Kiln 210 Integration Audit

Security Review

Cantina Managed review by:
R0bert, Lead Security Researcher
Akshay Srivastav, Security Researcher

April 13, 2026

Contents

1 Introduction **2**
- 1.1 About Cantina . . . 2
- 1.2 Disclaimer . . . 2
- 1.3 Risk assessment . . . 2
  - 1.3.1 Severity Classification . . . 2

2 Security Review Summary **3**
- 2.1 Scope . . . 3

3 Findings **4**
- 3.1 Medium Risk . . . 4
  - 3.1.1 Commission counts capped-exit leftovers as yield . . . 4
  - 3.1.2 `maxExitable = 0` does not freeze exit-queue payouts . . . 4
  - 3.1.3 Non-activating validators stay fully priced at 32 ETH . . . 5
  - 3.1.4 `report()` can over-request validator exits and block later queue funding . . . 6
  - 3.1.5 Global member ejection does not fully remove the global oracle's voting weight . . . 6
  - 3.1.6 Rotating the global oracle preserves stale votes and blocks the new global oracle . . . 7
  - 3.1.7 VTreasury clears the wrong pool share balance . . . 7
- 3.2 Low Risk . . . 8
  - 3.2.1 `VTreasury` keeps the previous operator's fee vote after operator rotation . . . 8
  - 3.2.2 `VTreasury` keeps the previous global recipient's fee vote after Nexus rotation . . . 8
  - 3.2.3 `VTreasury` lets authorized callers withdraw pool shares without funding autoCover 9
- 3.3 Informational . . . 9
  - 3.3.1 The global oracle can submit a report alone when it is the only listed member . . . 9
  - 3.3.2 Bare `Native20` upgrade locks users out until rights are re-seeded . . . 10

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

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

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings are a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

From Apr 5th to Apr 6th the Cantina team conducted a review of vsuite on commit hash cbb4df2d. The team identified a total of **12** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	7	0	7
Low Risk	3	0	3
Gas Optimizations	0	0	0
Informational	2	0	2
Total	**12**	**0**	**12**

2.1 Scope

The security review had the following components in scope for vsuite on commit hash cbb4df2d:

```
src
├── ctypes
│   ├── ctypes.sol
│   ├── operator_approvals.sol
│   ├── simple_balance_mapping.sol
│   └── withdrawal_channel_mapping.sol
├── integrations
│   ├── MultiPool.sol
│   └── MultiPool20.sol
└── interfaces
    └── integrations
        ├── IFeeDispatcher.sol
        └── INative20.sol
```

3 Findings

3.1 Medium Risk

3.1.1 Commission counts capped-exit leftovers as yield

Severity: Medium Risk

Context: vPool.sol#L838-L847

Description: `vPool.report()` still books pulled exit-queue `unclaimedFunds` as `__.traces.rewards`. That classification is wrong for the `2.1.0` integrations on `1.0.3` core combination. These funds are not fresh staking yield. They are leftovers created when an exit ticket is capped at its `maxExitable` amount and later gets matched against a richer cask in the exit queue.

The issue becomes integration-visible because `MultiPool` treats any growth in pool value beyond injected principal as fee-bearing performance. `_integratorCommissionEarned()` does that directly:

```
function _integratorCommissionEarned(PoolInfo memory pool) internal view returns
↪   (uint256) {
    uint256 stakedPlusExited = _stakedEthValue(pool) + $exitedEth.get()[pool.id];
    uint256 injected = $injectedEth.get()[pool.id];
    if (injected >= stakedPlusExited) {
        return 0;
    }
    uint256 rewardsEarned = stakedPlusExited - injected;
    return LibBasisPoints.compute(rewardsEarned, $fees.get()[pool.id]);
}
```

Therefore, once `vPool` reinjects capped-exit leftovers and labels them as `rewards`, `Native20` and the other `MultiPool20` descendants start accruing integrator commission on value that did not come from validator performance.

This matters because the leftover value should simply return to the pool and benefit the remaining holders. It should not be turned into commission-bearing revenue for the integrator. The user-visible effect is a silent fee leak from capped exits into integrator commission accounting, even if the report itself contains no new validator rewards.

