Kiln vSuite Security Review

Auditors

Emanuele Ricci, Lead Security Researcher

Optimum, Lead Security Researcher

Xiaoming90, Security Researcher

Akshay Srivastav, Associate Security Researcher

Report prepared by: Lucas Goiriz

May 23, 2024

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

Kiln is a staking platform you can use to stake directly, or whitelabel staking into your product. It enables users to stake crypto assets, manually or programmatically, while maintaining custody of your funds in your existing solution, such Fireblocks, Copper, or Ledger.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of vsuite 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 15 days in total, Kiln engaged with Spearbit to review the vsuite protocol. In this period of time a total of **60** issues were found.

Summary

Project Name	Kiln
Repository	vsuite
Commit	ad04b6...ad04b6
Type of Project	Enterprise Staking, DeFi
Audit Timeline	Mar 18 to May 4
Two week fix period	May 5 - May 7

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	3	1	2
Medium Risk	11	10	1
Low Risk	14	11	3
Gas Optimizations	5	4	1
Informational	27	17	10
Total	**60**	**43**	**17**

5 Findings

5.1 High Risk

5.1.1 `MultiPool` implementation lacks a proper way to fully exit a `vPool` when multiple `vPool` are supported

Severity: High Risk

Context: MultiPool20.sol#L126-L189

Description: Let's assume we have two pools, `vPool1` and `vPool2`, with a 10/90 split. `100 ETH` have been deposited and given such split, the integrator owns `10 ETH` worth of `vPool1` shares and `90 ETH` worth of `vPool2` shares.

The integrator wants to decommission and eject fully from `vPool1`. In this case, the `admin` would perform the following operations:

1) Set `$poolActivation.get()[vPool1] = false`.
2) Set `$poolRoutingList[vPool1] = 0`.

As a consequence, user's won't be able to stake into `vPool1` anymore (the pool is skipped when `split == 0` in the `_stake` logic). This is a correct behavior, given that the integrator should not supply anymore into an "ejected" pool.

Let's now see how the integrator's logic handles the withdrawal process that should bring the `vPool1` shares owned by the integrator to `0` to fully eject from such `vPool`.

If `ethValue` (`vPool` shares worth in ETH) is `>= $monoTicketThreshold.get()` the logic will try to withdraw from each pool given the split associated to them:

```
uint256[] memory splits = $poolRoutingList.toUintA();

/// [OTHER CODE]

PoolExitDetails[] memory details = new PoolExitDetails[](poolCount);
for (uint256 id = 0; id < poolCount;) {
    uint256 ethForPool = LibBasisPoints.compute(ethValue, splits[id]);
    if (ethForPool > 0) _sendToExitQueue(poolInfos[id], ethForPool, details[id], account);
    _checkCommissionRatio(poolInfos[id]);
    unchecked {
        id++;
    }
}
```

Because `$poolRoutingList[vPool1] == 0`, the `ethForPool` related to `vPool1` will be equal to `0` and the withdrawal process will only withdraw from `vPool2`. This brings two different side effects:

- No amount has been exited `vPool1`, the one that should be fully ejected.
- All the user's amount has been exited `vPool2` creating an even greater imbalance in the pool (given that it should have the 100% of the balance).

At this point the only possible solution would be to leverage the `$monoTicketThreshold` logic to "force" the withdrawal from `vPool1` that has an `imbalance` equal to the whole balance. Let's assume that the admin has set `$monoTicketThreshold` equal to `1000 ETH`.

For `vPool1` we have:

```
expectedValue = LibBasisPoints.compute(10 ETH, 0)  -> 0 ETH
uint256 poolValue = _ethAfterCommission(vPool1) -> ~10 ETH (let's assume that there are no commission
↪  to make things easier)
int256 imbalance = int256(poolValue) - int256(expectedValue) -> 10 ETH - 0 ETH = +10 ETH
```

Because of that, normally the `vPool1` would be selected to be the one from which perform the full withdrawal of the user amount. But this would not be the case if the user is trying to withdraw an `ethValue` that is greater than the `poolValue`. If the user is trying to withdraw `> 10ETH` this logic will be skipped, and we would be back to the loop-problem we have seen before. Note that this logic is not applied to `vPool2` because for such pool the `imbalance` is **negative** (`expectedValue == 100 ETH`, 'poolValue == 90 ETH).

Recommendation: Kiln should implement a proper and effective way to fully eject from a `vPool`.

Kiln: Kiln acknowledges this issue with the following context: having the possibility to moves the funds in a different pool would make the code more custodial, as currently only tokens holders can pull funds out of the pools. It also would lead to loss of rewards that would accrue during the time of the unstaking to fully eject a pool. And make users unable to access their liquidity during the process unless some complex logic is added. The pool operator going rogue is covered by legal agreements, as preventing this scenario would be the main reason for those changes we prefer not giving the admin the capability to do this.

Spearbit: The reason why the integrator admin wants to eject a pool is important in this context.

Let's say that there's the need to exit the pool, the admin set `$poolRoutingList[vPool1] = 0` (because this is the only way he can perform it).

If the `ethValue >= $monoTicketThreshold.get()` or if `ethValue < $monoTicketThreshold.get()` but the pool has not enough value to handle the `ethValue` request, the `$monoTicketThreshold` logic will be skipped totally for the pool that needs to be ejected.

If the `$monoTicketThreshold` logic is skipped, the exit-from-each-pool-by-split will anyway skip the need-to-be-ejected-pool because `ethForPool` for such pool will be equal to `0`.

Currently, it seems that there is not a valid implementation that handles the scenario where an integrator wants to eject the users from a `vPool`.

Kiln: To clarify I don't think there is a scenario that requires exiting all the shares very quickly. As it is not a hard need i don't think it's worth adding a feature that gives the integrator temporary custody of user funds. Due to concerns on the regulatory aspect and the added complexity we chose to not include this feature. If needed the re balancing can be done over a a few days to a few weeks depending on the volume on ETH in and out of the contract compared to the TVL using existing mechanisms.

Spearbit: Acknowledged.

5.1.2 `MultiPool20._performRequestExit` logic could revert because `MultiPool` does not own enough shares of a specific `vPool` to fulfill the request

Severity: High Risk

Context: MultiPool20.sol#L179-L187

Description: When the integrator's contract supports multiple `vPool`s, the exit process (when `ethValue >= $monoTicketThreshold.get()`) will try to exit the user position by looping thought all the `vPool`s and withdrawing a number of shares from each pool given the `split` associated to each pool.

This logic is not checking if the integrator has enough `vPool` shares to fulfill this exit request. The underlying `pool.transferShares(pool.exitQueue(), shares, abi.encodePacked(ticketOwner));` performed by `_sendSharesToExitQueue` could revert because the integrator contract does not own enough shares of that specific pool.

This could happen because:

1) The `vPool` splits have been changed, and this pool has higher split compared to before.
2) The `vPool` had losses but other `vPool` have gains (total overall supply and underlying supply has rewards).

Recommendation: Kiln should check if the integrator contract owns enough `vPool` shares to fulfill the `ethForPool` withdrawal request. If `ethForPool > valueOfSharesOwned` the amount withdrawn should be bound to `valueOfSharesOwned` and the remaining should be split between the rest of the pools. The correct implementation of this logic is OOS.

Kiln: This is an edge case that would only happen if there is a big exit after a radical change in the weights of the pools. We have an alternative implementation in mind that fixes this but this was not the priority for this upgrade as this issue is unlikely to happen and has solutions, like exiting in multiple calls with smaller amounts or changing the `monoTicketThreshold`.

Spearbit: Acknowledged.

5.1.3 `vFactory.storage_migration.migrate`: `$fundedValidators` is not increased for the migrated validators

Severity: High Risk

Context: vFactory.storage_migration.sol#L298

Description: `vFactory.storage_migration.migrate` performs an "off-code" insertion of the current deposits (deposits made on mainnet) to the new data structures introduced in the 2.1 version.The term "off-code" means that the newly introduced `vFactory.depositFromRoot` function is not used , but rather, the migration logic skips it and amends the state directly.

During the `depositFromRoot` function the newly introduced storage variable `$fundedValidators` is increased per withdrawal channel and `validationKeysLength` of a successful transaction. However, that is not the case in the migration script, which does not count the migrated validators in the value `$fundedValidators`. The implication is that the `exitTotal` function will read a corrupt value from storage evaluated in the `funded` local variable which will cause an error in the the exit process of validators during the oracle reporting flow. The `$fundedValidators` will hold a lower value than what it should which will result in an unintended capping of the number of validators that should exit. This in turn will result in a significant delay of withdrawal queue fulfillments in the best case or an effective block of withdrawal fulfillments in the worst case.

Recommendation: Consider increasing the value of `$fundedValidators` per the withdrawal channel to mimic the `depositFromRoot` function.

Kiln: This one was fixed with the final migrations commit that has been merged to `vsuite-2-1`, i.e. commit f8bda10e.

Spearbit: Fixed in commit f8bda10e by implementing the auditor's recommendation.

5.2 Medium Risk

5.2.1 `vPool $reportBounds.get().maxAPRUpperBound` could prevent the users from receiving the full amount of ETH deserved

Severity: Medium Risk

Context: vOracleAggregator.sol#L222

Description: The `$reportBounds.get().maxAPRUpperBound` upper bound is designed as a limit that should prevent the share `rate` to grow above a threshold that is calculated based on the total underlying supply (before the report) and the number of seconds that has passed since the previous report.

This upper bound is calculated and used when the `vPoolReporting.driver report` function is executed:

```
uint256 period = _epochTimestamp(cls, rprt.epoch) - _epochTimestamp(cls, lastRprt.epoch);
//
↪                            //
//  ----- The maximum allowed balance increase is now computed
↪                            //
__.traces.increaseLimit = uint128(_maxAllowedBalanceIncrease(__.traces.preUnderlyingSupply, period));
__.increaseCredit = __.traces.increaseLimit;
```

`__.increaseCredit` is then modified based on the performance of the validators between the previous and current report. It will be increased if rewards earned by validators are lower compared to the slashes/penalties; otherwise it will decrease.

`__.increaseCredit` will be used inside the `report` logic to determine how much `ETH` can be pulled from various sources to maintain the `vPool` rate inside the required upper bound.

Let's assume that we are in a scenario where the `int128(uint128(historicalVolume)) - int128(uint128(_-_.preHistoricalVolume)) == __.increaseCredit`, this means that `__.increaseCredit` will be decreased to zero. This is the scenario where the validators have earned more rewards than slashes/penalties and the delta is equal to the required upper bound.

Because now `__.increaseCredit = 0` no more ETH will be pulled from all the sources of revenue:

- Execution Layer Recipient that contains all the EL rewards (block proposal, MEV boosts, gas tips).
- Exit Queue unclaimed funds.

Because these funds are not pulled, the `vPool` rate will not grow as much as it should be and users won't be able to receive all the ETH they should receive if they perform an exit just after the execution of the `report` function.

Recommendation: Kiln should carefully configure the upper bound growth limit to consider all the gains that should be pulled from the EL recipient and the Exit Queue.

Kiln should document and inform the users that because of such an upper bound limit, it's possible that all the gains accrued in the EL recipient and the Exit Queue will not be used (pulled) to contribute to the increase of the `vPool` share price. As a consequence, exiting the `vPool` could lead to an amount of ETH lower compared to the one deserved (in the scenario where all the gains would have been pulled)

Kiln: Acknowledged. The ability to have a predictable rate is important for us, so we don't want to remove this for the exec layer rewards. This limits the ability of users to game the protocol outside of its assumptions. We prefer to keep it like it is.

