Fastlane Re-review

Security Review

Cantina Managed review by:
Riley Holterhus, Lead Security Researcher
Christos Pap, Security Researcher

January 29, 2026

Contents

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

ShMonad is a liquid staking protocol for Monad developed by FastLane Labs. The protocol allows users to stake MON in exchange for a liquid staking token that accrues staking rewards.

From Jan 5th to Jan 9th the Cantina team conducted a review of fastlane-contracts-internal on commit hash 903f80d3. The team identified a total of **10** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	4	4	0
Medium Risk	2	2	0
Low Risk	2	2	0
Gas Optimizations	0	0	0
Informational	2	1	1
Total	**10**	**9**	**1**

2.1 Scope

The security review had the following components in scope for fastlane-contracts-internal on commit hash 903f80d3:

```
src/shmonad
├── AtomicUnstakePool.sol
├── Coinbase.sol
├── Constants.sol
├── Errors.sol
├── Events.sol
├── FLERC20.sol
├── FLERC4626.sol
├── Holds.sol
├── libraries
│   ├── AccountingLib.sol
│   ├── FeeLib.sol
│   ├── HoldsLib.sol
│   ├── StakeAllocationLib.sol
│   └── StorageLib.sol
├── Policies.sol
├── PrecompileHelpers.sol
├── ShMonad.sol
├── StakeTracker.sol
├── Storage.sol
├── Types.sol
└── ValidatorRegistry.sol
```

3 Findings

3.1 High Risk

3.1.1 `completeUnstake` can zero reserves backing `rewardsPayable` and stall validator cranks

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: `_beforeCompleteUnstake` borrows just enough from the atomic pool to cover a redemption when `reservedAmount < amount`, then immediately debits the full redemption from `reservedAmount` and `redemptionsPayable`. If reserves were partly backing `rewardsPayable`, they are wiped out. The next validator crank pays rewards via `_handleRewardsPaidSuccess`, which does `reservedAmount -= amount`. With reserves zeroed this underflows and reverts, halting validator cranks and epoch advancement. The `_beforeCompleteUnstake` function:

```
/// @notice Handles accounting of completion of unstake request
/// @param amount The unstake completion amount
function _beforeCompleteUnstake(uint128 amount) internal virtual override {
    uint128 reservedAmount = s_globalCapital.reservedAmount;
    if (reservedAmount < amount) {
        // CASE: The reserved amount alone is not enough to meet the redemptions
        // To get the missing amount, we remove it from the atomic liquidity pool.
        uint128 _amountNeededFromAtomicLiquidity = amount - reservedAmount;

        AtomicCapital memory _atomicAssets = s_atomicAssets;
        uint128 _atomicAllocatedAmount = _atomicAssets.allocatedAmount;
        uint128 _atomicUtilizedAmount = _atomicAssets.distributedAmount;

        // The unutilized liquidity in the atomic unstaking pool must be enough to cover
        ↪  the required amount
        require(
            _atomicAllocatedAmount - Math.min(_atomicUtilizedAmount,
            ↪  _atomicAllocatedAmount)
                >= _amountNeededFromAtomicLiquidity,
            InsufficientReservedLiquidity(amount, reservedAmount)
        );

        // Take the last bit of the withdrawal from the atomic liquidity pool by
        ↪  crediting the allocated asset,
        // offset by debiting the reserved amount.
        s_atomicAssets.allocatedAmount -= _amountNeededFromAtomicLiquidity; // -Asset Cr
            // _amountNeededFromAtomicLiquidity
        s_globalCapital.reservedAmount += _amountNeededFromAtomicLiquidity; // +Asset Dr
            // _amountNeededFromAtomicLiquidity;
    }

    // Reduce the reserved amount and the liability by the amount withdrawn
    s_globalCapital.reservedAmount -= amount; //
    s_globalLiabilities.redemptionsPayable -= amount; // -Liability Dr amount
}
```

The `_handleRewardsPaidSuccess` function:

```
/// @notice Handles accounting of successful payment / transfer of escrowed MEV rewards
↪  to a validator
/// @param amount The amount successfully paid
function _handleRewardsPaidSuccess(uint128 amount) internal {
    // ShMonad MON -> Validator
    s_globalCapital.reservedAmount -= amount; // >>> UNDERFLOWS HERE
    s_globalLiabilities.rewardsPayable -= amount; // -Liability Dr amount
}
```

