# Fastlane LST Protocol

## 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 Nov 17th to Dec 4th the Cantina team conducted a review of fastlane-contracts-internal on commit hash 03b06cb0. The team identified a total of **36** issues:

**Issues Found**

| Severity | Count | Fixed | Acknowledged |
|---|---|---|---|
| Critical Risk | 0 | 0 | 0 |
| High Risk | 3 | 3 | 0 |
| Medium Risk | 8 | 6 | 2 |
| Low Risk | 8 | 6 | 2 |
| Gas Optimizations | 0 | 0 | 0 |
| Informational | 17 | 11 | 6 |
| **Total** | **36** | **26** | **10** |

## 2.1 Scope

The security review had the following components in scope for fastlane-contracts-internal on commit hash 03b06cb0:

```
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 `externalReward()` calls don't consider max external reward

**Severity:** High Risk

**Context:** StakeTracker.sol#L977-L1011

**Description:** The `_settleValidatorRewardsPayable()` function settles the `rewardsPayable` that a validator has accumulated from `_handleValidatorRewards()` calls in the previous internal epoch. The main function it uses is the staking precompile's `externalReward()` function.

The staking precompile reverts if the external reward exceeds a maximum allowed value. Monad's public docs currently reference 1,000,000 MON as the limit, however the codebase now appears to have a different threshold of 10,000,000 MON.

The ShMonad codebase does not account for this maximum. If a validator's `rewardsPayable` exceeds the precompile limit, the `externalReward()` call will revert and `_handleRewardsRedirect()` will run as a fallback. This causes the entire reward to be diverted to shMON holders, and the validator and its external delegators miss out on the rewards that were supposed to be distributed to them.

Because `rewardsPayable` represents MEV rewards, it may be possible for a rare MEV event to naturally trigger this scenario. Also, note that anyone can call `sendValidatorRewards()` to donate MON and increase `rewardsPayable` for a validator. Since `_handleRewardsRedirect()` immediately increases the shMON exchange rate (with no revenue smoothing), users may have an incentive to donate rewards to push a validator above the limit and cause a redirect.

Note that the same issue applies to the `externalRewards()` call in the `Coinbase` contract.

**Recommendation:** Add logic to ensure `externalReward()` calls respect the upper limit. One possible approach is to split large `rewardsPayable` amounts into multiple `externalReward()` calls. With this idea, one implementation decision to consider is whether the partial payments are executed in one main loop or spread into future epochs, and this decision may lead to other considerations.

**Fastlane:** Fixed in PR 616, PR 634, and PR 675.

**Cantina Managed:** Verified.

### 3.1.2 `s_globalPending` updates break accounting invariants

**Severity:** High Risk

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

**Description:** `s_globalPending` tracks the global `pendingStaking` and `pendingUnstaking` amounts. These values increase when delegate/undelegate actions are initiated and decrease once those actions are finalized in later epochs.

There is only one `s_globalPending` variable, and `_crankValidators()` updates `s_globalPending` as each validator is processed. This means that by the time a validator is reached, all validators processed earlier in the same crank have already incremented or decremented the pending values based on their own actions. As a result, the pending amounts are not a consistent epoch-level state that applies to all validators.

This breaks the accounting and can cause the system to error and become stuck. For example, if earlier validators initiate withdrawals (increasing `pendingUnstaking`), the global unstakeable amount returned by `getGlobalAmountAvailableToUnstake()` is reduced for validators processed later. A later validator can then hit an underflow in `calculateValidatorEpochStakeDelta()`, since that function expects the `globalCashFlows_Last.queueForUnstake` to have been clamped to be less than the global unstakeable amount, which is not correctly maintained.

**Recommendation:** Update the `s_globalPending` accounting to provide a consistent epoch-level snapshot of the system.

**Fastlane:** Fixed in PR 624 and PR 635.

**Cantina Managed:** Verified fix. Note that future upgrades should be careful with timing. Since the proxy admin owner is a Gnosis Safe, and Safe transactions can be executed by anyone once signatures are ready, the upgrade could be executed at an unexpected time. With this change, an upgrade in the middle of validator cranking could lead to accounting errors, since some validators would use the updated accounting and others would not. However, accounting errors already exist in the current code regardless of the upgrade, so this is unavoidable.

### 3.1.3 `agentWithdrawFromCommitted()` burn is not adjusted when withdrawal is clamped

**Severity:** High Risk

**Context:** ShMonad.sol#L210-L224

**Description:** In `agentWithdrawFromCommitted()`, when `amountSpecifiedInUnderlying == false`, the following logic runs:

```
uint256 _grossAssetsWanted = _convertToAssets(amount, OZMath.Rounding.Floor, true,
↪  false);
uint256 _grossAssetsCapped;
(_grossAssetsCapped, _feeTaken) =
↪  _getGrossCappedAndFeeFromGrossAssets(_grossAssetsWanted);
_assetsToReceive = _grossAssetsCapped - _feeTaken;
_sharesToDeduct = amount.toUint128();
```

Here, the `amount` is treated as a gross shares amount. This is converted into a gross asset amount via `_convertToAssets()`. That gross asset amount is then fed into `_getGrossCappedAndFeeFromGrossAssets()`.

The `_getGrossCappedAndFeeFromGrossAssets()` function checks whether the net assets (after fees) would exceed the unstaking pool's available liquidity. If so, it clamps the gross asset amount so the net withdrawal fits within the pool's limits.

If this clamping happens, notice that the `_assetsToReceive` in the above code will be reduced, yet the `_sharesToDeduct` is not adjusted to account for the fact that fewer assets are being withdrawn. This can lead to a very large loss of value if the `_assetsToReceive` are heavily clamped and the original `amount` was a large amount of shares.

In the worst case, an attacker could frontrun another user's call to `agentWithdrawFromCommitted()` with their own unstaking actions to intentionally reduce liquidity in the unstaking pool, which would force `agentWithdrawFromCommitted()` to realize a loss of assets relative to the shares burned.

**Recommendation:** Consider proportionally adjusting `_sharesToDeduct` when `_getGrossCappedAndFeeFromGrossAssets()` clamps the withdrawal. Alternatively, consider reverting when the clamping occurs. This is how the `amountSpecifiedInUnderlying == true` case works.

**Fastlane:** Fixed in PR 624 by adding a revert in this scenario.

**Cantina Managed:** Verified.

## 3.2 Medium Risk

### 3.2.1 Incorrect underflow protection prevents slashing accounting adjustments

**Severity:** Medium Risk

**Context:** StakeTracker.sol#L1477-L1483

**Description:** In the `_handleComplete_Allocation()` function in `StakeTracker`, when handling the case where `_actualActiveStake < _expectedActiveStake` (indicating potential slashing), the code attempts to decrement `validatorEpochPtrLast.targetStakeAmount` and `s_globalCapital.stakedAmount` by `_delta` with underflow protection. However, the underflow protection logic is flawed and prevents the decrement from occurring.

The code calculates `_delta128` as follows:

```
uint128 _delta128 = (_delta > _expectedTotalStake ? _delta - _expectedTotalStake :
↪  0).toUint128();

validatorEpochPtrLast.targetStakeAmount -= _delta128;
s_globalCapital.stakedAmount -= _delta128;
```

The issue is that `_delta` is calculated as `_expectedActiveStake - _actualActiveStake`, where `_expectedActiveStake = _expectedTotalStake - _actualPendingDeposits`. This means:

```
_delta = _expectedActiveStake - _actualActiveStake
       = (_expectedTotalStake - _actualPendingDeposits) - _actualActiveStake
       ≤ _expectedTotalStake
```

Since `_delta` can never exceed `_expectedTotalStake`, the condition `_delta > _expectedTotalStake` will always evaluate to false, resulting in `_delta128` always being set to 0. Consequently, the accounting adjustments to `validatorEpochPtrLast.targetStakeAmount` and `s_globalCapital.stakedAmount` never occur.

However, this scenario may not occur under normal circumstances, as slashing is not yet implemented on Monad and the condition `_actualActiveStake < _expectedActiveStake` is not expected to happen in typical operation. If such conditions were to happen, the underflow protection logic would prevent the necessary accounting adjustments from being applied, leading to potential accounting issues.

**Recommendation:** Consider fixing the underflow protection logic to properly handle the decrement. The intended behavior appears to be taking the minimum of `_delta` and the available stake amount. However, since `_delta` is already bounded by `_expectedTotalStake`, the underflow protection may be unnecessary for this specific case.

If underflow protection is desired, one solution may be to cap the `_delta` against `validatorEpochPtrLast.targetStakeAmount` instead:

```
uint128 _delta128 = Math.min(_delta,
↪  uint256(validatorEpochPtrLast.targetStakeAmount)).toUint128();

validatorEpochPtrLast.targetStakeAmount -= _delta128;
s_globalCapital.stakedAmount -= _delta128;
```

**Fastlane:** Fixed in PR 628.

**Cantina Managed:** Verified.

#### 3.2.2 `_crankGlobal()` uses stale unstakeable amount

**Severity:** Medium Risk

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

**Description:** When a new Monad epoch begins, `_crankGlobal()` typically runs before `_crankValidators()`. Entering a new epoch can change the effective global unstakeable amount, for example, prior deposits may now be active. However, this change is not reflected in the internal accounting until `_crankValidators()` updates the relevant state.

As a result, any usage of `getGlobalAmountAvailableToUnstake()` during the global crank is using stale data. This affects the `_offsetExcessQueueCapacityWithNet()` and `_clampQueuesToCapacityOrRoll()` functions, which use the unstakeable amount to adjust `queueToStake` and `queueForUnstake`. Using outdated values here can affect the downstream actions during validator cranking.

**Recommendation:** Consider updating the global unstakeable accounting to be up-to-date in the global crank. This could be accomplished if the global accounting knew how many pending deposits are now active in the new epoch.

**Fastlane:** We disagree with this finding, not because it's wrong but because we already do the proposed solution.

I believe that we want globalUnstakableAmount and the validatorUnstakableAmount to both be snapshotted to the same moment.

As discussed in finding "`s_globalPending` updates break accounting invariants", we're (going to be) snapshotting the `s_globalPending_Last` so that it doesn't update (and using that in stake allocation calculations) and then making it so that `s_globalPending` continues to continuously update.

In order for us to know how many pending deposits are now active in the new epoch, we'd need to call out to each of the validators. That's a potentially unbounded for loop of external calls, so instead we've just added that functionality to the validator crank.

Typically, after the global crank is run the validator crank will immediately be run next and it does exactly what you're describing, with the exception of it being separate transactions. Note that:

- The 'current' (not last, per fix of "`s_globalPending` updates break accounting invariants") GlobalUnstakableAmount is used only for the calculation of when a user can complete unstaking.
- We don't want to put an unbounded for loop inside the global crank - we may run out of gas.
- We don't want to break the global crank up into multiple transactions - that would allow the possibility of users doing deposits and withdrawals in the middle of the global crank stake updates which seems like a huge security risk.

So we've made the global crank concise and we've made the validator-specific stuff not-time sensitive. A downside of this is that the withdrawal waiting period is longer than itd otherwise have to be, but we view this as a totally acceptable tradeoff given the alternatives.

**Cantina Managed:** Acknowledged.

#### 3.2.3 Validator removal can strand `rewardsPayable` and `earnedRevenue` accounting

**Severity:** Medium Risk

**Context:** StakeTracker.sol#L1178-L1211

**Description:** When `sendValidatorRewards()` is called for a `validatorId` that's currently registered in shMonad, the function sets aside rewards to be distributed as `externalRewards()` for the validator in the next epoch.

A problem can arise if this happens while the validator is pending removal (e.g. via `deactivateValidator()`). If the global crank has already run but the validator has not yet been cranked, the subsequent validator crank might complete the removal. This means the rewards planned to be distributed in the next epoch will never be paid out, and will remain permanently as a reserved liability in the contract.

There may be similar concerns with the `_handleBoostYield()` and `_handleBoostYieldFromShares()` functions. Although these functions don't increment a validator's `rewardsPayable`, they do increment the `earnedRevenue`. As a result, in the next epoch if the validator is removed, the global `earnedRevenue` might no longer equal the sum of all validators' earned revenue.

**Recommendation:** Consider preventing these edge cases. One option is to modify the functions so validators pending deactivation are treated as no longer belonging to shMonad.

**Fastlane:** Fixed in PR 646 and PR 674.

**Cantina Managed:** Verified.

#### 3.2.4 Delayed `externalReward()` call allows delegation frontrunning

**Severity:** Medium Risk

**Context:** StakeTracker.sol#L975-L978

**Description:** The `sendValidatorRewards()` function sets aside rewards to be shared with the validator and its delegators in the next epoch. Because there is a delay before rewards are actually sent via `externalReward()`, a user could frontrun and delegate directly to the validator to gain access to the incoming rewards.

**Recommendation:** Consider making the `externalReward()` call immediately in `sendValidatorRewards()`.

**Fastlane:** We do not perceive this to be an issue at this time. It is an unfortunate side effect of the Monad staking design. Full mitigations are impossible, and partial mitigations would only work for MEV earned during the boundary block - a very small percentage of the overall MEV. We do not believe a minor mitigation is worth the extra complexity.

Note also that if we could deposit the validator rewards when they're collected then we would, but MEV transactions are the most gas-sensitive of any transaction type, and the external reward function is quite gas intensive. External rewarding in each MEV transaction is a non-starter. Running a second crank would not fully solve the issue, either.

We're lobbying the monad team for the deposit wait period to be the same as the withdrawal wait period (1 epoch - 2 in boundary) rather than its shorter, current duration (0 - 1). From our POV, that is the only real solution.

**Cantina Managed:** Acknowledged.

#### 3.2.5 Incorrect `pendingStaking` handling in `getValidatorAmountAvailableToUnstake()`

**Severity:** Medium Risk

**Context:** StakeAllocationLib.sol#L140-L159

**Description:** The `getValidatorAmountAvailableToUnstake()` function is implemented as follows:

```
function getValidatorAmountAvailableToUnstake(
    Epoch memory validatorEpoch_LastLast,
    Epoch memory validatorEpoch_Last,
    StakingEscrow memory validatorPendingEscrow_Last,
    StakingEscrow memory validatorPendingEscrow_LastLast
)
    internal
    pure
    returns (uint256 amount)
{
    amount = validatorEpoch_Last.targetStakeAmount;
    if (validatorEpoch_Last.hasDeposit) {
        uint256 _unavailable_Last = validatorPendingEscrow_Last.pendingStaking;
        amount = _saturatingSub(amount, _unavailable_Last);
    }
    if (validatorEpoch_LastLast.hasDeposit &&
    ↪  validatorEpoch_LastLast.crankedInBoundaryPeriod) {
        uint256 _unavailable_LastLast = validatorPendingEscrow_LastLast.pendingStaking;
        amount = _saturatingSub(amount, _unavailable_LastLast);
    }
}
```

The purpose of this function is to subtract deposits from previous epochs that are still pending. The only deposits that can still be pending in a given Monad epoch are those made during the boundary period of the previous Monad epoch. The current implementation does not reflect this rule and subtracts pending staking from earlier internal epochs in cases where it should not.

A more accurate function would look like:

```
function getValidatorAmountAvailableToUnstake(
    Epoch memory validatorEpoch_Last,
    StakingEscrow memory validatorPendingEscrow_Last
)
    internal
    pure
    returns (uint256 amount)
{
    amount = validatorEpoch_Last.targetStakeAmount;
    if (validatorEpoch_Last.hasDeposit && validatorEpoch_Last.crankedInBoundaryPeriod) {
        uint256 _unavailable_Last = validatorPendingEscrow_Last.pendingStaking;
```

```
        amount = _saturatingSub(amount, _unavailable_Last);
    }
}
```

However note that this logic also becomes incorrect when `validatorEpoch_Last` corresponds to a Monad epoch that occurred more than one Monad epoch ago. In such cases, the deposit is already finalized from Monad's perspective even though the internal epoch difference is only one.

In practice, most calls to `getValidatorAmountAvailableToUnstake()` happen after `_settlePastEpochEdges()` clears the `hasDeposit` flag for finalized deposits, which reduces the likelihood of issues. However the function is incorrect in isolation and it is incorrect whenever the previous internal epoch is more than one Monad epoch ago.

**Recommendation:** Consider removing the `validatorEpoch_LastLast` and `validatorPendingEscrow_LastLast` logic, and only subtract pending staking from `validatorPendingEscrow_Last` when the deposit was made in the boundary period of the Monad epoch directly before the current one.

**Fastlane:** Fixed in PR 672.

**Cantina Managed:** Verified. Other changes made the `getValidatorAmountAvailableToUnstake()` unused, so it has been removed.

#### 3.2.6 `_settlePastEpochEdges()` can finalize actions late

**Severity:** Medium Risk

**Context:** StakeTracker.sol#L412-L443

**Description:** The `_settlePastEpochEdges()` function settles delegate/undelegate actions from previous epochs that may now be finalized. It does this by checking the previous three internal epochs for `hasWithdrawal` or `hasDeposit` values. The implementation bases finalization solely on the number of internal epochs that have passed, even though multiple Monad epochs may have elapsed between two internal epochs.

As a result, the logic does not always update the accounting when Monad first considers these actions finalized. For example, if the previous internal epoch actually corresponds to two Monad epochs ago, the deposit is already finalized from Monad's perspective, but `_settlePastEpochEdges()` may not treat it as finalized yet.

The main consequence is that accounting updates can occur later than they really should, which has downstream effects on things like stake allocation decisions. The current behavior does not appear to cause missed finalizations, since all deposits and withdrawals are eventually finalized after three internal epochs.

**Recommendation:** To keep the accounting up-to-date with the current Monad state, consider updating `_settlePastEpochEdges()` to take into account the actual Monad epoch associated with the prior internal epochs when determining whether actions should be finalized.

**Fastlane:** Addressed in PR 645.

**Cantina Managed:** Verified. Note that this is a partial fix. The new code makes settlement earlier in some scenarios where the previous shmonad epoch is older than one monad epoch. Settlement for epochs further back is unchanged.

#### 3.2.7 Validators can be cranked twice within one Monad epoch

**Severity:** Medium Risk

**Context:** StakeTracker.sol#L196-L205

**Description:** If a validator is cranked in a later Monad epoch than the global crank, then that validator crank can be immediately followed by the next global crank. Because the global crank resets the validator cursor, the validator becomes eligible to be cranked again within the same Monad epoch. This causes two internal validator epochs to correspond to a single Monad epoch. This breaks assumptions in parts of the accounting logic.

For example, `_settlePastEpochEdges()` may treat a deposit from internal epoch -1 as finalized even though the deposit was just made in the same timestamp. This might cause the system to attempt to withdraw more stake than is actually available. While the implementation uses try/catch blocks to prevent the crank queue from getting stuck in these scenarios, this still has downstream accounting consequences.

Another concern is withdrawal finalization. The `_settlePastEpochEdges()` function only looks back three internal epochs when determining whether a withdrawal should be finalized. If internal epochs advance quickly enough relative to the amount of Monad epochs, it may be possible for four internal epochs to pass before a withdrawal becomes eligible from Monad's perspective. If this happens, the withdrawal would never be finalized or recorded, leading to an untracked withdrawal (which could also cause reverts once the withdrawal id is reached again). However it appears that this is not possible in the code, partially due to the fact that a global crank in an epoch's boundary period must wait one full epoch until the next global crank.

**Recommendation:** Ensure that internal validator epochs cannot advance in a way that causes two of them to map to the same Monad epoch.

**Fastlane:** Fixed in PR 637.

**Cantina Managed:** Verified. After the change, `globalEpochPtr_N(0).epoch` gets synced when validator cranking runs after a Monad epoch boundary, so `_crankGlobal()` can't immediately run again in the same Monad epoch and advance the internal epoch twice.

#### 3.2.8 `process()` early return blocks commission payouts

**Severity:** Medium Risk

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

**Description:** In the Coinbase `process()` function, the contract splits its current MON balance into a validator commission and a reward portion to send via the staking precompile's `externalReward()` function. Because `externalReward()` reverts if called with a value less than `MIN_VALIDATOR_DEPOSIT` (1 MON), the code adds an early return when `_rewardPortion < MIN_VALIDATOR_DEPOSIT`:

```
function process() external onlyShMonad returns (bool success) {
    // ...
    if (_rewardPortion < MIN_VALIDATOR_DEPOSIT) return false;
    // ...
    try this.sendCommissionAndRewards(_config.commissionRecipient, _validatorCommission,
    ↪  _rewardPortion) {
        success = true;
    } catch {
        success = false;
    }
}
```

This early return can stop `process()` prematurely even when it should still pay out the commission. For example if the validator's commission rate is 100%, then `_rewardPortion` will always be zero, so `process()` will always hit the early return and never call `sendCommissionAndRewards()`. This means the validator will never be sent the commission.

This issue was raised by the shMonad team during the audit.

**Recommendation:** Refactor the early return logic so that it doesn't unnecessarily block the validator commission payout.

**Fastlane:** Fixed in a refactor that created a new Coinbase contract. This included PR 614 and PR 665.

**Cantina Managed:** Verified.

### 3.3 Low Risk

#### 3.3.1 `ECDSA.recover` may prevent `ERC-1271` fallback path for contract accounts

**Severity:** Low Risk

**Context:** FLERC20.sol#L181

**Description:** The `permit()` function in `FLERC20.sol` uses `ECDSA.recover()` which reverts on malformed signatures, preventing the ERC-1271 fallback path from being checked for contract accounts. This may cause valid permit operations from smart contract wallets (including EIP-7702 accounts) to fail when the ECDSA signature format is invalid, even if the contract account has a valid ERC-1271 signature.

In the `permit()` function at `FLERC20.sol`, the code attempts to recover the signer address using `ECDSA.recover()`:

```
function permit(
    address owner,
    address spender,
    uint256 value,
    uint256 deadline,
    uint8 v,
    bytes32 r,
    bytes32 s
)
    public
    virtual
{
    require(block.timestamp <= deadline, ERC2612ExpiredSignature(deadline));

    bytes32 structHash = keccak256(abi.encode(PERMIT_TYPEHASH, owner, spender, value,
    ↪  _useNonce(owner), deadline));
    bytes32 hash = _hashTypedDataV4(structHash);

    // 1) Try ECDSA first (works for EOAs, including 7702 EOAs with transient code)
    address recovered = ECDSA.recover(hash, v, r, s);
    if (recovered != owner) {
        // 2) If not an ECDSA signer, try ERC-1271 for contract accounts
        if (owner.code.length == 0) revert ERC2612InvalidSigner(recovered, owner);
        bytes memory sig = abi.encodePacked(r, s, v);
        bool ok = SignatureChecker.isValidSignatureNow(owner, hash, sig);
        require(ok, ERC2612InvalidSigner(address(0), owner));
    }

    _approve(owner, spender, value);
}
```

The issue is that `ECDSA.recover()` reverts when the signature is malformed (e.g., invalid `v`, `r`, or `s` values). When this happens, the entire transaction reverts before reaching the ERC-1271 fallback check. This may prevent contract accounts from using ERC-1271 validation when their signatures are not valid ECDSA signatures.

OpenZeppelin's `SignatureChecker.isValidSignatureNow()` function addresses this by using `ECDSA.tryRecover()` instead of `ECDSA.recover()`, which returns an error code rather than reverting, allowing the code to proceed to ERC-1271 validation when ECDSA recovery fails.

**Recommendation:** It is recommended to replace `ECDSA.recover()` with `ECDSA.tryRecover()` to handle malformed signatures gracefully and allow the ERC-1271 fallback path to execute:

```
// 1) Try ECDSA first (works for EOAs, including 7702 EOAs with transient code)
(address recovered, ECDSA.RecoverError err, ) = ECDSA.tryRecover(hash, v, r, s);
if (err != ECDSA.RecoverError.NoError || recovered != owner) {
    // 2) If not an ECDSA signer, try ERC-1271 for contract accounts
    if (owner.code.length == 0) revert ERC2612InvalidSigner(recovered, owner);
    bytes memory sig = abi.encodePacked(r, s, v);
    bool ok = SignatureChecker.isValidSignatureNow(owner, hash, sig);
    require(ok, ERC2612InvalidSigner(address(0), owner));
}
```

Alternatively, consider using `SignatureChecker.isValidSignatureNow()` directly for both EOA and contract account cases, as it handles both scenarios internally and uses `tryRecover()` appropriately.

However, note that this approach checks for code length first, which may not align with the current implementation's logic that attempts ECDSA recovery first.

**Fastlane:** Fixed in PR 623.

**Cantina Managed:** Verified.

### 3.3.2 Unexpected error event emitted on successful withdrawal

**Severity:** Low Risk

**Context:** StakeTracker.sol#L948-L953

**Description:** In the `_settleCompletedStakeAllocationDecrease()` function in `StakeTracker`, the `UnexpectedStakeSettlementError` event is emitted when a withdrawal operation succeeds. The event name suggests an error condition, but it is triggered in the success path where `_success` is `true` and `_handleCompleteDecreasedAllocation()` is called normally.

```
} else if (_success) {
    _handleCompleteDecreasedAllocation(
        valId, validatorEpochPtr, validatorPendingPtr, _amountReceived.toUint120()
    );

    emit UnexpectedStakeSettlementError(coinbase, valId, _amountReceived, 1);
```

The same event is also correctly emitted in the failure path when `_success` is `false`, making it ambiguous whether the event indicates success or failure without examining the `actionIndex` parameter.

**Recommendation:** Consider removing the event emission from the success path, or renaming the event to better reflect its purpose.

**Fastlane:** Fixed in pull/620. Replaced incorrect `UnexpectedStakeSettlementError` emission in the success path of `_settleCompletedStakeAllocationDecrease()` with `ValidatorUnstakeCompleted` event.

**Cantina Managed:** Verified.

### 3.3.3 Block timing inconsistency between `uncommittingCompleteBlock` view and completion check

**Severity:** Low Risk

**Context:** Policies.sol#L664-L667

**Description:** There is a timing inconsistency between the `uncommittingCompleteBlock()` view function and the actual completion check in `_completeUncommitFromPolicy()`. The view function returns `uncommitStartBlock + escrowDuration` as the completion block, indicating that uncommitting can be completed at that block.

However, the `_completeUncommitFromPolicy()` function uses a strict greater than comparison (`block.number > uncommittingData.uncommitStartBlock + policyEscrowDuration`), requiring an additional block before completion is allowed.