The easiest way to see the problem is with a realistic example. Assume a `Native20` instance has a 10% integrator fee and currently tracks `100 ETH` of underlying value. Alice requests an exit worth `40 ETH`, so her ticket is capped at `40 ETH` through `maxExitable`. Before she claims, the exit queue later receives enough ETH that the same shares would now settle for `41 ETH`. Alice is still limited to `40 ETH`. The extra `1 ETH` is moved into the exit queue's `unclaimedFunds` buffer because it is no longer claimable by the ticket holder. On the next report, the old core pulls that `1 ETH` back into the pool and adds it to `__.traces.rewards`. `MultiPool` then interprets that `1 ETH` as reward-like growth and accrues `0.1 ETH` of commission, despite the fact that validators did not earn `1 ETH` and no new external reward entered the system. The value came from the capped exit settlement itself.

The regression test below reproduces the behavior directly. It creates an unclaimed-funds buffer in the exit queue, submits a zero-reward report and shows that `integratorCommissionEarned()` becomes positive only because those capped-exit leftovers were reinjected and counted as `rewards`.

```
// Add to: test/integrations/MultiPool20.t.sol
function test_report_pulledUnclaimedFundsIncreaseIntegratorCommission() external {
    uint256 amount = 64 ether;
    uint256 exitAmount = 32 ether;
    uint256 unclaimedDelta = 1e12;
    address staker = makeAddr("staker1");

    stake(staker, amount);

    oracles__reportToCommit(op, amount);
    poolAdmin__bootstrapValidatorSet(o, op);

    {
        ctypes.ValidatorsReport memory rep = oracles__warpAndForwardReport(op,
        ↪  op.pool.lastReport());
```

```
        oracles__report(op, rep);
    }

    vm.prank(staker);
    mp.requestExit(exitAmount);

    uint256 ticketId = op.exitQueue.ticketIdAtIndex(0);
    ctypes.Ticket memory ticket = op.exitQueue.ticket(ticketId);
    expect(ticket.maxExitable).toEqual(exitAmount, "unexpected capped exit value");

    vm.deal(address(op.pool), exitAmount + unclaimedDelta);
    vm.prank(address(op.pool));
    op.exitQueue.feed{value: exitAmount + unclaimedDelta}(ticket.size);

    uint256[] memory ticketIds = new uint256[](1);
    ticketIds[0] = ticketId;
    uint32[] memory caskIds = new uint32[](1);
    caskIds[0] = 0;

    vm.prank(staker);
    op.exitQueue.claim(ticketIds, caskIds, 0);

    expect(op.exitQueue.unclaimedFunds()).toEqual(unclaimedDelta, "unclaimed funds buffer
    ↪  not created");
    expect(mp.integratorCommissionEarned(POOL_ID)).toEqual(0, "commission should stay
    ↪  zero before the pull-back report");

    uint256 underlyingBefore = mp.totalUnderlyingSupply();

    {
        ctypes.ValidatorsReport memory rep = oracles__warpAndForwardReport(op,
        ↪  op.pool.lastReport());
        oracles__report(op, rep);
    }

    expect(op.exitQueue.unclaimedFunds()).toEqual(0, "report should pull the unclaimed
    ↪  funds buffer");
    expect(mp.totalUnderlyingSupply()).toBeGreaterThan(
        underlyingBefore, "integrator underlying should increase when unclaimed funds are
        ↪  reinjected"
    );
    expect(mp.integratorCommissionEarned(POOL_ID)).toBeGreaterThan(
        0, "integrator commission should only increase because the old core counts
        ↪  unclaimed funds as rewards"
    );
}
```

Recommendation: Backport the upstream `2.1.0` fix and stop adding pulled exit-queue unclaimed funds to `__.traces.rewards`. These funds can still be added to `__.traces.delta`, since they do increase pool value, but they should not be treated as fee-bearing yield.

The smallest safe change is to keep the pull-back accounting and remove the reward increment:

```
uint256 pulledAmount = _pullExitQueueUnclaimedFunds(__.increaseCredit);
__.increaseCredit -= pulledAmount;
__.traces.pulledExitQueueUnclaimedFunds = uint128(pulledAmount);
__.traces.delta += int128(uint128(pulledAmount));
```

If the core cannot be changed, the fallback is to exclude this source from integration commission calculations. That is a weaker fix because it spreads special-case logic into the integrations. Aligning `vPool` with the upstream `2.1.0` behavior is cleaner and fixes every `MultiPool20` descendant at once.