Proof of Concept: Add the `test_ShMonad_completeUnstake_partialCrank_leavesReservedShortfall`

test in `test/shmonad/TraditionalUnstaking.t.sol`. It showcases that partial crank leaves a validator pending with `rewardPayable`. Then `completeUnstake` borrows from atomic, zeroing reserves. Next `crank()` reverts with arithmetic underflow when rewards are paid.

```
function test_ShMonad_completeUnstake_partialCrank_leavesReservedShortfall() public {
    if (!useLocalMode) vm.skip(true);

    // Goal: show a natural path where a partial crank lets `completeUnstake` zero
    ↪  reserves while
    // rewardsPayable remains outstanding, causing the next crank to revert on underflow.

    // Keep most assets in the atomic pool so validator unstake capacity is limited per
    ↪  epoch.
    vm.prank(deployer);
    shMonad.setPoolTargetLiquidityPercentage(SCALE * 9 / 10);

    // Add a new validator at the tail so a partial crank can leave it unprocessed.
    address validator = makeAddr("partialCrankValidator");
    uint64 valId = staking.registerValidator(validator);
    vm.prank(deployer);
    shMonad.addValidator(valId, validator);
    _activateValidator(validator, valId);

    // Seed earned revenue across two epochs so staking allocations are enabled.
    uint256 rewardAmount = 2 ether;
    uint256 depositAmount = 200 ether;
    uint256 mevReward = 1 ether;
    vm.deal(alice, depositAmount + rewardAmount * 3 + mevReward + 1 ether);
    for (uint256 i = 0; i < 2; i++) {
        vm.prank(alice);
        shMonad.sendValidatorRewards{ value: rewardAmount }(valId, SCALE);
        _advanceEpochAndCrank();
    }

    // Deposit and then keep queueToStake eligible in this epoch to push capital into
    ↪  staking.
    vm.prank(alice);
    uint256 shares = shMonad.deposit{ value: depositAmount }(depositAmount, alice);

    vm.prank(alice);
    shMonad.sendValidatorRewards{ value: rewardAmount }(valId, SCALE);

    _advanceEpochAndCrank();

    // Request a large unstake so reserved capital is insufficient at completion.
    (uint128 stakedAmount,) = shMonad.getWorkingCapital();
    assertGt(stakedAmount, 0, "expected staked capital after revenue seeding");
    uint256 targetAssets = uint256(stakedAmount) * 5;
    uint256 sharesToUnstake = shMonad.convertToShares(targetAssets);
    if (sharesToUnstake > shares) sharesToUnstake = shares;
    assertGt(sharesToUnstake, 0, "sharesToUnstake must be non-zero");

    uint256 expected = shMonad.convertToAssets(sharesToUnstake);

    vm.prank(alice);
    uint64 completionEpoch = shMonad.requestUnstake(sharesToUnstake);

    // Advance to just before the completion epoch.
    _advanceToInternalEpoch(completionEpoch - 1);

    // Queue validator rewards payable in the prior epoch so they roll into last and
    ↪  remain unpaid.
    // Using feeRate=0 makes the full value a validator payout (i.e., rewardsPayable).
    vm.prank(alice);
    shMonad.sendValidatorRewards{ value: mevReward }(valId, 0);
```

```
    // Partial crank: advances global epoch but leaves validators pending.
    bool complete = _advanceEpochAndPartialCrank(1_050_000);
    assertFalse(complete, "partial crank should leave validators pending");
    assertEq(shMonad.getInternalEpoch(), completionEpoch);
    assertTrue(shMonad.getNextValidatorToCrank() != address(0), "validators should still
    ↪  be pending");
    assertTrue(shMonad.isValidatorCrankAvailable(valId), "validator should still be
    ↪  pending");

    {
        // Rewards payable should have rolled into the last slot but not been paid due to
        ↪  pending validator.
        (uint120 lastRewardsPayable,,,) = shMonad.getValidatorRewards(valId);
        assertEq(lastRewardsPayable, uint120(mevReward), "expected validator rewards
        ↪  payable to remain");
    }

    (, uint128 reservedBefore) = shMonad.getWorkingCapital();
    (uint128 atomicAllocatedBefore, uint128 atomicDistributedBefore) =
    ↪  shMonad.getAtomicCapital();

    // Ensure we take the reserved<amount branch and that atomic liquidity can cover the
    ↪  shortfall.
    assertLt(reservedBefore, expected, "expected reserved shortfall at completion");
    assertGe(
        uint256(atomicAllocatedBefore) - uint256(atomicDistributedBefore),
        expected - reservedBefore,
        "atomic pool should cover shortfall"
    );

    // completeUnstake borrows from atomic liquidity and then debits the full amount from
    ↪  reserved.
    _completeUnstakeAndAssertPayout(expected);

    (uint128 atomicAllocatedAfter,) = shMonad.getAtomicCapital();
    uint256 expectedAtomicDelta = expected - reservedBefore;
    assertEq(uint256(atomicAllocatedBefore) - uint256(atomicAllocatedAfter),
    ↪  expectedAtomicDelta);

    {
        // BUG SETUP: reserved is now zeroed, but rewardsPayable is still outstanding.
        // The next validator crank will call _handleRewardsPaidSuccess and underflow
        ↪  reservedAmount.
        (, uint128 reservedAfter) = shMonad.getWorkingCapital();
        (uint128 rewardsPayable,,) = shMonad.globalLiabilities();
        assertEq(reservedAfter, 0, "reserved should be zeroed by completeUnstake");
        assertEq(rewardsPayable, uint128(mevReward), "rewards payable should remain
        ↪  outstanding");
    }

    // BUG ASSERTION: crank should revert due to reservedAmount underflow during rewards
    ↪  payout.
    vm.expectRevert(stdError.arithmeticError);
    shMonad.crank();
}
```