```
function uncommittingCompleteBlock(uint64 policyID, address account) external view
↪   returns (uint256) {
    return s_uncommittingData[policyID][account].uncommitStartBlock +
    ↪   s_policies[policyID].escrowDuration;
}
```

```
require(
    block.number > uncommittingData.uncommitStartBlock + policyEscrowDuration,
    UncommittingPeriodIncomplete(uncommittingData.uncommitStartBlock +
    ↪   policyEscrowDuration)
);
```

When `block.number` equals `uncommitStartBlock + escrowDuration`, users querying `uncommittingCompleteBlock()` will receive a value indicating completion is ready, but calls to `completeUncommit()`, `completeUncommitWithApproval()`, or other completion functions will revert until the next block.

This creates a one block delay between when the view function indicates readiness and when completion is actually possible.

**Recommendation:** Consider aligning the timing logic by either:

1. Changing the condition in `_completeUncommitFromPolicy()` from `>` to `>=`, allowing completion at `uncommitStartBlock + escrowDuration` as the view function suggests:

```
    require(
-       block.number > uncommittingData.uncommitStartBlock + policyEscrowDuration,
+       block.number >= uncommittingData.uncommitStartBlock + policyEscrowDuration,
        UncommittingPeriodIncomplete(uncommittingData.uncommitStartBlock +
        ↪  policyEscrowDuration)
    );
```

2. Alternatively, updating the `uncommittingCompleteBlock()` view function to return `uncommitStartBlock + escrowDuration + 1` to match the actual completion requirement, and updating any related documentation accordingly.

**Fastlane:** Fixed in PR 622. Changed comparison from `>` to `>=` so users can complete uncommit at the exact block returned by the view function.

**Cantina Managed:** Verified.

### 3.3.4 ERC4626 standard compliance issues

**Severity:** Low Risk

**Context:** FLERC4626.sol#L97-L107, FLERC4626.sol#L482

**Description:** The implementation deviates from the ERC4626 standard. Two ways have been found, but as this is a custom implementation that has some differences from ERC4626, more may be present:

- The `deposit()` function in `FLERC4626` is declared as `public payable`, but the ERC4626 standard specifies that `deposit()` should be `external` and `non-payable`. The implementation relies on `msg.value` to receive native MON tokens, as seen in the `_deposit()` function which checks `require(assets == msg.value, IncorrectNativeTokenAmountSent())`. When integrators interact with the contract through the standard ERC4626 interface using `IERC4626(shmonad).deposit(...)`, the call may revert if native tokens are not sent, or the function may not behave as expected if integrators assume the standard non-payable signature.
- The `previewRedeem()` function may revert with extremely large share values when the contract has accumulated significant equity (for example, after `boostYield()` calls). The function internally calls `_convertToAssets()`, which performs a multiplication operation `shares.mulDiv(_equity + 1, _realTotalSupply() + 10 ** _decimalsOffset(), rounding)`. With very large share values near `type(uint256).max` and large equity values, this multiplication can overflow, causing the preview function to revert. Per ERC4626 specification, preview functions MUST NOT revert for any input value, as they are intended to be used for off-chain calculations and user interface displays.

```
function deposit(
    uint256 assets,
    address receiver
)
    public
    payable
    virtual
    notWhenClosed
    nonReentrant
    returns (uint256)
```

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

function _convertToAssets(
```

```
    uint256 shares,
    Math.Rounding rounding,
    bool deductRecentRevenue,
    bool deductMsgValue
)
    internal
    view
    virtual
    returns (uint256)
{
    uint256 _equity = deductMsgValue
        ? _totalEquity({ deductRecentRevenue: deductRecentRevenue }) - msg.value
        : _totalEquity({ deductRecentRevenue: deductRecentRevenue });
    return shares.mulDiv(_equity + 1, _realTotalSupply() + 10 ** _decimalsOffset(),
    ↪  rounding);
}
```

**Recommendation:** Consider the following approaches to improve ERC4626 compliance:

- For the `deposit()` function signature issue, consider documenting the deviation from the standard and providing clear integration guidance.
- For the `previewRedeem()` function, document that, as per the current design, this may overflow in some extreme cases.

**Fastlane:** Added documentation in PR 636.

**Cantina Managed:** Verified.

#### 3.3.5 Division by zero when `totalEquity` is zero can block the `_crankGlobal()` execution

**Severity:** Low Risk

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

**Description:** In the `_checkSetNewAtomicLiquidityTarget()` function, `_totalEquity` is calculated from the global capital state and used as the denominator in percentage calculations via `_scaledPercentFromAmounts()`. When all users request unstake of their equity valued shares, `redemptionsPayable` may reduce `_totalEquity` to exactly zero.

The `_scaledPercentFromAmounts()` function performs division without checking if the denominator is zero:

```
function _scaledPercentFromAmounts(
    uint256 unscaledNumeratorAmount,
    uint256 unscaledDenominatorAmount
)
    internal
    pure
    returns (uint256)
{
    return unscaledNumeratorAmount * SCALE / unscaledDenominatorAmount;
}
```

When `s_pendingTargetAtomicLiquidityPercent` equals `FLOAT_PLACEHOLDER` (which is 1) and `_totalEquity` is zero, the function enters the rebalancing logic and calls `_scaledPercentFromAmounts(oldAllocatedAmount, _totalEquity)`, causing a revert as a division by zero occurs. This also occurs other code paths when `_totalEquity` is zero.

Since `_checkSetNewAtomicLiquidityTarget()` is called from `_settleGlobalNetMONAgainstAtomicUnstaking()`, which is invoked during `_crankGlobal()`, the revert blocks epoch advancement and protocol operations.

This issue is reachable after the contracts are initially deployed, but it becomes harder to trigger once the system is live. This is because the revenue streaming logic will naturally leave residual equity behind during withdrawals. Also note that this issue could be resolved by someone transferring MON directly into the contract.

**Proof of Concept:** The following test demonstrates the issue:

```
function test_total_equity_zero() public {
    _advanceEpochAndCrank();

    address[4] memory depositors = [deployer, solverOneEOA, solverTwoEOA,
    ↪  solverThreeEOA];
    for (uint256 i; i < 4; ++i) {
        vm.startPrank(depositors[i]);
        shMonad.requestUnstake(shMonad.balanceOf(depositors[i]));
        vm.stopPrank();
    }

    // After all users request unstakes, totalEquity becomes 0
    // Next crank attempt will revert with division by zero
    _advanceEpochAndCrank(); // Reverts: panic: division or modulo by zero (0x12)
}
```

**Recommendation:** Consider adding zero checks for `_totalEquity` before performing division operations, while preserving admin configuration and pending targets. When `_totalEquity` is zero, the function may short circuit before any percentage math and treat the allocation as zero for that epoch:

```
function _checkSetNewAtomicLiquidityTarget(
    uint128 oldUtilizedAmount,
    uint128 oldAllocatedAmount
)
    internal
    returns (uint256 scaledTargetPercent, uint128 newAllocatedAmount)
{
    // Load any pending atomic liquidity percentage
    uint256 _newScaledTargetPercent = s_pendingTargetAtomicLiquidityPercent;

    // Load relevant values
    WorkingCapital memory _globalCapital = s_globalCapital;
    uint256 _totalEquity =
        _globalCapital.totalEquity(s_globalLiabilities, s_admin, address(this).balance);
    uint256 _currentAssets =
        _globalCapital.currentAssets(s_atomicAssets, address(this).balance);

    // Handle zero equity case: avoid division by zero but do not mutate admin config.
    if (_totalEquity == 0) {
        // Use the current or pending target percent only as an informational return
        ↪  value.
        uint256 effectiveScaled =
            _newScaledTargetPercent == FLOAT_PLACEHOLDER
                ? _scaledTargetLiquidityPercentage()
                : _newScaledTargetPercent;

        // With zero equity, the atomic pool allocation must also be zero.
        return (effectiveScaled, 0);
    }

    // ... rest of function logic ...
}
```

**Fastlane:** Acknowledged.

**Cantina Managed:** Acknowledged.

### 3.3.6 Placeholder validator `wasCranked` flag never set

**Severity:** Low Risk

**Context:** StakeTracker.sol#L771-L782

**Description:** The placeholder validator uses the global epoch storage to track its state. Its `wasCranked`

field is never set to `true`, even though `_crankPlaceholderValidator()` reads this value to verify it is `false`.