Kiln: Acknowledged. Known 1.0.3 core behavior, fixed in 2.1.0 core. This is an accepted limitation of running 2.1.0 integrations on the live 1.0.3 core. The pools on mainnet have and always had 0 as `operatorFee` meaning this path was never encountered.

Cantina Managed: Acknowledged.

3.1.2 `maxExitable = 0` does not freeze exit-queue payouts

Severity: Medium Risk

Context: vPool.sol#L962-L1010

Description: `vPool.report()` still processes the exit queue whenever it holds shares, even if the oracle sets `rprt.maxExitable` to zero. The code computes `exitDemand`, burns exit-queue shares and calls `feed()` without checking that `maxExitable` is greater than zero.

This matters because upstream `2.1.0` uses `maxExitable = 0` as a special emergency value to stop cask creation during slashing or other stress events. On this branch, that stop does not work. Earlier queued exiters can still be paid from newly exited ETH or exit-boost ETH while the oracle is explicitly trying to freeze payouts.

For example, assume the queue already holds a large pending exit and the next report also includes fresh exited ETH from validators. If the oracle sets `maxExitable = 0` to keep funds inside the pool until the slashing window is clear, the current code still burns the queued shares and sends ETH into the exit queue. Consequently, users who are already in the queue can leave before the loss is fully socialized, and the remaining holders absorb more of the eventual damage.

Recommendation: Backport the upstream `2.1.0` guard and skip exit-queue processing when `rprt.maxExitable == 0`.

The smallest safe change is to gate the block like this:

```
if (exitQueueBalance > 0 && rprt.maxExitable > 0) {
//...
}
```

The local tests and compatibility notes should also be updated to reflect that `maxExitable = 0` is an emergency stop for exit-queue payouts, not just a limit on future validator exit requests.

Kiln: Acknowledged. Known 1.0.3 core behavior, fixed in 2.1.0 core. The 1.0.3 core only checks `exitQueueBalance > 0` before processing the exit queue. The 2.1.0 core guards with `exitQueueBalance > 0 && rprt.maxExitable > 0`. The 1.0.3 core does not treat `maxExitable = 0` as an emergency stop. Accepted as a known limitation.

Cantina Managed: Acknowledged.

3.1.3 Non-activating validators stay fully priced at 32 ETH

Severity: Medium Risk

Context: vPool.sol#L1138-L1147

Description: `_totalUnderlyingSupply()` treats every purchased-but-not-yet-activated validator as if it were still backed by a full `32 ETH`. That is reasonable while activation is merely pending. It becomes wrong once a purchased validator never activates. This branch omitted the upstream `2.1.0` invalid-activation reporting and coverage mechanism, so there is no way to stop counting that missing validator as real backing.

This matters because the pool can continue reporting a par rate even after the backing is gone. A purchased validator consumes `32 ETH` from the pool at deposit time. If that validator later turns out to be invalid and never activates, the pool balance is still gone, but `_totalUnderlyingSupply()` keeps adding the missing `32 ETH` back through the purchased-versus-activated validator gap. The loss is hidden instead of crystallized.

That hidden loss becomes user-visible as soon as other users interact with the pool. A realistic example is a pool where Alice funded one validator, the validator never activates and the pool still reports `rate() == 1e18`. Bob then deposits a fresh `32 ETH` and mints at par because the pool still looks fully backed. Alice can transfer her shares to the exit queue and exit for a full `32 ETH`, funded by Bob's fresh deposit. Bob is left holding shares that still look fully backed on-chain even though the pool has already lost the first `32 ETH`.

Recommendation: Backport the upstream invalid-activation handling so the oracle can report non-activating validators separately and the pool can require explicit coverage before those validators continue to count in underlying supply.

If the full `2.1.0` mechanism cannot be backported, the minimum safe rule is to stop counting permanently non-activating purchased validators as `32 ETH` of backing without an explicit replacement or coverage step. Otherwise the pool can keep socializing a hidden loss onto later depositors and remaining holders.

Kiln: Acknowledged. Known 1.0.3 core behavior, fixed in 2.1.0 core. The 1.0.3 core has no `invalidActivationCount` reporting or coverage mechanism. `_totalUnderlyingSupply()` counts every purchased-but-not-activated validator at 32 ETH indefinitely. In practice, validator activation is monitored operationally and invalid activations have not occurred on the live deployment.