Spearbit: Acknowledged.

5.2.2 `vCoverageRecipient` admin shouldn't be able to skim what's needed to cover the `vPool` losses

Severity: Medium Risk

Context: vCoverageRecipient.sol#L136-L171

Description: The `vCoverageRecipient` contract allows the `admin` of the contract to skim `ETH` and `vPool` shares from the contract and transfer them to an arbitrary `recipient`. Currently, there's no logic that prevents the `admin` of the contract to remove the whole amount that is owned by the `vCoverageRecipient`.

While the `vPool` contract is still active, the `admin` should not be able to skim what would be needed to cover the current losses of the `vPool` that are reported during the Oracle report.

The amount of losses that needs to be covered (until there's a new, updated report) can be calculated by querying the `vPool` and retrieving the `$coveredBalanceSum.get()` and `$lastReport.get().slashedSum`.

If `$lastReport.get().slashedSum - $coveredBalanceSum.get() > 0` it means that there are slashes that still have to be covered by the coverage funds. The admin of the Coverage Recipient contract should not be able to skim that amount that will need (if not more, depending on the next oracle report) to cover the `vPool` losses.

Recommendation: The `vCoverageRecipient` should not allow the `admin` to skim the `ETH` and `vPool` shares needed to cover the known `vPool` losses.

Kiln: Fixed in PR 148.

Spearbit: The recommendations have been implemented in commit 5f42939f of PR 148.

5.2.3 Toggling the `$transferEnabled` flag of `vExitQueue` NFT can break its composability

Severity: Medium Risk

Context: vExitQueue.sol#L206

Description: The admin of `vExitQueue` possesses the right to enable or disable the `$transferEnabled` flag anytime. This flag is used to allow or disallow the NFT holders to transfer their NFTs to any other addresses. Toggling of this transfer enabled restriction can lead to problems. Consider the following scenario:

- Suppose the NFT transfers are enabled by admin.
- Now users will start moving around the NFTs to different addresses. An obvious case will be when users want to sell/collateralize their NFTs to get instant ETH liquidity.
- Now disabling the transfers again can break the `vExitQueue` NFT's composability with the larger DeFi ecosystem.

Recommendation: Consider implementing a one way toggle, essentially once enabled the transfer of NFTs should not be disabled.

Kiln: Fixed in PR 146.

Spearbit: Fixed as suggested in commit 30a928f2 of PR 146.

5.2.4 DoS of user exit flow if fee split of any integrator commission recipient is set to `0`

Severity: Medium Risk

Context: FeeDispatcher.sol#L114-L119 MultiPool.sol#L342-L345 vPool.sol#L156

Description: In `Multipool20` integration contract the integrator commission are disbursed automatically on every user exit request. This commission is splitted as per the `FeeDispatcher.$feeSplits` & `FeeDispatcher.$feeRecipients` configuration which is set by the admin using `FeeDispatcher._setFeeSplit` function.

But in case a fee recipient's fee split is set as `0` then the exit flow of all users will get broken, causing DoS of the exit operation.

Here is the scenario:

- A recipient's split is set as 0.
- A user who has a stake in MultiPool20 contract calls `requestExit`.
- The contract performs `_checkCommissionRatio` and calls the `_exitCommissionShares` function.
- The `_exitCommissionShares` calculates proportionate pool share of `recipient` (`= 0`) and tries to transfer `0` pool shares to the `vExitQueue`.
- The `vPool.transferShares` function has the check `LibSanitize.notNullValue` which enforces that the transfer amount must not be equal to `0`. So it reverts.

Hence the exit flow of all users get broken.

Recommendation: In the `FeeDispatcher._setFeeSplit` function validate that the split of a recipient must not be equal to `0`.

```
  for (uint256 i = 0; i < recipients.length; i++) {
      uint256 split = splits[i];
+     if (split == 0) revert();
      sum += split;
      $feeSplits.toUintA().push(split);
      $feeRecipients.toAddressA().push(recipients[i]);
  }
```

Kiln: Fixed in PR 135.

Spearbit: Fixed. A check was added in commit be2a77b0 of PR 135.

5.2.5 Users are favored instead of the protocol within the `MultiPool20._sendToExitQueue` function

Severity: Medium Risk

Context: MultiPool20.sol#L203

Description: The rounding direction during calculations should always favor the protocol over its users.

The `MultiPool20._sendToExitQueue` function relies on the `LibShares.previewWithdraw` function internally to compute the number of pool shares to send to the exit queue for a given ETH amount. The `LibShares.previewWithdraw` function rounds up internally while computing the shares. Thus, given an `ethAmount`, a slightly higher than expected `poolShares` will be returned and passed to the `_sendSharesToExitQueue` function.

As a result, more pool shares than expected will exit the protocol, sending more assets than expected to the users, which goes against the good practice of always favoring the protocol instead of the users.

Recommendation: Consider rounding down when computing the `poolShares` within the `MultiPool20._sendToExitQueue` function to ensure that the protocol is favored instead of the users.

Kiln: Fixed in PR 144.

Spearbit: Fixed in commit e1624458 of the PR 144.

5.2.6 `vFactory.storage_migration.migrate`: `$publicKeyRegistry` does not contain the migrated validators

Severity: Medium Risk

Context: Factory.storage_migration.sol#L298

Description: `vFactory.storage_migration.migrate` performs an "off-code" insertion of the current deposits (deposits made on mainnet) to the new data structures introduced in the 2.1 version.The term "off-code" means that the newly introduced `vFactory.depositFromRoot` function is not used , but rather, the migration logic skips it and amends the state directly.

`$publicKeyRegistry` is a storage variable inside the `vFactory` contract that is introduced in the 2.1 version for two main purposes; mapping between a public key and its corresponding internal id, and preventing deposits to an already activated validator.

The issue here is that the migration contract does not insert the migrated validators into the `$publicKeyRegistry` and thus `vFactory.validatorId` will return 0 for the migrated public keys instead of their real validator id, and in addition, it will be possible to register the same public key again if the merkle root allows it.

Recommendation: The public key hash of each migrated validator should be added to the `$publicKeyRegistry` during the migration script.

Kiln: This issue was fixed with the final migrations commit that has been merged to `vsuite-2-1`, i.e. commit f8bda10e.

Spearbit: Fixed in commit f8bda10e by implementing the auditor's recommendation.

5.2.7 `vOracleAggregator`: Reports can be processed before reaching quorum

Severity: Medium Risk

Context: vOracleAggregator.sol#L222

Description: When an address is both a `member` and `globalOracleMember` its vote count is considered as `2` irrespective of the `globalMemberEjectionStatus` flag. Since the `globalMemberEjectionStatus` is used to determine the `quorum`, the double count mechanism can cause premature pool report (without reaching quorum).

Consider this scenario:

- Member count is 5 (one of these member is `globalMember` as well).
- `globalMemberEjectionStatus` is set to true.
- Now during reporting, the `quorum` value will be `4`.
- In this case if `globalMember` performs a report then quorum will be reached by just 3 member votes (1 member + 1 member + 2 globalMember).

Hence report will be processed before actually reaching quorum.

Recommendation:

```
- uint256 voteCount = (isMember ? 1 : 0) + (isGlobalMember ? 1 : 0);
+ uint256 voteCount = (isMember ? 1 : 0) + (isGlobalMember && !globalMemberEjectionStatus ? 1 : 0);
```

Kiln: Fixed in PR 141.

Spearbit: Fixed. Vote count is adjusted as suggested in PR 141's commit a7213786.

5.2.8 Updating integrator's fee recipients without harvesting accrued fees will prevent previous recipients to receive the owed part of the commissions

Severity: Medium Risk

Context: MultiPool.sol#L168-L170

Description: The `FeeDispatcher._setFeeSplit` allows the integrator's `admin` to update the list of recipients and their corresponding fee split. The internal logic of such a function updates those values without checking if the old recipients have already accrued commissions in form of ETH or `vPool` shares.

Without harvesting and distributing the accrued commissions, the old receivers could receive less, more or no commissions compared to what should be really owed to them.

Let's assume that we have

- [r1: 10%, r2: 90%].
- The contract has collected 10 ETH of fees (when users stakes their ETH, and their ETH are "purchased back".

If `withdrawCommission` is not executed before `_setFeeSplit` some of the old recipients won't receive their deserved ETH accrued as fee.

- Scenario 1): `changeSplit(recipients[r2, r3], splits=[90, 10])`.
- Scenario 2): `changeSplit(recipients[r12, r2], splits=[5, 95])`.

Let's say that now someone calls `withdrawCommission`:

- Scenario 1): r1 has been removed and will not receive 1 ETH that it deserves, that ETH will be received by r3.
- Scenario 2): r1 will receive 0.5 ETH less, r2 will receive 0.5 ETH more.

The same example can be applied to the vPool shares that would be harvested and sent via the exitCommissionShares.

Recommendation: Kiln should execute withdrawCommission and exitCommissionShares (for each pool) before the execution of changeSplit.

Kiln: Thanks for this issue, it was resolved by PR 136.

Spearbit: The recommendations have been implemented in commit 92d06f64 of PR 136.

5.2.9 Integrator's fee receivers that can't receive ETH or is address(0) can disrupt the integrator's contract

Severity: Medium Risk

Context: FeeDispatcher.sol#L118, FeeDispatcher.sol#L74-L78, NFT.sol#L246-L248

Description: The list of recipients in the FeeDispatcher' contract are the recipients that will receive the commissions earned during the usage of the integrator's contract.

The fees are harvested in three different parts of the integrator's flows.

- Automatically in the form of ETH when part of the staker's ETH is bought back in exchange for vPool shares.
- Automatically in the form of vPool shares when the users exit their position.
- Manually in the form of vPool shares when exitCommissionShares is triggered by the admin.

The list of recipients and the relative fee split (that each recipient will get from the total amount) are configured in the FeeDispatcher._setFeeSplit function. Such function will revert only if the input's arrays are not of the same length or if the sum of the splits are not equal to 100%, but there are **no** sanity checks on the specific element value.

A recipient that can't receive ETH or the vPool share could disrupt the integrator's logic in the following ways:

- If the recipient can't receive ETH (low-level call fails) when FeeDispatcher.withdrawCommission is executed could prevent the update of the recipients/split (see the recommendations provided in the issue "Updating integrator's fee recipients without harvesting accrued fees will prevent previous recipients to receive the owed part of the commissions").
- If the recipient is address(0) it will break the user's exit process via the forbit() or requestExit(...) flow or the exitCommissionShares flow. When the recipient is address(0) the shares sent to the vExitQueue will try to mint an NFT ticket with the address(0) as the owner. The _mint underlying logic will revert when this happens.

Recommendation: Kiln should perform additional sanity checks on each recipients item when _setFeeSplit is executed:

- The recipient must not be address(0).
- The recipient should be able to receive ETH without reverting.

Kiln: Thanks for this issue, it was resolved by PR 135.

Spearbit: The recommendations have been implemented in commit be2a77b0 of PR 135.

Note: one of the recipient could break itself on purpose and changeSplit (that is the only way to remove a broken recipient) will revert because now it (correctly) distributes owed fees to the old recipient before changing them.

If an existing recipient breaks (on purpose or mistakenly) the system will be in a broken loop that can't be fixed (it can but with an ad hoc upgrade that forces to remove such recipient).

Kiln: I think this is ok, as it's quite improbable that the recipient would even be able to break (impossible for EOAs and most smart accounts, would need a custom smart contract made to revert when a flag is set), and there is no incentive for the beneficiaries to brick the system.