Recommendation: Separate the reserve portion backing `rewardsPayable` so redemptions cannot consume it.

Fastlane: Fixed in PR 677.

Cantina Managed: Verified. The `_beforeCompleteUnstake()` function now removes the rewards payable from the reserved amount that it uses to facilitate redemptions.

3.1.2 `_settleGlobalNetMONAgainstAtomicUnstaking()` adjustments don't affect `reservedAmount`

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: During a global crank, the `_settleGlobalNetMONAgainstAtomicUnstaking()` function adjusts `s_atomicAssets.allocatedAmount` and `s_atomicAssets.distributedAmount`. These adjustments depend on several factors, including the current total equity in the system and the `targetLiquidityPercentage` value.

These adjustments can either free liquidity from the atomic liquidity pool or require additional liquidity to be allocated for it. In either case, the net difference is ultimately utilized in the accounting, for example:

```
// CASE: stake delta is positive - we need to stake more unstaked assets
if (_stakeIn > _unstakeOut) {
    uint120 _netStakeIn = _stakeIn - _unstakeOut;
    uint120 _currentQueueForUnstake = _globalCashFlows.queueForUnstake;

    // First try to net the net stake in against the queueForUnstake
    if (_netStakeIn > _globalCashFlows.queueForUnstake) {
        _globalCashFlows.queueForUnstake -= _currentQueueForUnstake; // = 0
        _globalCashFlows.queueToStake += (_netStakeIn - _currentQueueForUnstake);
    } else {
        _globalCashFlows.queueForUnstake -= _netStakeIn;
    }
} // ...
```

However, this netting step has an accounting error. It only ever nets against `queueToStake` and `queueForUnstake`, and it does not consider the system's existing liabilities and `reservedAmount`, which should have higher priority.

One example consequence is when a large `requestUnstake()` executes. This would lower the total equity in the system and free liquidity from the atomic liquidity pool, but because the logic only nets against `queueToStake`/`queueForUnstake`, the freed liquidity ends up being routed into staking rather than toward the redemption liability. This has downstream effects on the accounting and can, for example, cause the system to fail to fulfill all redemptions and leave behind permanently staked MON even if all participants attempt to withdraw.

Recommendation: Update `_settleGlobalNetMONAgainstAtomicUnstaking()` so that liquidity adjustments also consider the current liabilities and `reservedAmount`.

Fastlane: Fixed in PR 677.

Cantina Managed: Verified.

3.1.3 Coinbase commission payout can freeze the system

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: When the `process()` function is called on the main `coinbase` implementation contract, it sends the validator commission via an EVM transfer. Because most functions that ultimately call `process()` (such as `crank()`) are not protected by reentrancy guards, the commission recipient can therefore gain control-flow and re-enter shMonad to perform balance-changing operations during the `process()` call. This affects the accounting pattern surrounding the `process()` call:

```
uint256 _balance = address(this).balance;
try ICoinbase(coinbaseContract).process{ gas: COINBASE_PROCESS_GAS_LIMIT }() { } catch {
↪  }
uint256 _rewardAmount = address(this).balance - _balance;
```

In the worst case, the commission recipient could perform an action that decreases the shMonad contract's balance (for example, completing a redemption). This would cause the

`address(this).balance - _balance` subtraction to underflow, which would revert a `crank()` and permanently freeze the system.