Cantina Managed: Acknowledged.

3.1.4 `report()` can over-request validator exits and block later queue funding

Severity: Medium Risk

Context: vPool.sol#L1083-L1096

Description: `vPool.report()` requests more exits whenever `exitDemand` is higher than `exitingProjection`, but it never caps the new requests by the number of validators that are actually activated. The downstream `vFactory.exitTotal()` call only clamps against funded validators. It does not clamp against activated validators. Therefore `requestedExits` can become larger than the number of validators that can really exit.

This matters because `report()` later reuses `requestedExits` as if it were real future liquidity. At the following lines:

```
// src/vPool.sol#956-961
//  ----- We compute the exiting projection, which is the amount of ethers that is
 ↪  expected to be exited soon
//        This amount is based on the exiting amount, which is the amount of eth detected
 ↪  in the exit flow of the
//        consensus layer, and the amount of unfulfilled exit requests that are expected
 ↪  to be triggered by the
//        operator soon
__.traces.exitingProjection = rprt.exiting;
uint256 currentRequestedExits = $requestedExits.get();
if (currentRequestedExits > rprt.stoppedCount) {
    __.traces.exitingProjection += uint128((currentRequestedExits - rprt.stoppedCount) *
     ↪  LibConstant.DEPOSIT_SIZE);
}
```

any `requestedExits - stoppedCount` gap is converted back into `exitingProjection` at `32 ETH` per validator. Once the stored request count is impossible, the pool starts assuming exit ETH is on the way when it is not. That false projection then suppresses exit-queue funding from fresh deposits and keeps queued exits pending longer than necessary.

For example, a pool that has purchased two validators, but only one has actually activated. If users request enough exits, the current code can still store `requestedExits = 2`. On the next report the pool treats both exits as pending future liquidity, even though only one validator can really leave. If a new depositor adds 64 ETH in the meantime, that deposit can remain idle in `deposited` instead of being used as exit-boost liquidity for the queue. The queue is delayed only because the accounting assumes a second exiting validator that does not exist.

Recommendation: Backport the upstream `2.1.0` cap before forwarding the request to the withdrawal recipient. `newExitRequests` should never exceed `rprt.activatedCount - currentRequestedExits`. The smallest safe change is:

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

That keeps `requestedExits` aligned with the oracle-reported activated validator set and prevents `exitingProjection` from being inflated by exits that cannot happen.

Kiln: Acknowledged. Known 1.0.3 core behavior, fixed in 2.1.0 core. The 1.0.3 core does not cap `newExitRequests` by `activatedCount`. The 2.1.0 core adds: `newExitRequests = LibUint256.min( newExitRequests, rprt.activatedCount - currentRequestedExits)`. Accepted as a known limitation. The downstream `vFactory.exitTotal()` provides a partial clamp against funded validators, limiting the practical blast radius.

Cantina Managed: Acknowledged.

3.1.5 Global member ejection does not fully remove the global oracle's voting weight

Severity: Medium Risk

Context: vOracleAggregator.sol#L223-L290

Description: `setGlobalMemberEjectionStatus(true)` only flips the ejection boolean and emits `SetGlobalMemberEjectionStatus`. It does not clear the current reporting state if the global oracle already voted. `submitReport()` also keeps granting the implicit global vote whenever `msg.sender == _globalOracleMember()`, even after ejection, as long as that address is also present in the local member list.

This means ejection mode does not actually deliver the quorum reduction it advertises. A stale global vote cast before ejection can still count after ejection. A global oracle that is also a listed member can also keep contributing two votes after ejection. In both cases, a report can finalize with fewer physical signers than the post-ejection quorum intends.

This matters because ejection mode is supposed to remove the global oracle from the active voting set once there are enough local members. On this branch, the protocol can say the global member is ejected while still inheriting its old vote or its implicit extra vote.

Recommendation: Backport the upstream `2.1.0` ejection handling. Ejection should use a real `_globalMemberEjected()` predicate, clear current reporting state if the global bit already voted and suppress the implicit global vote whenever the global member is considered ejected.

The minimum safe change is to clear the current vote tracker and variant counts when ejection becomes active after a global vote and to stop adding the implicit global vote to `voteCount` once ejection is active.