5.2.10 `AccountList._verifyApplyRightsAuthorization` can be executed infinite times until the deadline has expired

Severity: Medium Risk

Context: AccountList.sol#L131-L163, MultiPool.sol#L198-L200, Native20.sol#L91-L95

Description: The current implementation of `AccontList._verifyApplyRightsAuthorization` is used by the `MultiPool` contract to allow anyone to apply rights to an `account` in a permissionless way via a signature. The function is also used indirectly during the `stake` flow provided by `Native20.authorizeAndStake`.

The problem is that the current logic of `_verifyApplyRightsAuthorization` does not bound the `signature` to any account `nonce`, allowing such a signature to be used infinite times until the `deadline` has expired.

This means that even if the signature has been already used and the admin had updated the authorization rights of the `account`, anyone can override the last admin operation by re-executing the `authorize(uint248 rights, address account, uint256 expiration, uint8 v, bytes32 r, bytes32 s)` with the same signature already used.

Recommendation: Kiln should:

1) Add the support to `nonces` to make the signature unique and usable only once for a specific `account`.
2) Bonus: allow the `admin` or `authrorizer` to increase the user's nonce to invalidate an existing signature (not yet used).

Kiln: Resolved in PR 145.

Spearbit: The recommendations have been implemented in commit 43121c9c of PR 145.

5.2.11 Execution Layer failed deposits are incorrectly accounted in the `vPool` total underlying supply

Severity: Medium Risk

Context: vPool.internals.sol#L402-L404

Description: The `vPool` purchase validators and deposit ETH to the `vFactory` the operation is registered by the L1 Deposit contract and after XYZ amount of time the Deposit will be processed on the CL. Once the deposit has been processed, the validator is activated and will be accounted by the Oracles in their report.

During the time between EL deposit and CL activation, Kiln is accounting for those 32 ETH not-yet-activated and reported by performing this logic.

The problem is that there's an edge case where the Deposit on EL will "revert silently" (without any accounting). This happens if the validator is new (not yet in the CL validators list) and `BLS` verification returns false.

Please see the Beacon chain spec for Deposits. In specific the `apply_deposit` function where the `bls.Verify(pubkey, signing_root, signature)` returns false. In this case, the 32 ETH deposited by the `vFactory` will be lost forever without any way to recover them.

Those 32 ETH won't be accounted by the Oracle records as a lost because they have never been tracked as active balance and the same happens for the validator, but the `vPool` will always report that such validator has indeed provided 32 ETH, and they should be accounted.

I still need time to investigate on which are the side effects of this issue, but I would say that for sure it messes up the `vPool` and `vFactory` variables.

On `vFactory`:

- `$fundedValidators` is reporting +1 validator that should not be there (`vPool` can't exit such validator).
- `$publicKeyRegistry.get()[validatorPublichKeyHash]` is marked as used and can't be re-used anymore, but such a validator has not been activated in reality.
- `$deposits.get()[validatorId]` has been created for a `validatorID` that will never be activated.

On `vPoolReporting.driver`:

- `__.traces.preUnderlyingSupply` will be inflated, as a consequence:
  - `__.traces.increaseLimit` will be inflated → more funds will be pulled from the CL Recipient/EL Recipient/Exit Queue/Coverage Recipient → share rate can increase more than it should.
  - `__.traces.coverageIncreaseLimit` will be inflated → more funds than expected can be pulled from the Coverage Recipient → share rate can increase more than it should.
  - `__.traces.decreaseLimit` will be bigger than expected → this influence the `delta` check performed to revert in case the pull loss (after pulling rewards/coverage) is still too high compared to the upper bound (of loss) expected → `vPool` will accept a loss too high compared to what it should → share rate will decrease more than expected.
- `__.traces.postUnderlyingSuppl` will be inflated, as a consequence:
  - `postUnderlyingSupplyIncludingExitAllocation` will be inflated → `__.exitDemand` will be greater than it should be → this will influence the whole exit logic inside the `report(...)` function.

On `vPool.internal`:

- `_distributeRewards` will mint fewer shares to the Operator Treasury → operator earn fewer fees and Coverage Recipient will earn fewer shares (to be used later on to fulfill the `vPool` losses)

`vPool` contract:

- In the `deposit` function, the `_totalUnderlyingSupply()` is used to calculate how many shares will be minted for the deposited amount → inflated value of `_totalUnderlyingSupply()` will mint fewer shares for the end user.

On `Multipool`:

- `_getPoolInfo()` will return an inflated value for the `totalUnderlyingSupply` struct attribute that will influence all the calculations that are based on such value:
  - `Native20.totalUnderlyingSupply()`
  - `Multipool20.rate()`
  - ... many more...

The inflated `_totalUnderlyingSupply()` will influence numerous logic.

On various Getters

There are other getters function that rely on the value returned by `vPool._totalUnderlyingSupply()`. Those getters will return an invalid value.

Note: The report is referencing only the `vPool` because this is (in the current codebase state) the only "app" that interacts with the `vFactory` right now. This is not specific to the `vPool` itself.

Recommendation: Kiln should implement an authed function that accounts for the "failed deposit" of new validators (if the validator already exists, the balance gets increased without problem). This can be implemented in different ways, updating directly the `$validators` state variable or creating a new one that needs to be accounted in the `vPool._totalUnderlyingSupply` function's logic.

Because this failure responsibility is on the Node Operator, the "failed deposit" loss should also be accounted in the funds that are pulled from the Coverage Recipient.

Kiln: Resolved by PR 148.

Spearbit: The recommendations have been implemented in the PR 148 with more natspec coverage in commit 8af5a0e8.

5.3 Low Risk

5.3.1 `vOracleAggregator` could allow the `globalOracleMember` to submit a report without reaching a "real" quorum

Severity: Low Risk

Context: vOracleAggregator.sol#L222

Description: `vOracleAggregator` is a crucial part of the whole protocol because it's responsible to reach a quorum on each Consensus Layer report that will influence everything around the `vPool`.

Two actors can submit and vote for a report:

- Oracle members managed by the `vFactory` admin (a role owned by the node operator itself).
- Oracle global member that is dynamically fetched from the `Nexus` contract, owned by Kiln itself.

The number of required votes to reach the quorum is based on the current number of members and the state of the `$globalMemberEjectionStatus` flag:

```
function _quorum() internal view returns (uint256) {
    uint256 memberCount = $members.toAddressA().length;
    // unless we have 5+ members AND the global member ejection status is true, we account for the
↪   global oracle member
    if (memberCount < 5 || !$globalMemberEjectionStatus.get()) {
        ++memberCount;
    }
    // quorum = (memberCount * 75%) + 1
    return ((memberCount * 3) >> 2) + 1;
}
```

Let's assume that we are in a clean state without any members and `Alice` as the `Nexus`'s `globalOracleMember`. In this state, any submission performed by the `globalOracleMember` would revert because of the `oracleReady` modifier check (at least one member is needed).

The `vFactory` admin add the `globalOracleMember` itself as a "normal" member. Because the now we have a member, the `globalOracleMember` is allowed to submit a report. Given the current state, the `_quorum()` function will return `2` as the number of votes (not members) need to reach the quorum. The value is calculated a `((2 * 3) / 4) + 1` that returns `2` (given the rounding down division).

Because just two votes are needed, and because the current `submitReport` allows to "double count" the `globalOracleMember` vote as double if it's also a "normal" member, the report submitted by such user will reach the quorum and reported directly to the `vPool`.

This means that a single, unique, user was allowed to submit a report that has not been verified and validated by any other users.

Recommendation: Kiln should not allow this scenario and in general prevent that the `globalOracleMember` can be added or contained as a "normal" oracle member.

Kiln: Fixed in PR 141.

Spearbit: The commit e0da4736 of PR 141 will return `_ready() == false` when the global oracle member is also the only "normal" member.

The remaining part of the recommendations has not been implemented.

5.3.2 Missing validation check in `Multipool20.setPoolActivation`

Severity: Low Risk

Context: MultiPool20.sol#L80

Description: The `Multipool20.setPoolActivation` function does not validate the `poolId` input parameter. Due to which the `$poolActivation` status of an arbitrary `poolId` can be set by the admin.

Recommendation: Consider adding a sanity check like this:

```
  function setPoolActivation(uint256 poolId, bool status, uint256[] calldata newPoolPercentages)
↪  external onlyAdmin {
+     if (poolId >= $poolCount.get()) revert InvalidPoolId(poolId);
      $poolActivation.get()[poolId] = status.v();
      _setPoolPercentages(newPoolPercentages);
  }
```

Kiln: Fixed in PR 132.

Spearbit: Verified that commit e76dceb1 of PR 132 fixes the issue.

5.3.3 Key verification after `vFactory` migration is incomplete

Severity: Low Risk

Context: Migration.base.t.sol#L435

Description: At the end of the vFactory migration, the code will verify that all the keys have been migrated successfully via the following code. However, the issue is that the `idx` is incremented by 100 instead of 1 after each loop. As a result, the verification will fail to detect any key migration failure, as most of the keys were not checked.

```
for (uint256 idx = 0; idx < totalKeys; idx += 100) {
    OldDepositCache memory oldCache = oldDepositCache[address(factory)][idx];
    vFactory_2_1_Like.Deposit_2_1 memory d = vFactory_2_1_Like(address(factory)).deposit(idx + 1);
    if (oldCache.channel != bytes32(0)) {
        expect(d.recipientSalt).toEqual(bytes32(0), "recipientSalt mismatch");
    } else {
        expect(d.recipientSalt).toEqual(keccak256(oldCache.publicKey), "recipientSalt mismatch");
    }
    expect(d.owner).toEqual(oldCache.owner, "owner mismatch");
    expect(d.feeRecipient).toEqual(oldCache.feeRecipient, "feeRecipient mismatch");
}
```

Recommendation: The index should increment by 1 instead of 100:

```
- for (uint256 idx = 0; idx < totalKeys; idx += 100) {
+ for (uint256 idx = 0; idx < totalKeys; ++idx) {
```

Kiln: Fixed in PR 141.

Spearbit: Fixed in commit 769c3b0c of the PR 141.

5.3.4 `RIGHTS__TRANSFER` right not checked within the `Native20.stakeFor` function

Severity: Low Risk

Context: Native20.sol#L83, MultiPool20.sol#L216,

Description: The specification states that for the `Native20.stakeFor()` function:

1. Caller and recipient must not have `RIGHTS__FORBIDDEN` active.
2. Caller must have `RIGHTS__STAKE` active.
3. Caller and recipient must have `RIGHTS__TRANSFER` active.

It was observed that the `RIGHTS__TRANSFER` right of the caller and recipient (3rd requirement) was not checked within the `Native20.stakeFor` and `MultiPool20._stake` functions.

Recommendation: Consider checking the `RIGHTS__TRANSFER` right of the caller and recipient within the `Native20.stakeFor` and `MultiPool20._stake` functions to align with the specification.

Kiln: Fixed in PR 134.

Spearbit: Fixed in commit d5000587 of PR 134.

5.3.5 Authorize function does not support EIP-1271

Severity: Low Risk

Context: MultiPool.sol#L428

Description: The `MultiPool.authorize` and `Native20.authorizeAndStake` functions call the `MultiPool._checkAndApplyAuthorization` function internally to verify that a signed authorization has been performed by the admin or the authorizer and apply the rights.

The signature verification within the `MultiPool._checkAndApplyAuthorization` function depends on the native `ecrecover` and does not support EIP-1271. As a result, only EOA can be the admin or authorizer.