**Recommendation:** Ensure the placeholder validator has its `wasCranked` value set to `true` after it's cranked.

**Fastlane:** Fixed in PR 626.

**Cantina Managed:** Verified.

#### 3.3.7 Crank timing mismatch within an internal epoch

**Severity:** Low Risk

**Context:** StakeTracker.sol#L207-L224

**Description:** Validators within the same internal epoch can be cranked at significantly different times, potentially separated by multiple Monad epochs. This timing gap may lead to unexpected accounting outcomes.

For example, if the system goes down for a period of time, a validator may not be cranked until hours after the previous validator. By that time, the later validator will have accrued substantially more rewards when `_claimRewards()` runs. During the next crank, this validator will appear to have higher `earnedRevenue` and will receive a larger share of new stake purely due to the timing difference, not due to their performance.

**Recommendation:** Consider documenting this behavior.

**Fastlane:** We disagree with this finding, because the stake allocation is based on an average earnedRevenue per validator, across the last 2 epochs. So if they crank early to move earnings into the later epoch, its just reducing earnings from the prev epoch.

**Cantina Managed:** Agreed if a validator has a longer reward period in epoch `e` then it means they'll have a shorter reward period in epoch `e+1`, and this would be averaged out in epoch `e+2`.

But in epoch `e+1` it'd be using the average of `e` and `e-1`, and in epoch `e+3` it'd be using the average of `e+2` and `e+1`, so there would still be epochs where a validator gets a higher/lower % of the new stake due to timing. And since the queueToStake changes across epochs, the overall outcome is different compared to if all validators were cranked at the same time.

**Fastlane:** I think the important thing here is that cranking is permissionless and the cost is very very cheap due to the chain. If any validators feel that they'd be disadvantaged by an epoch extending they can just do the crank themselves. Similarly, the validators that are already getting a lot of stake--the ones who'd benefit from this due to having a higher revenue--are incentivized to do the crank as soon as possible so that their new stake can be processed and start earning. In essence, the cost of the delay applies to all validators and the benefit is probabalistic at best.

**Cantina Managed:** Acknowledged.

#### 3.3.8 EIP-7702 allows Coinbase EOA to turn into arbitrary code

**Severity:** Low Risk

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

**Description:** When a validator is cranked, the final step is to call `process()` on the validator's configured coinbase address, but only if that address currently has code:

```
address coinbase = _validatorCoinbase(_valId);
if (coinbase.code.length > 0) {
    try ICoinbase(coinbase).process{ gas: COINBASE_PROCESS_GAS_LIMIT }() { } catch { }
}
```

The idea of this system is that the shMonad admin configures each validator's coinbase as either their EOA or as the standard `Coinbase` contract implementation in the codebase.

However note that with EIP-7702, an EOA can essentially change into a contract. As a result, a validator with an EOA coinbase can setup arbitrary code to be executed during the crank. This expands the validator's power in a way that may be unexpected.

This issue was raised by the shMonad team during the audit.

**Recommendation:** Consider preventing this behavior by disallowing EOAs outright, or by checking that a coinbase address having code isn't due to an EIP-7702 delegation.

**Fastlane:** Fixed in PR 633 by skipping coinbase addresses that have code that begin with the EIP-7702 prefix.

**Cantina Managed:** Verified.

## 3.4 Informational

### 3.4.1 Missing NatSpec documentation, typos and other code improvement issues may affect the readability of the contracts

**Severity:** Informational

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

**Description:** Some code quality issues were identified across multiple contracts that affect documentation, code clarity, and maintainability:

1. Missing NatSpec documentation: The `requestUnstake()` function in `ShMonad.sol` lacks NatSpec documentation. This public function is part of the core unstaking functionality and should include parameter descriptions and return value documentation.
2. Comment typo: In `StakeTracker.sol` at line 270, a comment contains a typo: ”aount” should be ”amount”.
3. Unnecessary variable declaration: In `FLERC4626.sol`, the `_previewDeposit()` and `_previewMint()` functions declare a local variable `_deductRecentRevenue` set to `false` that is only used once as a function argument. This variable can be removed and `false` can be passed directly to `_convertToShares()` and `_convertToAssets()`.

**Recommendation:** It is recommended to implement the above changes in order to improve the readability and the maintainability of the contracts.

**Fastlane:** Fixed in PR 617.

**Cantina Managed:** Verified.

### 3.4.2 Missing event emission for staking commission zero-yield balance update

**Severity:** Informational

**Context:** StakeTracker.sol#L1649

**Description:** In `StakeTracker.sol`, the `_handleEarnedStakingYield()` function tracks staking commission by directly updating the owner's zero-yield balance without emitting the corresponding event. At line 1650, when `_grossStakingCommission` is added to `s_zeroYieldBalances[OWNER_COMMISSION_ACCOUNT]`, no `DepositToZeroYieldTranche` event is emitted.

```
// 1650:1650:src/shmonad/StakeTracker.sol
s_zeroYieldBalances[OWNER_COMMISSION_ACCOUNT] += _grossStakingCommission;
```

This pattern is inconsistent with the current approach used in `_depositToZeroYieldTranche()` in `ShMonad.sol`, which emits `DepositToZeroYieldTranche` whenever zero-yield balances are updated:

```
// 358:375:src/shmonad/ShMonad.sol
function _depositToZeroYieldTranche(uint256 assets, address from, address to) internal {
    AdminValues memory _admin = s_admin;

    require(msg.value == assets, IncorrectNativeTokenAmountSent());

    // Increase zero-yield tranche balance
    s_zeroYieldBalances[to] += assets;
```

```
    // Increase total liabilities to reflect the new assets held in the zero-yield
    ↪  tranche
    _admin.totalZeroYieldPayable += uint128(assets);

    // Queue up the new funds to be staked
    _accountForDeposit(assets.toUint128());

    // Persist AdminValues changes to storage
    s_admin = _admin;

    emit DepositToZeroYieldTranche(from, to, assets);
}
```

**Recommendation:** Consider emitting the `DepositToZeroYieldTranche` event when updating the owner commission zero-yield balance in `_handleEarnedStakingYield()` as this would maintain consistency with the established pattern.

**Fastlane:** Fixed in PR 618. Now we emit `DepositToZeroYieldTranche` event in `_handleEarnedStakingYield()` when staking commission is credited to the owner's zero-yield balance.