Kiln: Acknowledged. Known 1.0.3 core behavior, fixed in 2.1.0 core. The 1.0.3 `setGlobalMemberEjectionStatus()` only flips the boolean and emits an event. It does not clear reporting state or suppress the implicit global vote. The 2.1.0 core adds `_globalMemberEjected()` helper, state clearing on ejection, and gates vote registration / counting / emission on ejection status. Accepted as a known limitation.

Cantina Managed: Acknowledged.

3.1.6 Rotating the global oracle preserves stale votes and blocks the new global oracle

Severity: Medium Risk

Context: Nexus.sol#L338-L342, vOracleAggregator.sol#L257-L259, vOracleAggregator.sol#L304-L306, vOracleAggregator.sol#L337-L339, vOracleAggregator.sol#L375-L377

Description: `vOracleAggregator` represents the implicit global oracle in vote-tracker bit `0`. The Nexus can later change the global oracle address through `setGlobalOracle()`, but the aggregator does not clear its current reporting state when that happens. As a result, bit `0` is reused across two different identities.

If the old global oracle already voted for the current epoch, the new global oracle inherits that stale bit and is treated as `AlreadyReported`. The old vote is still counted even though the underlying identity changed. This creates two problems at once. The current epoch can keep counting influence from an address that is no longer the global oracle. It can also deadlock report finalization if the new global oracle vote is required to reach quorum.

Recommendation: Clear the aggregator's reporting state whenever the corresponding Nexus global oracle address changes.

If that is not feasible through the current architecture, stop keying implicit-global participation off a bare bit position that survives identity rotation. The voting state should either be reset on rotation or tied to the current global-oracle address rather than to a fixed slot alone.

Kiln: Acknowledged. Known 1.0.3 core behavior. `Nexus._setGlobalOracle()` does not clear aggregator reporting state on rotation. If the old global oracle already voted for the current epoch, the new oracle inherits a stale bit and is treated as `AlreadyReported`. Neither 1.0.3 nor 2.1.0 clear aggregator state directly from the Nexus setter, though 2.1.0's improved ejection handling mitigates the impact. Oracle rotation is an infrequent admin action typically performed between epochs when the vote tracker is already cleared. Accepted as a known limitation.

Cantina Managed: Acknowledged.

3.1.7 VTreasury clears the wrong pool share balance

Severity: Medium Risk

Context: vTreasury.sol#L156

Description: `vTreasury` records received pool shares under the pool address in `onvPoolSharesReceived`, since `msg.sender` is the pool when shares are transferred to the treasury. `exitShares` also reads the pending balance with that same pool key before calling `_exitAndFundCoverageFund`.

`_exitAndFundCoverageFund` then clears `$poolShares` with `msg.sender.k()`. At that point `msg.sender` is the operator or the global recipient, not the pool whose shares are being exited. Therefore the real pool entry is left unchanged even though the treasury has already transferred the underlying shares out.

The impact is not theft. The impact is that treasury accounting for one pool can become stuck in a permanently inflated state after the first successful `exitShares(pool)` call. Once that happens, later attempts to exit or withdraw shares for that pool can revert because the treasury believes it owns more shares than it actually holds. Consequently, operator commission distribution and any auto-cover handling that depends on those exits can stop working for the affected pool until storage is repaired.

For example, assume the treasury receives `100` shares from pool `P`. It stores that amount under `$poolShares[P]`. The operator then calls `exitShares(P)`, which transfers the real `100` shares out, but the contract clears `$poolShares[operator]` instead of `$poolShares[P]`. The treasury now holds `0` real shares from `P`, while bookkeeping still says it holds `100`. If the treasury later receives `20` more shares from `P`, the real balance becomes `20` but the stored balance becomes `120`. A new `exitShares(P)` call will try to transfer `120` shares even though only `20` exist in the treasury, so the call reverts. The `100` share mismatch remains forever unless someone repairs storage.

Recommendation: Clear the pool-keyed entry instead of the caller-keyed entry. The smallest safe change is to reset `$poolShares` with `address(pool).k()` before transferring any shares out.

```
$poolShares.get()[address(pool).k()] = 0;
```

If this function has already been used on live deployments add a one-off repair step to fix any stale `$poolShares` entries before relying on future exits or withdrawals.

Kiln: Acknowledged. Known 1.0.3 core behavior, fixed in 2.1.0 core. Accepted as a known limitation. The operator fee on the live pools is set to 0%, `vTreasury` are not used in the current flows, so these have no practical impact on the live deployment.