If an admin or authorizer's address maps to a smart contract wallet (e.g., Gnosis Safe), they will not be able to execute these functions, as the signature verification will fail.

Recommendation: Review if there will be a case where the admin or authorizer's address maps to a smart contract wallet within the protocol. If yes, consider implementing EIP-1271 so that the smart contract wallet's signatures can be verified within the protocol.

Kiln: Kiln acknowledges this issue with the following context: the feature was mainly made for an EOA running in the integrator's backend to provide signatures on demand, if more security is needed for those authorizations they can be made through an MPC solution to generate a valid secp256k1 signatures which would eliminate the need to support EIP-1271.

Spearbit: Acknowledged.

5.3.6 `vOracleAggregator` voting state should be reset when the Global Member has already voted and `$globalMemberEjectionStatus` is turned to `true`

Severity: Low Risk

Context: vOracleAggregator.sol#L172-L178, vOracleAggregator.sol#L199-L202

Description: When the `vOracleAggregator` contract has more than 5 members and the `$globalMemberEjectionStatus == true`, the vote of the Global Oracle Member (unless it has been added as a "normal" oracle member) is rejected.

Given such logic, if the Global Oracle Member has already cast a vote and the `$globalMemberEjectionStatus` flag is switched from `false` to `true`, the `setGlobalMemberEjectionStatus` should execute the `_clear()` internal function resetting all the vote related state variables.

Recommendation: Kiln should reset all the state variables related to the voting system when the Global Oracle Member has already cast a vote and the `$globalMemberEjectionStatus` state variable is changed from `false` to `true`.

Spearbit: The recommendations have been implemented in the commit f6f39bfe of the PR 141.

Additionally, there is a small typo that needs to be fixed: f6f39bfe diff 485a6f98: `we'r` → `we're`.

5.3.7 Votes in `vTreasury` and `vOracleAggregator` should be resetted if the corresponding member in the `Nexus` is updated

Severity: Low Risk

Context: Nexus.sol#L258-L260, Nexus.sol#L263-L265

Description: The Nexus `$globalRecipient` and `$globalOracle` state variables can be changed at any time by the Nexus's `admin`. Those global members are then used by each of `vTreasury` and `vOracleAggregator` instances.

When the address of these members changes and the old member has already cast a vote in one of the instance of the `vTreasury` or `vOracleAggregator` contract, such vote should be voided and reset.

An additional logic that should be applied in `vOracleAggregator` when the `$globalOracle` is changed on the `Nexus` is to check if the old `$globalOracle` was also part of the `vOracleAggregator.$members` mapping. If that's the case, such member should be removed and his votes voided.

Recommendation: Kiln should implement a logic to void the vote of the old member on `vTreasury` or `vOracleAggregator` contract when such member has changed on the `Nexus` contract. Another suggestion is to remove the previous `$globalOracle` from the `vOracleAggregator.$members` mapping if it was previously added.

Kiln: We decided to partially fix this issue, we're adding the clear of votes when the operator changes in the `vTreasury`. We decided to acknowledge the change in `globalOracle` or `globalRecipient` and not clear votes. In both cases a malicious actor would be deemed malicious is voting invalid data, and the quorums are here to prevent unilateral damage:

- If the oracle is not voting properly, and other members have a different report, then impact of malicious global oracle member is limited.
- If the global recipient votes for an insanely high fee in the `vTreasury`, the operator still has to vote the same value and explicitly agree on the commission.

Spearbit: Kiln has implemented the vote resetting in the `vTreasury` contract when the operator is changed inside the commit 4b6c7d41 of PR 141.

The rest of the recommendations have been acknowledged by the client.

5.3.8 Pool's commission fee is not validated against `$maxCommission` when `addPool` is executed

Severity: Low Risk

Context: MultiPool.sol#L397

Description: When a new `vPool` is added to the `MultiPool` batch, the pool's fee is only sanity checked by executing `LibBasisPoints.check(fee);` but is not validated against the `$maxCommission` upper bound.

This check is instead performed inside `_setFee` when `changeFee(uint256 poolId, uint256 newFeeBps)` is executed. Because of this lack of validation, a freshly added pool could end up with a commission fee that is higher compared to the upper bound defined by the `$maxCommission`.

Recommendation: This is an example of pseudo of code that Kiln could take inspiration from to solve the issue

```
  function _addPool(address newPool, uint256 fee) internal {
-     LibBasisPoints.check(fee);
      LibSanitize.notZeroAddress(newPool);

      uint256 poolId = $poolCount.get();
      for (uint256 i = 0; i < poolId;) {
          if (newPool == $poolMap.get()[i].toAddress()) {
              revert PoolAlreadyRegistered(newPool);
          }
          unchecked {
              i++;
          }
      }

      $poolMap.get()[poolId] = newPool.v();
-     $fees.get()[poolId] = fee;
      $poolActivation.get()[poolId] = true.v();
      $poolCount.set(poolId + 1);

      emit PoolAdded(newPool, poolId);
-     emit SetFee(poolId, fee);
+     _setFee(fee, poolId)
  }
```

Note that `LibBasisPoints.check(fee);` is not needed anymore because `$maxCommission` cannot be above 100% given the checks performed inside `_setMaxCommission`.

Kiln: Thanks for this issue, it was resolved by PR 129.

Spearbit: The recommendations have been implemented in commit b8be3aac of PR 129.

The `_setMaxCommission(args.maxCommissionBps);` in `Native20.initialize` has been correctly moved before the execution of `_addPool` to have the correct value of `$maxCommission` set before the pool's fee validation.

5.3.9 OFAC sanction list restriction should be applied to integrator's commission recipients

Severity: Low Risk

Context: FeeDispatcher.sol#L64-L93, MultiPool.sol#L334-L354,

Description: Integrator's `$feeRecipients` should be treated as normal users and should receive the same treatment. For this reason, they should not be able to exit the integrator's pool if they are included in the OFAC sanction list.

The `MultiPool._exitCommissionShares` and `FeeDispatcher.withdrawCommission` should not allow the exit of `vPool` shares or `ETH` if one of the recipients is included in the OFAC list.

Recommendation: Kiln should prevent **any** OFAC user (normal user or fee recipients) to be able to move funds into or out of the integrator's pool. The fix to the issue is complex and OOS of the report, but Kiln should be aware that:

1) The `MultiPool._exitCommissionShares` performed during the user's exit should not disrupt the user's operation (revert)
2) The change needed to `_setFeeSplit` will need a more complex logic given the suggestion provided in the issue "Updating integrator's fee recipients without harvesting accrued fees will prevent previous recipients to receive the owed part of the commissions".

Kiln: Acknowledged. The sanctions verification are meant for the users not the addresses that belong to known entities such as the admin or fee recipients.

The recipient addresses are not used to interact with protocols on the network and have no reason to be sanctioned anyway, on top of that as you noted this would create a DoS vector for in `_exitCommissionShares`, thus possibly

prevent requesting exits in some edge cases.

Spearbit: Acknowledged.

5.3.10 An OFAC sanctioned user's position can be force-exited by an `admin` via the executing of `forbid(...)`

Severity: Low Risk

Context: MultiPool20.sol#L102

Description: From the vsuite 2.1 PR specs about "Add ofac list checker":

> The integrations contracts will now query the ofac list contract of chainalysis for sanctioned addresses. The contract can set its ofac list address once and is only able to activate / deactivate its usage at will, but not able to edit the list address. Sanctioned users cannot stake, unstake or transfer funds

By reading the specs it appear clear that a sanctioned user should not be able to exit the integrator's pool and receive ETH. The current logic of `MultiPool.forbid` is instead allowing the `MultiPool` admin to "force exit" a user

```
function forbid(address account) external onlyAdmin {
    _applyRights(uint248(MultiPool.RIGHTS__FORBIDDEN), account);
    uint256 balance = $balances.get()[account];
    if (balance > 0) {
        _performRequestExit(account, balance, _getPoolInfos());
    }
}
```

Recommendation: If the user is sanctioned, the `forbid` function should revert or only be able to execute `_applyRights(uint248(MultiPool.RIGHTS__FORBIDDEN), account)`.

The `IMultiPool.forbid` natspec documentation should be updated accordingly. The current natspec `/// @dev If the user has any balance, it will be sent to the exit queue automatically` should specify that the balance will be exited only if the user is not OFAC sanctioned.

Kiln: Thanks for this issue, it was resolved by PR 128.

Spearbit: The recommendations have been implemented in commit ba77dce6 of PR 128.

Note that a fix for an issue found on the new changes to the sanction list has been provided in commit ba77dce6.

5.3.11 `Native20.authorizeAndStake` allows an OFAC sanctioned `msg.sender` to perform stake ETH

Severity: Low Risk

Context: Native20.sol#L91-L95

Description: The `authorizeAndStake` stake in the `Native20` contract allows the `msg.sender` to bypass the `revertIfSanctioned` check performed in `stake` and `stakeFor`. As a consequence, an OFAC sanctioned user will be enabled to perform a stake operation.

Recommendation: Add the `revertIfSanctioned(msg.sender);` check to the function's logic.

Kiln: Thanks for this issue, it was resolved by PR 131.

Spearbit: The recommendations have been implemented in commit 7b98b2f1 of PR 131.

5.3.12 Integrator and `vPool` mint logic when supply is below min-supply-threshold are different

Severity: Low Risk

Context: MultiPool20.sol#L240-L249, vPool.sol#L220-L228

Description: There's a difference between how the `vPool.deposit` logic mints new tokens when the current supply is below an "initial mint threshold".

In `vPool` the contract will mint ETH/tokens 1:1 up to reaching `INITIAL_MINT_THRESHOLD`. After that, if there's still ETH left to be deposited, the number of tokens to be minted will be given by the pool's ratio `MultiPool20` instead, if the supply is below the threshold, will mint the whole ETH amount staked 1:1.

Recommendation: Kiln should decide which of the two behaviors should be adopted and apply the same one to both the contracts' logic.

Kiln: Thanks for this issue, Kiln acknowledges it with the following context: this issue will be tackled in the next update along with possible changes to the minting logic.

Spearbit: Acknowledged.

5.3.13 OFAC Sanctions List should be hard-coded as a constant value

Severity: Low Risk

Context: MultiPool.sol#L207-L219

Description: Following the vsuite 2.1 specification PR related to "Add OFAC list checker", the integrator's contract should be allowed to use only the Chainalysis sanctions oracle as the source of the OFAC list.

Recommendation: Kiln should:

1) Store the Chainalysis sanctions oracle address as a `constant` variable.
2) Refactor the `$sanctionsData` data structure. Now it just needs to store the `shouldCheckSanctions` boolean.
3) Refactor the logic that uses `$sanctionsData`, it now only needs to check if the OFAC feature has been enabled. The address to be used to query the oracle will always be equal to the Chainalysis one.

Kiln: Thanks for this issue, it was resolved by PR 147.

Spearbit: The recommendations have been implemented in the commit 88582cba of PR 147.

5.3.14 `Liquid20A` and `Multipool20 transferFrom` allowance decrease does not adhere to OpenZeppelin standard and `vPool` logic

Severity: Low Risk

Context: Liquid20A.sol#L96, MultiPool20.sol#L377-L379, vPool.internals.sol#L248-L254

Description: In both the OpenZeppelin ERC20 implementation and the `vPool` implementation, when tokens are transferred via the `transferFrom` flow, the spender's allowance is decreased only if the `currentAllowance != type(uint256).max`.