**Cantina Managed:** Verified.

### 3.4.3 Mint function requires previewMint call to determine correct msg.value

**Severity:** Informational

**Context:** FLERC4626.sol#L120-L138

**Description:** The `mint()` function in `FLERC4626` accepts a `shares` parameter as input, but requires that `msg.value` exactly matches the calculated asset amount. The function internally calls `_previewMint()` to determine the required assets based on the current exchange rate, then passes this value to `_deposit()`, which enforces `assets == msg.value`.

```
function mint(uint256 shares, address receiver) public payable virtual returns (uint256)
↪  {
    uint256 maxShares = maxMint(receiver);
    require(shares <= maxShares, ERC4626ExceededMaxMint(receiver, shares, maxShares));

    uint256 assets = _previewMint(shares, true);
    _deposit(_msgSender(), receiver, assets, shares);

    return assets;
}
```

The `_deposit()` function validates that the sent native token amount matches the calculated assets:

```
function _deposit(address caller, address receiver, uint256 assets, uint256 shares)
↪  internal virtual {
    require(assets == msg.value, IncorrectNativeTokenAmountSent());
    // ...
}
```

Because the exchange rate used by `_previewMint()` depends on the current on chain state (including total equity and supply), EOAs and users must call `previewMint()` off chain first to determine the exact `msg.value` required before executing `mint()`. This creates a two step process that may not be immediately obvious to integrators or end users.

**Recommendation:** Consider adding documentation or NatSpec comments to `mint()` that explicitly state users should call `previewMint()` first to determine the required `msg.value`.

**Fastlane:** Added NatSpec documentation in PR 621 to `mint()` function in `FLERC4626.sol` explaining that callers must call `previewMint()` off-chain first to determine the exact `msg.value` required.

**Cantina Managed:** Verified.

### 3.4.4 Staking precompile does not trigger `receive()` as expected

**Severity:** Informational

**Context:** StakeTracker.sol#L2161-L2173

**Description:** The `_claimRewards()` and `_completeWithdrawal()` functions use the `expectsStakingRewards` and `expectsUnstakingSettlement` modifiers. These modifiers set the `t_cashFlowClassifier` transient storage variables so that any incoming MON sent to the contract through `receive()` can be classified properly.

However, the Monad staking precompile does not transfer MON via an EVM call that would trigger `receive()`. Instead it updates the contract's balance at the protocol level without transferring control-flow. Because no `receive()` call occurs, these modifiers appear to have no effect and can be removed.

**Recommendation:** Consider removing the `expectsStakingRewards` and `expectsUnstakingSettlement` modifiers, and potentially remove the transient capital system altogether.

**Fastlane:** Fixed in PR 651.

**Cantina Managed:** Verified.

### 3.4.5 Incorrect index in `__StakeTracker_init()`

**Severity:** Informational

**Context:** StakeTracker.sol#L95-L101

**Description:** The `__StakeTracker_init()` function contains the following assignments:

```
globalEpochPtr_N(-2).epoch = _currentEpoch < 3 ? 0 : _currentEpoch - 3;
globalEpochPtr_N(-2).epoch = _currentEpoch < 2 ? 0 : _currentEpoch - 2;
globalEpochPtr_N(-1).epoch = _currentEpoch < 1 ? 0 : _currentEpoch - 1;
globalEpochPtr_N(0).epoch = _currentEpoch;
globalEpochPtr_N(1).epoch = _currentEpoch + 1;
globalEpochPtr_N(2).epoch = _currentEpoch + 2;
```

The first assignment appears intended for index -3, but it is written to index -2 instead. This does not appear to have any notable effect in the rest of the code.

**Recommendation:** Change the first assignment to use index -3 instead of -2.

**Fastlane:** Fixed in PR 629.

**Cantina Managed:** Verified. Also note that this code will likely never be used again.

### 3.4.6 Storage clearing happens earlier than expected

**Severity:** Informational

**Context:** StakeTracker.sol#L256-L258

**Description:** The `_crankGlobal()` function clears the `globalRevenuePtr_N` and `globalCashFlowsPtr_N` storage at index offset +2. This is unexpected because the next epoch slot that is actually about to be used is index offset +1. This does not cause issues in practice, since the storage is still cleared ahead of time, just earlier than necessary.

A similar pattern appears in `_rollValidatorEpochForwards()`, which clears `validatorRewardsPtr_N` and `validatorPendingPtr_N` at index offset +2 rather than +1.

**Recommendation:** Consider documenting this behavior or updating the code to clear index offset +1 instead.

**Fastlane:** Documented in PR 640.

**Cantina Managed:** Verified.

### 3.4.7 Validator can set `commissionRecipient` to an address that reverts

**Severity:** Informational

**Context:** Coinbase.sol#L110-L116

**Description:** The validator associated with a given Coinbase contract can update the `commissionRecipient` to any arbitrary address. This allows the validator to set the recipient to an address that will revert when `sendCommissionAndRewards()` attempts to transfer MON.

This does not appear to change the trust assumptions of the system. The `sendCommissionAndRewards()` call is wrapped in a try/catch, and the validator already has the ability to direct all MON to themselves by setting the `commissionRate` to 100% anyway.

**Recommendation:** Consider documenting this behavior for clarity.

**Fastlane:** Acknowledged.

**Cantina Managed:** Acknowledged.

### 3.4.8 Withdrawals accrue rewards not expected by the `StakeTracker`

**Severity:** Informational

**Context:** StakeTracker.sol#L1571-L1587

**Description:** When a delegator undelegates from a Monad validator, the withdrawal amount independently accrues rewards for a period of time during the withdrawal process. This additional yield is not expected in the StakeTracker logic. As a result, `_handleCompleteDecreasedAllocation()` will typically observe a surplus. This does not appear to cause issues, as the code treats the surplus as additional yield.

**Recommendation:** Consider documenting this behavior.

**Fastlane:** Documented in PR 641.

**Cantina Managed:** Verified.

### 3.4.9 Stake allocation starts very slow for new validators

**Severity:** Informational

**Context:** StakeAllocationLib.sol#L43-L96

**Description:** The shMonad system allocates new stake to validators based on their `earnedRevenue` over the past two internal epochs relative to the global `earnedRevenue`. A validator's `earnedRevenue` increases either through staking yield or rewards (e.g. MEV) sent via `sendValidatorRewards()` and the boost yield functions.

A newly added validator starts with no stake and therefore typically has little or no `earnedRevenue`. This can slow its ability to gain new deposits. These validators may have to rely on `sendValidatorRewards()` and boost yield rewards to eventually accumulate enough `earnedRevenue` to begin receiving stake.

**Recommendation:** Consider whether this behavior could cause problems. If not, consider documenting this behavior. If so, consider adding mechanisms for newly added validators to receive stake.

