc72bf5c1727e3bf6fe15007554);

    uint64 internal constant EPOCHS_PER_FRAME = 225;
    uint64 internal constant SLOTS_PER_EPOCH = 32;
    uint64 internal constant SECONDS_PER_SLOT = 12;
    uint64 internal constant GENESIS_TIME = 1606824023;

    uint256 internal constant UPPER_BOUND = 1000;
    uint256 internal constant LOWER_BOUND = 500;

    function setUp() public {
        oracleInput = IRiverV1(payable(address(new RiverMock())));
        oracle = new OracleV1();
        oracle.initOracleV1(
            address(oracleInput),
            admin,
            EPOCHS_PER_FRAME,
            SLOTS_PER_EPOCH,
            SECONDS_PER_SLOT,
            GENESIS_TIME,
            UPPER_BOUND,
            LOWER_BOUND
        );
    }

    event PostTotalShares(uint256 postTotalEth, uint256 prevTotalEth, uint256 timeElapsed, uint256
↪   totalShares);

    function testRemoveMemberPushToRiver() public {
```

```
        uint64 timeFromGenesis = 1;
        uint64 balanceSum = 1;
        uint32 validatorCount = 1;

        RiverMock(address(oracleInput)).sudoSetTotalShares(1e9 * uint256(balanceSum));
        RiverMock(address(oracleInput)).sudoSetTotalSupply(1e9 * uint256(balanceSum));
        vm.warp(uint256(GENESIS_TIME) + uint256(timeFromGenesis));

        address om1 = uf._new(1);
        address om2 = uf._new(2);
        vm.startPrank(admin);
        oracle.addMember(om1, 1);
        oracle.addMember(om2, 2);
        vm.stopPrank();

        uint256 epochId = oracle.getCurrentEpochId();
        uint256 frameFirstEpochId = oracle.getFrameFirstEpochId(epochId);

        vm.prank(om2);
        oracle.reportConsensusLayerData(frameFirstEpochId, balanceSum, validatorCount);

        vm.prank(admin);
        vm.expectEmit(false, false, false, false);
        emit PostTotalShares(1000000000, 1000000000, 0, 1000000000);
        oracle.removeMember(om2, 1);
    }
}
```