The current logic of `Liquid20A` and `Multipool20 transferFrom` will instead always reduce the spender's allowance.

Recommendation: Kiln should reduce the spender's allowance only if it's not equal to `type(uint256).max`.

Kiln: Thanks for this issue, it was resolved by PR 143.

Spearbit: The recommendations have been implemented in commit dcc9ddb3 of PR 143.

5.4 Gas Optimization

5.4.1 `vPool.deposit`: `if (msgValue > type(uint128).max)` can be removed

Severity: Gas Optimization

Context: vPool.sol#L212

Description: The `vPool.deposit` function implements a check to prevent an overflow in the `$ethers.get().deposited` variable:

```
if (msgValue > type(uint128).max || msgValue > type(uint128).max - deposited) {
    revert DepositTooLarge(msgValue, deposited);
}
```

As we can see, the left part of the clause - `msgValue > type(uint128).max` is redundant since it is included in the right part of the clause because the operator `||` is used.

Recommendation: Consider removing the left part of the clause.

Kiln: Resolved by PR 141.

Spearbit: Fixed in commit 8488a6d1 by implementing the auditor's recommendation.

5.4.2 Type hash can be a constant

Severity: Gas Optimization

Context: AccountList.sol#L152

Description: The type hash is re-computed in every execution even though the value remains the same.

```
bytes32 hashStruct =
            keccak256(abi.encode(keccak256("ApplyRights(uint248 rights,address account,uint256
↪  expiration)"), rights, account, expiration));
```

Recommendation: Consider defining a constant for the type hash `(keccak256("ApplyRights(uint248 rights,address account,uint256 expiration))` instead of re-computing it in every execution for gas efficiency.

Kiln: Kiln acknowledges this issue with the following context: after trying to apply the suggested changes we saw essentially no gas savings and the code was a bit less clean so we decided to not apply a fix to this issue.

Spearbit: Acknowledged.

5.4.3 Consider using `poolInfos.length` instead of `$poolCount.get()` when possible

Severity: Gas Optimization

Context: MultiPool20.sol#L135, MultiPool20.sol#L230, MultiPool20.sol#L455

Description: Given that `poolInfos` comes from `_getPoolInfos()`, the functions referenced in the context section could avoid loading the `$poolCount.get()` information and use directly `poolInfos.length`.

This should be done **only** if `poolInfos` comes from a "trusted" source and the pool structure has not been modified after retrieving the pool list from storage.

Recommendation: Consider applying the suggested gas optimization, avoiding an additional SLOAD when possible.

Kiln: Thanks for this issue, it was resolved by PR 139.

Spearbit: The recommendations have been implemented in commit 69f6a495 of PR 139.

5.4.4 `MultiPool.revertIfOneIsSanctioned` can be refactored to save gas

Severity: Gas Optimization

Context: MultiPool.sol#L566, MultiPool.sol#L570

Description: The `MultiPool.revertIfOneIsSanctioned` logic can be refactored to save some gas:

1) The `if (address(_sanctionsList) != address(0))` check can be removed. The `sanctions.listAddress` is already checked by the previous `if` statement.
2) If `user1 == user2`, `_sanctionsList.isSanctioned(user2)` can be avoided given that we have already checked the same user when `_sanctionsList.isSanctioned(user1)` has been executed a few lines above.

Recommendation: Kiln should consider refactoring the function to save gas.

Kiln: Thanks for this issue, it was resolved by PR 133.

Spearbit: The recommendations have been implemented in commit cf52a95b of PR 133.

5.4.5 `vPool.purchaseValidators` should revert as early as possible if there's not enough funds to purchase the requested validators

Severity: Gas Optimization

Context: vPool.sol#L244

Description: When `$$.validationKeys.length > purchaseAmount` the `vPool.purchaseValidators` should revert as early as possible. When we're in this case, it means that `vPool` is trying to purchase more validators than it can afford (given the `commited` ethers amount).

`vPool` does not have a logic to slim down the `$$.validationKeys` to `purchaseAmount` so it will revert as soon as `vFactory` executes the `if (msg.value != LibConstant.DEPOSIT_SIZE * validationKeysLength)` check.

Recommendation: If the `vPool` does not own enough balance to purchase all the requested validators, it should revert as early as possible.

Kiln: Resolved by PR 141.

Spearbit: The recommendations have been implemented in commit 73633e87 of PR 141.

Kiln should be aware that new revert custom errors have been introduced, and the new logic will revert early and won't throe the `vFactory` error `InvalidMessageValue`. With this in mind, Kiln and other integrators should remember to update their dApps and monitoring tools.

5.5 Informational

5.5.1 Consider using the `SafeCast` library for unsafe castings

Severity: Informational

Context: vPoolReporting.driver.sol#L218-L219, vPoolReporting.driver.sol#L235, vPoolReporting.driver.sol#L240, vPoolReporting.driver.sol#L246-L250, vPoolReporting.driver.sol#L278, vPoolReporting.driver.sol#L299-L301, vPoolReporting.driver.sol#L318-L319, vPoolReporting.driver.sol#L342, vPoolReporting.driver.sol#L350-L351, vPoolReporting.driver.sol#L363-L388, vPoolReporting.driver.sol#L409-L414

Description: The codebase contains multiple instances of unsafe casting operations that might result in a practical underflow in certain scenarios (although unlikely) which will lead to data corruption and therefore wrong execution and therefore wrong state transitions.

In the current version of the code we were not able to find any exploitable scenarios while taking to account the inherent trust assumptions of the system, nevertheless, we recommend to make sure the code reverts for these unwanted scenarios to ensure better hardening of the system.

Recommendation: Consider using the `SafeCast` library for the mentioned instances. Keep in mind it is not an exhaustive list and there are more instances that were not mentioned.

Kiln: Resolved by PR 141.

Spearbit: Fixed in 9ded0459 by implementing the auditor's recommendation.

5.5.2 `vPool $lastReport` should be initialized during the `vPool` initialization to avoid having possible huge values during the first report

Severity: Informational

Context: vPoolReporting.driver.sol#L232

Description: When the `vPool` is deployed and initialized, the `vPool $lastReport`, that represents the latest oracle's report, is left uninitialized.

When the very first oracle report is submitted, the `period` local variable of `vPoolReporting.driver report(...)` will result in a high number equal to the difference in seconds between the Consensus Layer genesis time (1 December 2020) and the report's epoch.

The `period` variable is crucial to calculate the `__.traces.increaseLimit`, `__.increaseCredit` and `__.traces.coverageIncreaseLimit` limits:

```
//  ----- The period is computed by computing the timestamps from the epoch values
uint256 period = _epochTimestamp(cls, rprt.epoch) - _epochTimestamp(cls, lastRprt.epoch);
//  ----- The maximum allowed balance increase is now computed
__.traces.increaseLimit = uint128(_maxAllowedBalanceIncrease(__.traces.preUnderlyingSupply, period));
__.increaseCredit = __.traces.increaseLimit;

// [... other code]

__.traces.coverageIncreaseLimit = uint128(_maxCoverageBalanceIncrease(__.traces.preUnderlyingSupply,
↪  period));
```

- `__.traces.increaseLimit` is used to limit the maximum rate increase and establish the initial value of `__.increaseCredit`.
- `__.increaseCredit` is used to limit how much `ETH` will be pulled from EL recipient, Exit Queue unused funds and Coverage Recipient.
- `__.traces.coverageIncreaseLimit` is used to limit how much it can be pulled, on top of the remaining `__.increaseCredit`, from the Coverage Recipient.

Let's assume that the new report is performed with `epoch === 274725` (4 April 2024), `period` will be equal to `105494400` that is equal to ~1221 days. Let's also assume that someone has already staked `1 ETH` into the `vPool` before that the first report is submitted. With the bounds configured like

- `upperBound = 1500 bps`.
- `upperCoverageBound = 500 bps`.
- `lowerBound = 500 bps`.

The limits will be initialized like:

- `__.traces.increaseLimit = 501780821917808219 =~ 0,5 ETH`.
- `__.increaseCredit = 501780821917808219 =~ 0,5 ETH`.
- `__.traces.coverageIncreaseLimit = 501780821917808219 =~ 0,167 ETH`.

This would mean that, theoretically, the `vPool` reporting logic could pull up to `~0.5 ETH` from the EL Recipient/Exit Queue and up to `~0.5 ETH + ~ 0,167 ETH` from the Coverage Recipient.

Given that the pool has just been initialized and assuming that the report has been submitted by an honest oracle with correct information, funds can only be pulled from them `Execution Layer Recipient` that is the only one that allows to receive permissionless donations. `WithdrawalRecipient` can receive funds only from the Consensus Layer, the `ExitQueue` can have unused funds only if there have been exit that have produced exceeding funds

(growth of the rate in the meanwhile) and the Coverage Funds can receive shares/funds only from authorized donors.

The funds pulled from the Execution Layer Recipient, if any, can be seen as donations that will increase the value of existing pool shares.

If before the report, no stakers have staked funds in the `vPool`, those three local values will be equal to `0` even with high `period` value because `__.traces.preUnderlyingSupply` will be equal to zero (no funds staked).

Recommendation: Kiln should consider initializing the `$lastReport.epoch` value when the `vPool` is deployed and initialized to avoid having a very high `period` value during the very first oracle report.

Kiln: Resolved by PR 141.

Spearbit: The recom,endations have been implemented in commit a3c6cd69 of PR 141.

Note that Kiln has not provided yet a logic that handles already deployed `vPool` that have not been correctly initialized.

5.5.3 `vPool` can potentially take commissions while being at a net loss

Severity: Informational

Context: vPoolReporting.driver.sol#L239-L240

Description: The reporting logic in `vPoolReporting` calculates rewards based upon the relative difference between the last report and new report volume. Further, commisions are taken on the relative reward amount.

Due to this design there can be scenarios where a vPool can take commissions while being at a net loss. Here is a scenario:

- Suppose `balanceSum` is `32 ETH` initially.
- On next report `balanceSum` is reported as `30`, this is a loss of `2` ETH so no commission is taken.
- On next report `balanceSum` is reported as `31`, while this is neither skimmed ETH nor execution layer rewards it is still considered as a profit of `1` ETH and a commission is taken.

As discussed with the team, due to the `vPool`'s design there is no easy way to track the absolute values hence this behavior is considered as accepted risk.

Recommendation: Consider omitting commissions when the pool is at a net loss.

Kiln: Acknowledged. For this one, as we have the coverage in place, we deem that there are sufficient mechanisms that the operator can use to prevent such situations. It's up to the operator to properly setup his coverage policy.

In case of a loss due to inactivity, this could happen as well, but we consider that the overhead of managing this case is far greater than the problem we would solve.

In case of some very exceptional loss, where the operator would take a commission when the pool is at loss, it's also up to the Operator to properly redistribute the funds back to the users if the Operator believes that the commission shouldn't be taken.

Spearbit: Acknowledged.

5.5.4 `vPoolInternals._maxAllowedBalanceIncrease,_maxCoverageBalanceIncrease`: calculated values are not precise since a linear proximity is implemented

Severity: Informational

Context: vPoolReporting.driver.sol#L235, vPoolReporting.driver.sol#L342

Description: `vPoolInternals._maxAllowedBalanceIncrease,_maxCoverageBalanceIncrease` are used in the `report` function to determine whether the upper bounds for incoming funds and insurance are crossed. While the current implementation serves its purpose and we don't see any concrete issue around it, it is important to note that the current implementation of both function is only a linear proximity which does not take into account compounding interest and thus is not necessarily representing the accurate value. Let's take `_maxAllowedBalanceIncrease` as an example; this function should return the max allowed increase for a period of time assuming a specific APR denoted as `maxAPRUpperBound`. One example in which this calculation will not be accurate is the case of an interpolation, i.e. calculating the max allowed value for a point in time before the first year ended. To better understand the issue let's consider a simple example:

Assuming 5% APR, balance of 100 ETH, `_maxAllowedBalanceIncrease` will return 2.5 but what should `_maxAllowedBalanceIncrease` return after 6 months?