Cantina Managed: Acknowledged.

3.2 Low Risk

3.2.1 `VTreasury` keeps the previous operator's fee vote after operator rotation

Severity: Low Risk

Context: vTreasury.sol#L270-L274

Description: `vTreasury` stores fee votes as raw values with an active bit. It does not store which operator cast the vote. When `setOperator()` updates the operator address, `_setOperator()` only writes the new address and emits `SetOperator`. It does not clear an already active operator vote.

Consequently, an outgoing operator can leave a live vote behind. The current global recipient can later cast the same fee value and finalize the treasury fee even though the current operator never agreed. This is a governance-integrity issue for treasury fee changes. It does not directly expose user balances, but it means operator rotation does not revoke pending fee influence from the old operator.

Recommendation: Clear treasury vote state whenever the operator changes.

```
function _setOperator(address newOperator) internal {
    LibSanitize.notZeroAddress(newOperator);
    $operator.set(newOperator);
    emit SetOperator(newOperator);
    $operatorFeeVote.set(0);
    $globalRecipientFeeVote.set(0);
}
```

If you want a stronger fix, bind stored votes to the voter identity so a role rotation automatically invalidates votes cast by the previous role holder.

Kiln: Acknowledged. Known 1.0.3 core behavior, fixed in 2.1.0 core. Accepted as a known limitation. In 2.1.0. `_setOperator()` at line 281 clears votes if an operator vote exists: `if ($operatorFeeVote.get() > 0) { _clearVotes(); }`. The operator fee on the live pools is set to 0%, `vTreasury` are not used in the current flows, so these have no practical impact on the live deployment.

Cantina Managed: Acknowledged.

3.2.2 `VTreasury` keeps the previous global recipient's fee vote after Nexus rotation

Severity: Low Risk

Context: vTreasury.sol#L194-L223

Description: `voteFee` stores the global recipient vote as a raw fee value with an active bit. It does not bind that vote to a specific `globalRecipient` address. Authorization is checked against the current `Nexus.globalRecipient()` value at call time, but the stored vote itself remains in the treasury when `Nexus.setGlobalRecipient()` rotates the role.

Consequently, a previous global recipient can preload a vote, get rotated out, and still influence a later fee update. The current operator can match that stale value and finalize the treasury fee even though the current global recipient never agreed. This is a governance-integrity issue around treasury fee changes rather than a direct loss of user funds.

Recommendation: Clear treasury vote state whenever `Nexus.globalRecipient` changes, or store the address associated with each live vote and reject it if that address is no longer the current global recipient.

If the treasury is expected to keep using role-based voting, consider adding a reset hook on global-recipient rotation so old pending votes cannot survive the role change.

Kiln: Acknowledged. Known limitation, not fixed in any version. Valid observation. Neither 1.0.3 nor 2.1.0 clear treasury vote state on global recipient rotation - `Nexus._setGlobalRecipient()` only writes the new address and emits the event. A previous global recipient's vote can survive rotation and be matched by the current operator to finalize a fee change. However, the practical risk is limited: it requires the current operator to independently and deliberately match the stale vote value and the impact is constrained to treasury fee governance, not user funds. The operator fee on the live pool is set to 0% and the treasury is not used in the current integration flow.

Cantina Managed: Acknowledged.

3.2.3 `VTreasury` lets authorized callers withdraw pool shares without funding autoCover

Severity: Low Risk

Context: vTreasury.sol#L227-L249

Description: `vTreasury` only applies `autoCover(pool)` inside `_exitAndFundCoverageFund`, where part of the operator's shares are redirected to the pool's coverage recipient before the rest is sent to the exit queue. `withdraw(token)` does not enforce that rule. If `token` is a pool share token, the function simply splits the entire treasury balance between the global recipient and the operator. Consequently, an authorized caller can bypass operator-configured coverage funding by choosing `withdraw(address(pool))` instead of `exitShares(pool)`. The coverage recipient receives nothing even when `autoCover(pool)` is non-zero. This weakens the intended slashing buffer for that pool and lets a fee recipient skip the coverage routing that the treasury configuration was supposed to enforce.