Recommendation: Consider adding reentrancy guards on all functions that can reach the `process()` call, as well as on all functions that modify the shMonad contract's balance.

Also, evaluate whether any external contracts or mechanisms can send ETH into shMonad in ways that could be triggered permissionlessly when the commission recipient gains control flow, as these paths would not be restricted by shMonad's reentrancy protections.

Fastlane: Fixed in PR 670.

Cantina Managed: Verified. Reentrancy guards have been added to all functions that can modify the shMonad contract's balance.

3.1.4 Malicious coinbase addresses can be used in `processCoinbaseByAuth()`

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: The `processCoinbaseByAuth()` function is permissionless and takes an arbitrary `coinbase` address. It then calls `VAL_ID()`, `SHMONAD()`, and `AUTH_ADDRESS()` on that address. If the return values match expected values (e.g. `address(this)` or `msg.sender`), the function proceeds to call `process()` on the provided `coinbase`.

Because these checks rely only on return values, a malicious `coinbase` contract can be supplied that simply returns values that satisfy the checks. For example, a malicious `coinbase` can return the attacker's address for `AUTH_ADDRESS()`.

If an attacker does this, the shMonad contract will call `process()` on the attacker's contract. One consequence is that the attacker can then re-enter shMonad during the `process()` call, perform an action that changes shMonad's balance (e.g. a `deposit()` call), which will then misinterpret the balance change as a donation:

```
// Track balance in order to allow validators to donate priority fees to boost yield.
uint256 _balance = address(this).balance;
try ICoinbase(coinbaseContract).process{ gas: COINBASE_PROCESS_GAS_LIMIT }() { } catch {
↪  }
uint256 _rewardAmount = address(this).balance - _balance;

if (_rewardAmount > 0) {
    _handleValidatorRewards(valId, _rewardAmount, SCALE);
}
```

This essentially allows an attacker to wrap normal balance-changing actions so they are double-counted as validator reward donations. This will create unbacked revenue and protocol commission, which breaks the accounting.

Recommendation: Consider adding a check that the provided `coinbase` address actually corresponds to a registered validator in the shMonad system, rather than allowing an arbitrary contract. This could be enforced by checking the `s_valIdByCoinbase` / `s_valCoinbases` mappings.

If it is necessary to support processing `coinbase` addresses that are not explicitly registered (for example, previous `coinbase` implementations), consider mitigating this issue by preventing any balance-changing actions from executing during the `process()` call. One way to do this is to add reentrancy guards to `processCoinbaseByAuth()` and all functions that can change the contract's ETH balance. In this csae, it should be carefully considered whether calling `process()` on arbitrary addresses can still cause issues in the system. It should also be considered whether there are any external contracts or mechanisms that can send ETH into the system in a way that could be triggered permissionlessly from within `process()`.

Fastlane: Fixed in PR 670.

Cantina Managed: Verified. The `processCoinbaseByAuth()` function is now an `onlyOwner` function, and reentrancy guards have been added to balance changing functions that previously did not have reentrancy guards.

3.2 Medium Risk

3.2.1 User controlled overflow in `shares + minCommitted` can block agent debits

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: `Policies._spendFromCommitted` compares `fundsAvailable` to `shares + committedData.minCommitted` in `uint128` space. A user can set `minCommitted` near `type(uint128).max` (via `setMinCommittedBalance` or `_requestUncommitFromPolicy`), so any nonzero `shares` addition overflows and reverts. After this, any policy agent entrypoint that debits committed shares (`agentTransferFromCommitted`, `agentTransferToUncommitted`, `agentWithdrawFromCommitted`) reverts even when the user has committed or uncommitting balances, preventing payment that require that call not to be reverting.

A malicious user can perform the following to block agent debits:

1. Ensure the account has committed shares under that policy.
2. User sets an extreme minimum: `setMinCommittedBalance(policyId, type(uint128).max)` (or `requestUncommit(..., newMinBalance)` with the same value).
3. Policy agent calls `agentTransferToUncommitted(policyId, user, shares, user)` (or any other agent debit). `_spendFromCommitted` executes `shares + committedData.minCommitted` in `uint128`, overflows, and the call reverts, indefinitely blocking agent debits for that user/policy until the user lowers `minCommitted`.