Let's denote the result in x, such that:

After solving for x:

Recommendation: As mentioned, in the current version of the code the usage of these functions is safe, but the dev team should be aware that these values are approximated and as such should not be considered precise in future versions of the code.

Kiln: Acknowledged. The bounds are mainly security feature to catch obvious reporting errors, and are not meant to be precisely following the expected reward rate of the validators. The current implementation is enough for our needs.

Spearbit: Acknowledged.

5.5.5 `vPoolReportingDriver.report`: Validators will be asked to exit even for tiny amounts of withdrawals in case of exit queue demand is greater than `exitingProjection`

Severity: Informational

Context: vPoolReporting.driver.sol#L567-L568

Description: `vPoolReportingDriver.report` implements two different strategies to optimize unnecessary exits of validators in the system. The first one is to use a portion of users deposits (represented as `exitBoostEthers`) to cover queue withdrawal requests, and the second is to count in the eth that should be received from upcoming exits and delay the queue withdrawal fulfillment to the next report (represented as `exitingProjection`). By the end of the function, in case the exit queue demand is greater than `exitingProjection`, the difference between the demand and the supply is taken as is and is used to determine how many new exit requests should be made. Here is the calculation that is used:

```
uint256 newExitRequests =
LibUint256.ceil(LibUint256.min(__.exitDemand - __.traces.exitingProjection, rprt.maxExitable),
↪  LibConstant.DEPOSIT_SIZE);
```

As you can see, the `ceil` function is used which can lead to un-optimized scenarios. For instance, `newExitRequests` will be 1 even if the difference between `exitDemand` and `exitingProjection` equals 1 wei. So 32 eth will be exited to fulfill only a 1 wei demand.

Recommendation: Consider implementing a threshold mechanism to make sure validators are exited only for large enough amounts of withdrawal queue demand.

Kiln: Acknowledged. I do agree with the issue, but there are two things I took into account before the acknowledged:

- It is very rare to have such low amounts left to trigger a validator exit, and even if it's possible, would the added complexity of yet another variable be worth it?
- What happens if the threshold is above the only requested exit demand for ever? This means it must be in storage and have all the associated getters / setters. I would actually prefer not adding complexity to the current pool reporting design as it's already very complicated.

Spearbit: Acknowledged. It is informational, so it is subjective by nature. You can monitor the real world usage and see if this behavior is common or not.

5.5.6 `vPoolReportingDriver.report`: Using `type(int256).min` as a neutral value for `maxCommittable` will cause the transaction to revert

Severity: Informational

Context: vPoolReporting.driver.sol#L519

Description: Negating the most negative signed integer in solidity for solc versions >= 0.8.0 will result in transaction revert. In the current version of the code, Line 519 negates `rprt.maxCommittable`:

```
uint256 commitment = LibUint256.min(deposited, uint256(-rprt.maxCommittable));
```

The value of `rprt.maxCommittable` is determined off-chain by the oracles which may decide to set it to `type(int256).min` as a neutral "wild-card" value so that the value of `deposited` will be chosen by the `LibUint256.min` function all the time. In that case, the transaction will revert which is not intended.

Recommendation: Make sure that the offchain logic used to determine the value of `rprt.maxCommittable` never defaults to `type(int256).min`.

Kiln: Acknowledged. We've added this issue to our daemon and will add a check to prevent this value from being set.

Spearbit: Acknowledged.

5.5.7 Variables re-namings

Severity: Informational

Context: vPoolReporting.driver.sol#L108, vPoolReporting.driver.sol#L460, vPool.internals.sol#L448

Description: Variable and error names and inline comments may be misleading or outdated in several places in the code, below is a list of names we think should be changed:

1. `ValidatorsReport.exiting`: Consider changing `exiting` to `exitingSum` or any other name that will reflect this variable holds an amount of ETH.
2. `totalFulfillableDemand`: Consider changing `totalFulfillableDemand` to a name which reflects this variable holds the portion of `extraDemand` that can not be covered by `__traces.exitingProjection` and should be covered by `__.traces.exitBoostEthers` instead. In addition, consider changing the comment to reflect it as well.
3. EpochTooOld: `EpochTooOld` is an error raised in case the provided epoch is less than the expected epoch, therefore, the name of the error should be `EpochTooLow` instead.
4. tokensToTransfer: The `_transfer` function is moving tokens but these tokens are actually shares represented as tokens, so consider changing the `tokensToTransfer` to `sharesToTransfer`.

Recommendation: Consider applying the recommended changes for the chosen names.

Kiln: Partially resolved by PR 149.

For `EpochTooOld`, the `epoch` just like a timestamp is a time related variable, so we would keep the current error. For `tokensToTransfer`, the linked method moves integration contract shares (that we prefer calling tokens so we don't have "shares" everywhere) and not `vPool` shares.

Spearbit: Fixed.

5.5.8 `vCoverageRecipient.cover` upper bound on share void is not documented or explained

Severity: Informational

Context: vCoverageRecipient.sol#L118, vPoolReporting.driver.sol#L350

Description: The `vCoverageRecipient.cover(uint256 max)` is called when the `vPool` oracle reports a loss for the pool that must be covered by the Coverage Funds recipient.

```
uint256 _coveredBalanceSum = $coveredBalanceSum.get();
__.traces.coverageIncreaseLimit = uint128(_maxCoverageBalanceIncrease(__.traces.preUnderlyingSupply,
↪  period));
{
    //  ----- We increase the allowed margin for coverage
↪                          //
    uint256 maxPullableCoverage = __.increaseCredit + __.traces.coverageIncreaseLimit;
    //  ----- We only pull funds if there is a difference with the reported slashed balance and the
↪  amount covered by          //
    //        the vPool
↪                          //
    if (maxPullableCoverage > 0 && _coveredBalanceSum < rprt.slashedSum) {
        uint256 maxCoverableAmount = LibUint256.min(rprt.slashedSum - _coveredBalanceSum,
↪  maxPullableCoverage);
        __.traces.pulledCoverageFunds = uint128(_pullCoverage(maxCoverableAmount));
        __.traces.delta += int128(__.traces.pulledCoverageFunds);
    }
}
```

The `maxCoverableAmount` is accumulated loss (accumulated since the previous reports) that has not been covered yet. In theory, all the funds (ETH and shares) owned by the `vCoverageRecipient` should be used to cover such an amount of loss.

When the `vCoverageRecipient.cover` logic will try to cover the loss first with `ETH` and if that's not enough it will calculate the number of shares (owned by the coverage recipient) to void (burn) to cover the remaining amount.

Even if the `vCoverageRecipient` owns all the needed shares to recoup the loss, the logic is upper bounding them to 1% (100 bps) of the total share supply of the `vPool`.

```
function cover(uint256 max) external onlyPool {
    // [... other code]
    if (maxCoverableEther < max && sharesForCoverage > 0) {
        uint256 totalUnderlyingSupply = vpool.totalUnderlyingSupply();
        uint256 totalSupply = vpool.totalSupply();
        uint256 amountToVoid = totalSupply
            - LibUint256.mulDivFloor(totalUnderlyingSupply, totalSupply, (totalUnderlyingSupply + (max
↪  - maxCoverableEther))) - 1;
        uint256 maxRelativeAllowedShares = LibBasisPoints.compute(totalSupply, 100);
        uint256 maxCoverableShares = LibUint256.min(LibUint256.min(sharesForCoverage, amountToVoid),
↪  maxRelativeAllowedShares);
        // [... other code]
    }
}
```

This logic is not explained or documented in any part of the codebase, and is adding another upper bound limit to an amount that is already upper bounded by the `vReportingDriver` itself.

Recommendation: Kiln should document and explain why the `vCoverageRecipient` is imposing a fixed upper bound on the amount of share to be voided to recover from the losses, considering that an upper bound is already placed at the `vPool` level.

If such an upper bound is needed, the upper bound logic should be moved inside the `vPool` itself, alongside all the upper bounds that already exist and are used by the `vPool` when the oracle executes the report logic.

Kiln: Fixed in PR 141.

Spearbit: The reccomendations have been implemented in commit 9322aa4d of PR 141.

5.5.9 Consider implementing a cross-factory check to verify that a validator has not already been "used" by another factory

Severity: Informational

Context: vFactory.sol#L253-L255

Description: When a `vPool` purchases validators from a `vFactory` the purchase requested is validated against the Merkle Tree root (of available validators) and `$publicKeyRegistry` that represents the mapping of validators already purchased within the `vFactory` context.

While this is an edge case because each validator should belong to a single node operator, the `vFactory` contract does not verify that the same validator's public key has already been used by another `vFactory`. The L1 Deposit Contract and Consensus Layer allow depositors to deposit on behalf of the same validator that will result in a top-up of the validator's balance. When this happens, the new `withdrawal credential` sent along with the deposit request will be ignored without updating it.

Without knowing the specifics and logic of the Kiln Oracle that performs the CL reports, we cannot predict the full range of possible side effects. Assuming that the oracle will not report the activation and balance change of the purchase (in case this was the purchase of a validator already activated by another `vFactory`) one side effect would be that the `vPool._totalUnderlyingSupply` will return an invalid value because it will think that a validator that has been purchased, has not yet been reported as active by the oracle

```
uint256 consensusLayerValidatorCount = lastRprt.activatedCount;
uint256 executionLayerPurchasedValidatorCount = $validators.toUintA().length;
if (consensusLayerValidatorCount < executionLayerPurchasedValidatorCount) {
    total += (executionLayerPurchasedValidatorCount - consensusLayerValidatorCount) *
↪   LibConstant.DEPOSIT_SIZE;
}
```

Recommendation: Kiln should consider the implementation of a global mapping `$publicKeyRegistry` of the validator's public key already consumed by all the `vFactory` and revert `vFactory.depositFromRoot` if such public key has been already used.

Kiln: Acknowledged.

For this current audit, we don't want to go too much out of scope with what we're currently bringing to the system. This is a good idea, we've been thinking to use something like this for other use cases (have a central spot that some contracts would be using to improve their security / efficiency)

ex: we were thinking about a central contract where `vPool`s would post their rate after each oracle report. Integration contracts relying on several `vPool`s would be able to make high gas optimizations by only checking if there has been some changes in rate in the related `vPool`s (instead of doing `vPool.totalSupply()` and `vPool.totalUnderlyingSupply()` for every pool each time someone stakes). This would mean that an Integration contract would be easily able to be plugged to a large number of `vPool`, and would cache once per day the rate for all the next stakes.

Maybe it might be the opportunity for us to think about this central coordination logic a bit more, but not for current audit.

Spearbit: Acknowledged.

5.5.10 Outdated Openzeppelin smart contract library is used

Severity: Informational

Context: TUPProxy.sol#L21

Description: The entire codebase uses an old Openzeppelin library version (commit `c22db810` released in Sept 2022).

Recommendation: Consider updating to a recent and stable OZ library version.

Kiln: Openzeppelin contracts were upgraded to `v5.0.2`. Fixed in PR 140.

Spearbit: Openzeppelin was upgraded to `v5.0.2` commit `dbb6104c` in commit 3c2befe0 of PR 140.