**Fastlane:** Acknowledged.

**Cantina Managed:** Acknowledged.

### 3.4.10 Duplicate clearing logic in validator deactivation

**Severity:** Informational

**Context:** ValidatorRegistry.sol#L320-L334

**Description:** The `_completeDeactivatingValidator()` function contains the following logic:

```
// Setting this enables future reactivation
validatorEpochPtr_N(0, validatorId).epoch = 0;

delete s_validatorIsActive[validatorId];
```

```
_unlinkValidatorCoinbase(validatorId);
delete s_validatorData[validatorId];

// Remove this validator from the link and connect together the validators on either end
_removeValidatorFromCrankSequence(validatorId);

for (uint256 i; i < EPOCHS_TRACKED; i++) {
    delete s_validatorEpoch[validatorId][i];
    delete s_validatorRewards[validatorId][i];
    delete s_validatorPending[validatorId][i];
}
```

The assignment `validatorEpochPtr_N(0, validatorId).epoch = 0` is redundant because the loop that follows clears all epoch-related storage.

**Recommendation:** Consider simplifying the function by removing the redundant storage assignment.

**Fastlane:** Fixed in PR 617.

**Cantina Managed:** Verified.

#### 3.4.11 Storage variables are hard to reason about

**Severity:** Informational

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

**Description:** The codebase uses many storage variables and mappings, and it can be hard to understand what each one is responsible for.

For example, `s_validatorData[validatorId].isActive` and `s_validatorIsActive[validatorId]` sound very similar but behave differently. During pending deactivation, only `s_validatorData[validatorId].isActive` is set to false while `s_validatorIsActive[validatorId]` stays true. A single state variable that represents active, pending, inactive states would be easier to reason about.

Another example is the reuse of the `Epoch` struct. It's used for validator epoch data (`s_validatorEpoch`) as well as global epoch data (`s_globalEpoch`). Some fields don't apply to validators, like the `frozen` or `closed` values. Also, `s_globalEpoch` is used to store placeholder validator fields such as `wasCranked`. Reusing one struct for several purposes makes it difficult to know which fields matter in which context.

**Recommendation:** Consider simplifying and separating storage structs in a future version of the code.

**Fastlane:** Acknowledged.

**Cantina Managed:** Acknowledged.

#### 3.4.12 `calculateDeactivatedValidatorEpochStakeDelta()` is unused

**Severity:** Informational

**Context:** StakeAllocationLib.sol#L119-L131

**Description:** The `calculateDeactivatedValidatorEpochStakeDelta()` is defined in `StakeAllocationLib` but it is not used.

**Recommendation:** Consider removing the `calculateDeactivatedValidatorEpochStakeDelta()` function.

**Fastlane:** Removed in PR 631.

**Cantina Managed:** Verified.

#### 3.4.13 Missing reentrancy guards on external MONAD transfer functions

**Severity:** Informational

**Context:** Coinbase.sol#L110-L113

**Description:** Two functions perform external ETH transfers without reentrancy guards, which may create opportunities for reentrancy attacks if the codebase evolves or gas limits are modified.

In the `process()` function in `Coinbase`, the `sendCommissionAndRewards()` function calls `trySafeTransferETH()` to send validator commission to a recipient. This function is invoked during `_crankValidator()` in `StakeTracker`, which does not have reentrancy protection. While the external call is limited by `TRANSFER_GAS_LIMIT` (100,000) and `COINBASE_PROCESS_GAS_LIMIT` (500,000), making reentrancy unlikely in the current implementation, the lack of explicit guards creates a potential risk if these gas limits may be increased in the future.

In the `completeUnstake()` function in `ShMonad`, the function performs a `safeTransferETH()` call to send unstaked funds to the user account. The transfer occurs after state updates (deleting the unstake request and calling `_beforeCompleteUnstake()`), which mitigates immediate reentrancy concerns.

While neither function appears exploitable in the current implementation, adding reentrancy guards would provide protection against future code changes.

**Recommendation:** Consider adding `nonReentrant` modifiers to both functions to provide explicit reentrancy protection.

**Fastlane:** Fixed in PR 638 and PR 642.

**Cantina Managed:** Verified.

#### 3.4.14 Validator epoch roll copies unused fields

**Severity:** Informational

**Context:** StakeTracker.sol#L806-L819

**Description:** The `_rollValidatorEpochForwards()` function sets the next validator epoch storage at offset +1 and copies the `frozen` and `closed` fields from the current validator epoch into it. However, these fields are not used for validator-level storage and will always remain `false`, so copying them has no effect.

**Recommendation:** Consider removing the logic that copies the `frozen` and `closed` fields, and consider simplifying the storage structs.

**Fastlane:** Acknowledged. We are reusing the placeholder Validator to track the global epoch. This inefficiency is by design.

**Cantina Managed:** Acknowledged.

#### 3.4.15 Inaccurate comment above `_settleEarnedStakingYield()`

**Severity:** Informational

**Context:** StakeTracker.sol#L715-L716

**Description:** Above the logic in the validator crank that calls `_settleEarnedStakingYield()`, the code includes the following comment:

```
// Pull validator rewards (net of commission) so rebalancing reflects latest earnings.
_settleEarnedStakingYield(_valId);
```

This comment may be misleading. The rewards pulled in by `_settleEarnedStakingYield()` update the validator's `earnedRevenue` at index offset 0, but stake allocation is based on `earnedRevenue` at offsets -1 and -2. Therefore, this function should not affect the upcoming rebalancing.

**Recommendation:** Consider updating this comment.

**Fastlane:** Fixed in PR 625.

**Cantina Managed:** Verified.

### 3.4.16 Unattributed MON is intentionally excluded from global `earnedRevenue`

**Severity:** Informational

**Context:** StakeTracker.sol#L1212-L1227

**Description:** There are some locations in the code (e.g. `_handleValidatorRewards()`, `_handleBoostYield()`) where incoming MON is intentionally not added to the global `earnedRevenue` because the revenue cannot be attributed to a validator within shMonad. This decision was made because adding revenue globally without assigning it to a validator would dilute the stake distribution in the next crank.

Note that one side effect of this behavior is that the incoming MON will not go through the revenue smoothing logic, and so it will increase the shMON exchange rate immediately.

**Recommendation:** No changes appear necessary for this behavior. Keep this behavior in mind.

**Fastlane:** Acknowledged.

**Cantina Managed:** Acknowledged.

### 3.4.17 Initialization pattern is difficult to reason about

**Severity:** Informational

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

**Description:** The current `initialize()` function is annotated with `reinitializer(10)`. This is because the contract has already gone through deployment and multiple upgrades, and the same `initialize()` function has been reused. This can make it hard to reason about the upgrade history, since the `initialize()` function has changed over time.

A simpler pattern would be to introduce a new function for each upgrade, for example `initialize()` for the original deployment, `initializeV2()` for the first upgrade, etc., each with its own `initializer` or `reinitializer` guard. This makes it easier to audit and to replay initializations in order in a fresh deployment, since each function represents a specific upgrade step.

**Recommendation:** Consider simplifying the initialize logic in future versions of the code.

**Fastlane:** Acknowledged.

**Cantina Managed:** Acknowledged.