Recommendation: Perform the comparison in widened arithmetic (cast to `uint256` before addition) or cap `minCommitted` so `shares + minCommitted` cannot overflow for any valid debit.

Fastlane: Fixed in PR 682.

Cantina Managed: Verified.

3.2.2 `_carryOverAtomicUnstakeIntoQueue()` double counts `earnedRevenue` and inflates atomic liquidity

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: Atomic liquidity is freed during the global crank by `_carryOverAtomicUnstakeIntoQueue()` using the entire `earnedRevenue`, even when part of that revenue is already earmarked in `allocatedRevenue` (shortfall path) or has been spent paying atomic withdrawals. Because `earnedRevenue` is never reduced on withdrawal, this can reduce `distributedAmount` by "phantom" revenue and overstate atomic liquidity (`allocatedAmount - distributedAmount`) relative to real MON on balance, inflating `maxWithdraw` or fee quotes and letting users pull funds that should remain reserved.

```
function _carryOverAtomicUnstakeIntoQueue() internal {
    // NOTE: We set this to globalRevenue.earnedRevenue so that there is no "jump" in the
    ↪  fee cost
    // whenever we crank
    uint120 _amountToSettle =
        Math.min(globalRevenuePtr_N(0).earnedRevenue,
        ↪  s_atomicAssets.distributedAmount).toUint120();

    s_atomicAssets.distributedAmount -= _amountToSettle; // -Contra_Asset Dr
    ↪  _amountToSettle
    // Implied: currentAssets -= _amountToSettle; // -Asset Cr _amountToSettle

    globalCashFlowsPtr_N(0).queueForUnstake += _amountToSettle;
}
```

Recommendation: Settle atomic utilization only against *available* revenue. You can cap the carry over by `earnedRevenue - allocatedRevenue` (and ideally by liquid

`currentAssets/address(this).balance)` and then update `earnedRevenue/allocatedAmount` accordingly so revenue isn't double used.

Fastlane: Fixed in PR 679.

Cantina Managed: Verified.

3.3 Low Risk

3.3.1 `redeem()/previewRedeem()` skip liquidity cap and `maxRedeem` guard can still revert

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: `redeem()/previewRedeem()` compute fees with `_quoteFeeFromGrossAssetsNoLiquidityLimit` and never clamp net assets to the atomic pool's available liquidity. The only guard is `maxRedeem()`, which derives its bound via `previewWithdraw(maxWithdraw(owner))`. `maxWithdraw` is liquidity aware, but `previewWithdraw` is an inverse that ignores the cap and uses different rounding. At liquidity limits, `shares <= maxRedeem` can still produce a `netAssets` quote above current availability, causing `_accountForWithdraw` to revert with `InsufficientBalanceAtomicUnstakingPool` after the precheck passes. This violates ERC4626 expectations (redeem with `<= maxRedeem` should not revert) and it can potentially DoS integrators that rely on `redeem` for atomic exits.

```
function redeem(uint256 shares, address receiver, address owner) public virtual
↪  nonReentrant returns (uint256) {
    require(receiver != address(0), ZeroAddress());
    uint256 maxShares = maxRedeem(owner);
    require(shares <= maxShares, ERC4626ExceededMaxRedeem(owner, shares, maxShares));

    (uint256 netAssets, uint256 feeAssets) = _previewRedeem(shares);

    _withdraw(_msgSender(), receiver, owner, netAssets, shares, feeAssets);

    receiver.safeTransferETH(netAssets);

    return netAssets;
}

function maxRedeem(address owner) public view virtual returns (uint256) {
    // `maxWithdraw()` factors in fee and liquidity limits
    uint256 maxWithdrawAssetsAfterFee = maxWithdraw(owner);

    // Then, use `previewWithdraw()` to go from netAssets back to gross shares, as per
    ↪  ERC-4626 semantics.
    return previewWithdraw(maxWithdrawAssetsAfterFee);
}

function previewRedeem(uint256 shares) public view virtual returns (uint256) {
    (uint256 netAssets,) = _previewRedeem(shares);
    return netAssets;
}

function _previewRedeem(uint256 shares) internal view virtual returns (uint256
↪  netAssets, uint256 feeAssets) {
    uint256 _grossAssets = _convertToAssets({
        shares: shares,
        rounding: Math.Rounding.Floor,
        deductRecentRevenue: true,
        deductMsgValue: false
    });
    feeAssets = _quoteFeeFromGrossAssetsNoLiquidityLimit(_grossAssets);
    netAssets = _grossAssets - feeAssets;
}

function _quoteFeeFromGrossAssetsNoLiquidityLimit(uint256 grossRequested)
    internal
```

```
    view
    virtual
    returns (uint256 feeAssets);
```