5.5.11 Signature verification can be hardened

Severity: Informational

Context: AccountList.sol#L155

Description: Signature verification can be hardened.

Appendix F in the Ethereum Yellow paper, defines the valid range for s in (301):

$$0 < s < \frac{\text{secp256k1n}}{2} + 1$$

and for v in (302):

$$v \in 27, 28$$

1. Verify `s` is within valid bounds to avoid signature malleability:

```
if (uint256(s) > 0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F46681B20A0) {
    revert("ECDSA: invalid signature 's' value");
}
```

Recommendation: Consider using Openzeppelin's ECDSA library for signature verification, which already implemented the required check to ensure that the `s` is within valid bounds to avoid signature malleability by default.

Kiln: Fixed in PR 138.

Spearbit: Fixed in commit 6bdefd52 of the PR 138.

5.5.12 `vTreasury VoteChanged` event should be better documented and refactored

Severity: Informational

Context: vTreasury.sol#L138-L167

Description: When the `operator` or `GlobalRecipient` of the `vTreasury` cast a vote, a `VoteChanged` event is emitted. If the two parties agree on the same `fee` value, the function emits two `VoteChanged`, one for the vote cast, one for the voting system being "resetted" to the default value.

The value that is passed as an input to the `VoteChanged` is not the fee voted itself, but the "representation" of the fee as a vote (the value returned by `_applyVote(fee)`).

Kiln should consider defining two or three different events. One when a `fee` vote is cast, but the parties are not agreed on the vote or one of them has not yet voted. One for when the votes match, and they have agreed on the new `fee` value, and the last one (bonus) to notify that the vote system has been resetted. The latter is optional.

A possible readability improvement (for dApps and monitoring tools) would be to pass directly the `fee` value used for the vote instead of the "vote" value (that includes the "has voted" bit of information). The "has voted" information could be included in the event as additional input parameters.

Recommendation: Kiln should consider refactoring the voting system and event emission, and improve the documentation to explain when those events are emitted and the value included in the event (not the `fee` itself but the `vote` for the `fee` with the "has voted" information included).

Kiln: Partially fixed by PR 141.

The PR only adds documentation to the event. For the other suggestions of adding more events to cover specific cases, as it's already pretty trivial to derive the current state of voting inside offchain indexing, we decide to acknowledge.

Spearbit: Kiln has decided to only document the behavior of the emitted events and the meaning of the vote. The changes made in the commit c92efcb5 of PR 141. can be considered enough.

5.5.13 `vTreasury` will never revert with `TransferError` when `vPool.transferShares` return false

Severity: Informational

Context: vTreasury.sol#L241-L243, vTreasury.sol#L252-L254, vTreasury.sol#L257-L259, vTreasury.sol#L296-L298

Description: The error `TransferError` is thrown in `vTreasury` when the amount of token (shares) transferred is `> 0` and the `IvPool(token).transferShares(...)` execution returns `false`.

The current implementation of `vPool.transferShares` returns the result of the execution of `vPool._transfer(...)`. The internal `_transfer` function is **always** returning `true` or directly `reverting` if something wrong happens during the process.

Because `vPool.transferShares` returns `true` or `revert` directly, the `TransferError` reverts will never be triggered in case of failure.

Recommendation: Kiln should be aware of this "dead code" path and monitor the internal revert errors triggered by the `vPool.transferShares` execution instead of `TransferError`

Kiln: Acknowledged. As returning the bool is standard ERC20, it wouldn't be clean to ignore it, so we decided to cover the `false` case even if it never happens.

Spearbit: Acknowledged.

5.5.14 `vFactory` does not emit `SetValidatorExtraData` event in a coherent way

Severity: Informational

Context: vFactory.sol#L300-L302, vFactory.sol#L471

Description: The `vFactory` contracts emits the `SetValidatorExtraData` event in two different places:

1) During the execution of `depositFromRoot` only if `bytes($$.extradata).length > 0`.
2) During the execution of `setExtraData(uint256[] calldata ids, string calldata extradata)` (is validation passes correctly).

While in `depositFromRoot` the event is emitted only if the user has specified a non-empty `extradata`, inside the `setExtraData` logic, the event is **always** emitted, even if `bytes(extradata).length` is equal to zero.

Recommendation: Kiln should consider using a coherent logic to decide when the `SetValidatorExtraData` event should be emitted. The recommended behavior should be to consider to **always** emit such the event, even when the `extradata` is an empty string.

Kiln: Resolved by PR 141.

Spearbit: The recommendations have been implemented in PR 141. Now, the `SetValidatorExtraData` is always emitted in both `setExtraData` and `depositFromRoot`.

5.5.15 `LibMerkleTree`: Consider removing this library and use the `MerkleProof` library by OZ instead

Severity: Informational

Context: LibMerkleTree.sol#L15

Description: The `LibMerkleTree` library consists of a combination of wrapper functions of `MerkleProof` without any added functionality, functions that were copied from the `MerkleProof` library and edited for gas optimization purposes and the `leaf` function. While we could not spot any significant flaw in the current version of the code, it is highly recommended to avoid editing external libraries this way as we think that the drawbacks outweigh the benefits in that case. The main drawback is that the process of supporting upstream patches made by the maintainers of the external library will become more complex and as such prune to error. As one can see, this library was already patched less than a year ago and might be patched again in the future.

Recommendation: Consider removing the `LibMerkleTree` library and placing the `leaf` function in the `vFactory` contract instead. Either way, consider bumping the `MerkleProof` version to the most recent version as well.

Kiln: Acknowledged. We will update the OZ library, we will still use the modified version of the multiproof and make sure it's also up to date with the updated oz version we would use. I will also submit a PR to the OZ library to include the modification we brought as an alternative method, our goal being that this method gets included and we can then switch to it in the future.

Spearbit: Acknowledged.

5.5.16 `Nexus`: `pool.reporting` can be stored as a constant variable to avoid typo errors

Severity: Informational

Context: Nexus.sol#L173 Nexus.sol#L378

Description: The value of `pool.reporting` is used twice in the `Nexus` contract to refer to the corresponding value stored at the `$drivers` mapping. The usage of strings that represent real words is susceptible to potential typo errors, the likelihood to perform such a mistake rises in a direct relation with the number of times the string is being typed. In the current version of the code the two instances of usage of this value are consistent but it might become a problem in a future version. In case there is a typo, it may result in a severe damage as the the wrong key will be used.

Recommendation: Consider storing `pool.reporting` as a constant in the `Nexus` contract and use it instead.

Kiln: Resolved by PR 141.

Spearbit: Fixed in commit 9ded0459 by implementing the auditor's recommendation.

5.5.17 Consider tracking `msg.sender` inside the `Stake` when the user performs a stake operation

Severity: Informational

Context: MultiPool20.sol#L299

Description: The new codebase allows users to stake on behalf of an arbitrary `redipient`. This means that the `msg.sender` could be different compared to the `redipient` that will receive the integrator's token.

Recommendation: Kiln should consider tracking the original `msg.sender` inside the `Stake` event to monitor this information.

Kiln: Thanks for this issue, it was resolved by PR 147.

Spearbit: The recommendations have been implemented in the commit b4a54166 of PR 147.

5.5.18 Consider better documenting the "sell commission pool share to the user" logic during the user's stake flow

Severity: Informational

Context: MultiPool20.sol#L253-L273

Description: The integrators/fee recipients earn the fee in two different ways:

1) When there's some commission owed, and it's above the `MIN_COMMISSION_TO_SELL` threshold, the protocol sells the commission pool shares to the users by purchasing the user's ETH and storing this into the `Multipool` contract. At this point, anyone can call `FeeDispatcher.withdrawCommission` to send the fees to the fee-receiver.
2) Otherwise, if there's some commission owed to the fee-receivers some of the "virtual shares owed" will be sent to the exit queue, and they will follow the same flow/path that any staker would follow when they exit the pool.

In the second scenario, because they are passing through the `vExitQueue` and they need to wait for the Validator exit, the fee-receiver will have the same risk that any staker would have. If in during that wait time the validators gets slashed, they will receive less ETH compared to the original one they have asked to withdraw.

In the first scenario instead, where the user purchases the commission shares with their ETH, the fee-receivers will skip all the risk of the `vExitQueue` process.

If Kiln decides to keep this behavior, it should be documented and explained to their users.

Recommendation: When commissions are paid by purchasing user's staked ETH, the pool's recipients are not susceptible to the same risks that the normal users are up to. This behavior should be better and explicitly documented.

Kiln: Thanks for this issue, it was resolved by PR 147.

Spearbit: The recommendations have been implemented in commit 128bbbea of PR 147.

5.5.19 `MultiPool._exitCommissionShares` should adopt the same logic applied by `FeeDispatcher.withdrawCommission`, sending possible dust shares to the last recipient

Severity: Informational

Context: MultiPool.sol#L341-L349

Description: The `_exitCommissionShares` logic should adapt its logic to follow the same one used by `FeeDispatcher.withdrawCommission` to avoid leaving dust `vPool` shares in the contract.

Recommendation: Consider refactoring the `MultiPool._exitCommissionShares` following the same one used by `FeeDispatcher.withdrawCommission`

Kiln: Acknowledged. The code to prevent having dust left after withdrawal is not as elegant in `MultiPool._exitCommissionShares` as in `FeeDispatcher.withdrawCommission`. It's also a function that is very rarely used, meant to prevent edge cases, for these reason we chose to not apply fix for this issue.

Spearbit: Acknowledged.

5.5.20 `FeeDispatcher._setFeeSplit` should not allow zero value split

Severity: Informational

Context: FeeDispatcher.sol#L115

Description: The current implementation of `FeeDispatcher._setFeeSplit` allows the Integrator to setup different recipients with custom split to have a custom logic on how the harvested fee will be distributed to such recipients.

Given the scope of such logic, it would not make sense to have a recipient that will receive `0` fee.

Recommendation: The `_setFeeSplit` should validate each entry of the `splits` input and revert if it's equal to `0`

Kiln: Thanks for this issue, it was resolved by commit be2a77b0.

Spearbit: The recommendations have been implemented in commit be2a77b0 of PR 135.

5.5.21 Consider decreasing the committed ethers amount only after the execution of `vFactory.depositFromRoot`

Severity: Informational

Context: vPool.sol#L253-L269

Description: The current logic of `vPool.purchaseValidators` is updating the committed ethers amount before actually purchasing (and spending) the validators.

Unless there are callbacks/external calls (that rely on committed ethers value) it would be better to update the new value of `committedEthers` after the execution of `vfactory.depositFromRoot` with the actual number of validator purchased by the `vPool` or at least validate that `purchasedValidatorIds.length == purchaseAmount`

With the current `vFactory` logic is fine also like this, just because it will revert if the requested validators are not in the `root` or if one of the requested validators has been already purchased. `vPool` and `vFactory` are controlled contracts, but it's always better to double-check and validate these flows.

Recommendation: Kiln should consider executing `_setCommittedEthers` after `vfactory.depositFromRoot` has been executed and use `LibConstant.DEPOSIT_SIZE * purchasedValidatorIds.length` as the actual amount to be deducted from the current `committedEthers` value.

Kiln: Resolved by PR 141.

Spearbit: The recommendations have been implemented in commit 73633e87 of PR 141.

Kiln should be aware that new revert custom errors have been introduced, and the new logic will revert early and won't throe the `vFactory` error `InvalidMessageValue`. With this in mind, Kiln and other integrators should remember to update their dApps and monitoring tools.