Recommendation: Do not allow direct withdrawal of treasury-held pool shares. Force that case through `exitShares(pool)`, where `autoCover` is applied.

```
function withdraw(address token) external onlyOperatorOrGlobalRecipient {
    LibSanitize.notZeroAddress(token);
    if ($poolShares.get()[token.k()] > 0) {
        revert CannotWithdrawPoolShares(token);
    }
//...
}
```

This matches the local `2.1.0` behavior and preserves the intended coverage routing for treasury pool shares.

Kiln: Acknowledged. Known 1.0.3 core behavior, fixed in 2.1.0 core. Accepted as a known limitation. The operator fee on the live pools is set to 0%, `vTreasury` are not used in the current flows, so these have no practical impact on the live deployment.

Cantina Managed: Acknowledged.

3.3 Informational

3.3.1 The global oracle can submit a report alone when it is the only listed member

Severity: Informational

Context: vOracleAggregator.sol#L254-L325

Description: `vOracleAggregator` treats the contract as ready as soon as `members.length > 0`. It also gives `msg.sender` one vote as a local member and one more vote if the same address is also the Nexus global oracle. Therefore, if the only listed member is the global oracle itself, that one address gets 2 votes against a quorum of 2.

This breaks the intended quorum model. The report is supposed to require some non-global corroboration before it reaches the pool. On this branch, the global oracle can satisfy the entire quorum alone by being added as the only listed member.

Upstream `2.1.0` explicitly lists this as an audit fix.

Recommendation: Backport the upstream `2.1.0` readiness rule so a sole listed member only makes the aggregator ready when that member is not the global oracle.

If that full backport is not taken, the minimum safe fix is to reject the configuration where the global oracle is the only effective member in the voting set. The aggregator should never let one address satisfy both the local-member vote and the global-member vote with no other signer participating.

Kiln: Acknowledged. Known 1.0.3 core behavior, fixed in 2.1.0 core. The 1.0.3 `_ready()` only checks `members.length > 0`, allowing a sole member who is also the global oracle to satisfy quorum alone with 2 votes. The 2.1.0 core's `_ready()` rejects this configuration: it returns false when the sole member is the global oracle. Accepted as a known limitation.

Cantina Managed: Acknowledged.

3.3.2 Bare `Native20` upgrade locks users out until rights are re-seeded

Severity: Informational

Context: AccountList.sol#L178-L207

Description: The new `Native20` implementation starts enforcing `AccountList` rights immediately. When an account has no explicit rights entry, `_getRights()` falls back to `defaultRights`. `_checkNotForbiddenAndAuthorizations()` then reverts if the required bit is missing.

This becomes an upgrade issue because the live mainnet deployments still have `defaultRights == 0`. After only `upgradeTo(newImplementation)`, ordinary users no longer satisfy the new authorization checks. `MultiPool20._stake()` starts failing for fresh users, and `MultiPool20._requestExit()` starts failing for existing holders for the same reason. The live-fork reproduction in `test/e2e/Native20.upgrade.v2_1_0.research.t.sol` shows this on all 24 listed mainnet `Native20` proxies: a user can stake before the upgrade, but after only the implementation swap a new user reverts on `stake()` and the pre-upgrade holder reverts on `requestExit()` with `MissingAuthorizations`.

This matters because restoring access is not part of the same privileged action. `upgradeTo()` is restricted to the ERC1967 proxy admin, while `setDefaultRights()` is restricted to the separate integration admin stored in contract state. Therefore the rollout is not a safe standalone implementation swap. There is a user lockout window until the second admin action completes. The existing fork test hides this because it always calls `setDefaultRights(STAKE | REQUEST_EXIT)` before checking any user behavior.

Recommendation: Do not treat `upgradeTo(newImplementation)` as a safe standalone rollout.

If current user stake and exit access must be preserved, seed the intended default rights atomically during the upgrade, for example through an upgrade reinitializer or `upgradeToAndCall()`. If that is not possible, make the `setDefaultRights()` transaction an explicit required part of the rollout procedure and document that users are locked out until it executes.

Kiln: Acknowledged. Valid observation. The recommendation to use `upgradeToAndCall()` to seed default rights atomically is acknowledged but breaks the separation of concerns between admins and would require extra code for a one time action. Upgrade is a one time action, both actions can be coordinated fairly easily, at block n and n+1 for example, which is far simpler and requires no extra code while having virtually no impact on users.

Cantina Managed: Acknowledged.