Recommendation: Make the redeem path liquidity aware. You can mirror withdraw and clamp or require against the capped forward model. Alternatively, you can redefine `maxRedeem` to use the same liquidity capped forward computation as `redeem` so `shares <= maxRedeem` never reverts at the liquidity boundary.

Fastlane: Fixed in PR 683.

Cantina Managed: Verified.

3.3.2 `_currentAssets` usage in `_accountForDeposit()` is unclear

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: The `_accountForDeposit()` function inspects the system's current liabilities to determine how much of an incoming deposit will eventually be set aside to cover liabilities. If a portion of the deposit does not need to be set aside, part of it will be redirected toward the atomic liquidity pool. This is implemented as follows:

```
function _accountForDeposit(uint256 assets) internal virtual override {
    // ...
    if (_currentLiabilities > _reservedAssets + _pendingUnstaking + _currentAssets) {
        uint256 _uncoveredLiabilities = _currentLiabilities - (_reservedAssets +
        ↪  _pendingUnstaking);
        if (assets > _uncoveredLiabilities) {
            uint256 _surplus = assets - _uncoveredLiabilities;

            // Send a portion of the surplus to atomic liquidity, then readd the
            ↪  uncovered liabilities
            // to get the assets that need to be queued to stake.
            assets = _uncoveredLiabilities + _subtractNetToAtomicLiquidity(_surplus);
        }

        // CASE: None of the assets are needed to cover liabilities
    } else {
        assets = _subtractNetToAtomicLiquidity(assets);
    }
    // ...
}
```

Notice that the initial check for whether part of the deposit must be set aside takes `_currentAssets` into account, while the calculation of `_uncoveredLiabilities` does not. The reason for this discrepancy is unclear.

Related to this is the fact that `_accountForDeposit()` does not increment `reservedAmount` directly, it only is checking what will likely happen during the next global crank.

This discrepancy can have downstream effects. For example, if two deposits occur, the first increases `_currentAssets` and affects the initial check on the second, but it does not change the `_uncoveredLiabilities` in the second deposit.

Recommendation: Consider clarifying or adjusting how `_currentAssets` is treated between the initial shortfall check and the `_uncoveredLiabilities` calculation.

Fastlane: Fixed by adding clarifying comments in PR 678.

Cantina Managed: Verified.

3.4 Informational

3.4.1 Cross account commits emit misleading ERC20 Transfer logs

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: `_commitToPolicy` emits `Transfer(accountFrom, sharesRecipient, shares)` when committing to another recipient, even though the recipient's ERC20-visible `balanceOf()` does not increase (only their committed bucket does). The same call also emits `Transfer(accountFrom, address(this), shares)`, so an indexer may see two outgoing transfers for one spend. Because `balanceOf()` is defined as `uncommitted` only, these logs do not reflect real ERC20 balance movements.

Recommendation: Only emit ERC20 `Transfer` logs for actual ERC20-visible balance moves (uncommitted). Remove the cross-account `Transfer(accountFrom, sharesRecipient, ...)` and rely on dedicated commitment events (`Commit`, `RequestUncommit`, `CompleteUncommit`).

Fastlane: Fixed in PR 671.

Cantina Managed: Verified.

3.4.2 `Coinbase` rate updates apply retroactively

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The `Coinbase` contract allows the `authAddress` to update `s_config.mevCommissionRate`, `s_config.donationRate`, and `s_config.priorityCommissionRate`. When these rates are updated, the new values are used the next time `process()` runs. This means that the rate updates apply retroactively to MON that accrued under the old rates. The one exception is MON that is already moved into `s_unpaid` storage.

Recommendation: Keep this behavior in mind and ensure that validators are aware of this behavior.

Fastlane: Acknowledged.

Cantina Managed: Acknowledged.