5.5.22 Consider reverting `vFactory depositFromRoot` or `initialize` when the contract's `$root` has not been configured

Severity: Informational

Context: vFactory.sol, vFactory.sol#L204

Description: The `vFactory` allows whitelisted depositors to purchase validators if the purchase request is included in the factory's Merkle Tree.

If the `$root` is empty, the `depositFromRoot` should revert early. An empty `$root` should be considered an invalid or uninitialized state for the `vFactory`. This behavior can also be deducted by the fact that it's not possible to set the `$root` to an empty state via the `setRoot` function.

Recommendation: Kiln should consider to:

1) Add the `root` parameter to the `initialize` and enforce it to not be empty. This will prevent the deployment and initialization of a `vFactory` that has an invalid initial state.

2) `depositFromRoot` should revert early if the `$root.get().root` is empty. It should not be possible to deposit if the Merkle Tree root is not initialized.

Kiln: Acknowledged.

Using `bytes32(0)` as the initial root is good enough:

- `bytes32(0)` as root would prevent any deposit.
- `bytes32(0)` is a value that would mean that the vFactory has not been configured yet.
- the fact that the vFactory cannot provide validators is not detectable anyway, you can add a root without any available validators and there's no way to know.

Spearbit: Acknowledged.

5.5.23 `vFactory` setters should adopt the same logic behavior and be coherent with the current codebase

Severity: Informational

Context: vFactory.sol#L322-L354, vFactory.sol#L357-L389, vFactory.sol#L394-L445, vFactory.sol#L449-L477

Description: The `vFactory` contract has four different setters that allow the deposit's owner to update the deposit's data:

- `setFeeRecipient`
- `setOwner`
- `setAutoClaimThreshold`
- `setExtraData`

The behavior and logic of these setters are not coherent between themselves and with the existing codebase:

1) `setAutoClaimThreshold` is the only setters that will revert when `d.recipientSalt == bytes32(0)`. This means that you can update the `threshold` of the deposit only if the deposit has been created from a deposit that `wc == bytes(0)`. Currently, only the `vPool` can purchase validators and create deposits in the `vFactory` and in this case, the `wc` is **enforced** to be not empty. Currently, the `vPool` does not expose any function to update the deposit and both the `owner` and `feeRecipient` are "static" (cannot be changed in the `vPool`).

   Given such premises and the fact that the code base does not offer a way to perform a deposit with `wc == bytes(0)`, Kiln should consider adopting one of the following options.

   - Adopt the same behavior and prevent executing the `setOwner` and `setFeeRecipient` if `d.recipientSalt == bytes32(0)`.
   - Remove all the setters, given that right now, no contracts can actually enter these functions.

2) `setAutoClaimThreshold` logic skips the update and event emission if the current value is equal to the new one. This logic is not adopted by the other setters that allow the caller to "spam" the `Set*` event. In this case, Kiln should consider adopting one of the following logic when those values are equal:

   1) Early return without updating the storage and emitting the event.
   2) Early revert.

Recommendation: Kiln should consider following the recommendations listed above.

Kiln: Acknowledged. The behavior is as expected, owners can still be changed, same for feeRecipient and extra data even if channel is not 0. The only things that cannot be done for channel !=0 are:

- Setting the threshold.
- Withdrawing.

Spearbit: Acknowledged.

5.5.24 Consider renaming all the `threshold` references inside `vFactory` (and referenced contracts) to a name that explicitly includes the `Gwei` unit

Severity: Informational

Context: vFactory.sol

Description: The `threshold` variable passed as an input parameter to `vFactory.depositFromRoot` and stored in the deposit's struct is expressed in `Gwei` and not in `Wei`. This information is not documented or explained in any part of the codebase or natspec documentation (see for example `DepositFromRootParameters` natspec docs in `IvFactory`). Given that `Wei` is the base, default and common unit in Ethereum, such difference should be well documented and explicit in the name of the variable.

The name (without the explicit unit) is also used and referenced in these places:

- function `autoClaimDetails`.
- struct `DepositFromRootParameters`.
- struct `Deposit`.
- function `setAutoClaimThreshold`.
- event `SetValidatorThreshold`.

Recommendation: Kiln should rename the variables, functions, structs and events to include the explicit unit in their names and provide an appropriate documentation

Kiln: Resolved by PR 141.

Spearbit: The recommendations have been implemented in the commit e671f7ef of PR PR 141.

5.5.25 Improve the `vFactory.depositFromRoot` documentation about all the assumptions and requirements in the scenario of an empty withdrawal channel

Severity: Informational

Context: vFactory.sol#L260-L288

Description: The empty withdrawal channel (`wc == bytes32(0)`) should require that the deposit's `feeRecipient` (the address that will receive the validator's fees like MEV boosts and gas tips) is equal to the withdrawal address, this is mainly needed to be sure that all the funds received (MEV boost, tips, validator's EL rewards/withdrawal) will be withdrawable or auto-claimable as explained in the proposed vsuite 2.1 PR (see "Add `autoClaim threshold` for `MinimalRecipient`" section).

Recommendation: Kiln, depending on their needs or specifications, has two options:

1) enforce that the `feeRecipient` is equal to `computedWithdrawalAddress` when `$$.wc == bytes32(0)`. They can override the input's value or revert if they do no match.
2) allow the `feeRecipient` to have an arbitrary value and enforce this requirement on the caller's level. Note that at the moment of the security review, we do not have access to the code of the contracts that implements the `$$.wc == bytes32(0)` scenario.

In both cases, the logic adopted and applied by Kiln should be documented in the function's code and natspec documentation.

Kiln: Partially fixed by PR 141.

We acknowledge that it's possible for a channel 0 validator to use a custom fee recipient that is not the minimal recipient. Added documentation that explains both activation modes and example configs for each.

Spearbit: Kiln has added some specific documentation in the `IvFactory` interface about the possible values of `$$.wc`

5.5.26 Bulk suggestions, typos or wrong natspec documentation

Severity: Informational

Description:

`vFactory`: create a `_deployWithdrawalAddressFromRawSaltAndInit(salt, id)` that deploys the contract, init it and returns the deployed contract address. This logic is used in three different places and should be placed inside a function instead of being copy/pasted:

- `depositFromRoot` when `wc` is empty and `threshold > 0`.
- `_withdraw` when `withdrawalRecipient` has not been deployed yet (code empty).
- `setAutoClaimThreshold` when `withdrawalRecipient` has not been deployed yet (code empty).

The `vFactory.onlyDepositor` natspec documentation is outdated. The withdrawal address is **not** authorized anymore to access those functions guarded by the modifier. Remove the `@dev` natspec comment.

`vFactory.depositFromRoot`: move `__.validatorId = __.lastId + idx;` above and use it directly instead of recalculating it. It's easier to read and understand.

`vPool.internals._pullExitQueueUnclaimedFunds` natspec is incorrect. Funds are pulled from the exit queue and not the Execution Layer recipient.

`vPool.initialize` is performing a sanity check on `addrs[VPOOL_REPORTING_DRIVER]` but the very same check is already performed inside `_setReportingDriver` that will be called later on. The explicit check in the `initialize` function can be removed.

`FeeDispatcher.withdrawCommission` has repeated code that could be moved to a separate internal function and replaced where needed.

Consider documenting in `FeeDispatcher.withdrawCommission` why the code loops from 0 to n-1 recipient and handles separately the last recipient outside the loop.

☐ In general, code could be further refactored to result more lean and easier to read and maintain. Some logics are repeated during the function execution and could be replaced with an internal function. In other cases, some function's returned value could be stored in local variables and re-used during the flow instead of re-executing the function. This does not only save gas but also makes the code more readable and easier to read and maintain (see for example caching. `_balanceOfUnderlying(...)` returned value during `Liquid20A._transfer`).

`_checkNotForbiddenAndAuthorizations` should avoid using "magic numbers" like `0`. Always use a constant variable with a self-explanatory name.

☐ Consider replacing all the instances of `LibBasisPoints check`, `checkTotal` and `checkWithMaximum` with the `LibSanitize` version of them. This behavior would be more consistent with the codebases that use `LibSanitize` to perform all the input validation checks.

Replace the "magic number" `100` in `vCoverageRecipient.cover` with a proper constant variable. Document such value, explaining that it represents `100bps == 1%`.

`$globalMemberEjectionStatus` natspec is incorrect, the state variable represents the boolean flag relative to the ejection status and not the address of the Nexus.

Consider refactoring some of the `vTreasury` logics that will never be executed:

- In `_transfer`, the `if (poolShares > 0)` branch will never be executed. `_transfer` is only called by `withdraw` that reverts if `$poolShares.get()[token.k()] > 0`. The `withdraw` function only allows the caller to withdraw ETH or tokens that have not been deposited via the `vPool`.
- In `_balance`, the `if (poolSharesBalance > 0)` branch will never be executed. `_balance` is only called by `withdraw` that reverts if `$poolShares.get()[token.k()] > 0`. The `withdraw` function only allows the caller to withdraw ETH or tokens that have not been deposited via the `vPool`.

In `MultiPool` contract, a user's rights authorization can be signed by `authorizer` and `admin`. Currently, in the `_checkAndApplyAuthorization` the `retrievedSigner` value is not logged in any event. This value can

be logged in an event so that in future it can be tracked that who among the admin & authorizer actually signed the authorization.

vPool.internals.sol#L102 Wrong natspec for the `$requestedExits` state variable.

Consider refactoring the `_distributeRewards` function: it could be clearer to not update `currentTotalSupply`, pass `currentTotalSupply + sharesToMint` in the event and `currentTotalSupply` to the `_mint` function.

Consider using `__.traces.increaseLimit` instead of `__.increaseCredit` inside the `else if (__.traces.rewards <= int128(uint128(__.increaseCredit)))` check.

Unused Code:

`MultiPool._checkPoolIsEnabled` function is never used and should be removed from the codebase.

`DepositFromRootInternalVariables.wc` variable is never used. Consider removing it from the struct.

`LBalancePerIdMapping` library is never user. Consider removing it.

`ctypes.User4907` struct is unused.

`LUser4907Mapping` library is unused.

`Freezable.onlyFreezer` modifier is never used.

`LibKey` library is never used.

`MerkleVault` is a legacy contract (vNFT related) so it should be removed from the current codebase.

Typos:

- ☐ `vPoolReporting.driver.sol#L208` `consesnsus` → `consensus`.

Recommendation: Kiln should fix all the recommendations suggested above.

Kiln: Most of these points for the core contracts are resolved in PR 147 and PR 149.

Spearbit: The vast majority of the recommendations have been implemented in the following commits/PRs:

- PR 147.
- PR 140.
- Commit bcbb731f.

The remaining one has been acknowledged and won't be fixed by the client.

5.5.27 Improve the documentation of the `MinimalRecipient.exec` behavior that does not revert when low-level call it is not successful

Severity: Informational

Context: MinimalRecipient.sol#L68

Description: The current implementation of `MinimalRecipient.exec` allows the `owner` of the contract to perform a low-level call that forwards the whole contract's balance

```
return target.call{value: value}(cdata);
```

The function does not check the result of the `call` directly but returns both the `success` and return `data` to the caller that will need to handle it. This means that the upper caller needs to correctly handle the result and revert if the operation was not successful or the data returned that not adhere to what is expected.

This specific behavior should be documented.

Recommendation: Consider documenting such behavior or revert directly in the `exec` function logic if the low-level call fails.

Kiln: Resolved by PR 141.

Spearbit: The recommendations have been implemented in commit 8488a6d1 of PR 141.