

FASTLANE · SECURITY REVIEW

Fastlane ShMonad Updates.

A security review of ShMonad Updates.

CLIENT	PROJECT	AUDITORS	AUDIT WINDOW	FINAL REPORT
Fastlane	ShMonad Updates	0xCiphky ., Michael Lett, Minato7namikazi	April 30 – August 21, 2026	August 21, 2026

§ 00

Contents.

230 pages: overview, confidence ranking, scope, 118 findings, disclaimer, and about Guardian.

§ 01	Overview	P. 03
§ 02	Confidence Ranking	P. 04
§ 03	Scope & Classifications	P. 05
	Files in Scope	P. 06
§ 04	Findings	P. 09
	Main Review 1 · 38 findings	P. 14
	Remediation Review 1 · 12 findings	P. 89
	Remediation Review 2 · 13 findings	P. 107
	Main Review 2 · 49 findings	P. 127
	Remediation Review 3 · 5 findings	P. 218
	Remediation Review 4 · 1 finding	P. 227
§ 05	Disclaimer	P. 229
§ 06	About Guardian	P. 230

Overview.

Guardian reviewed ShMonad Updates across 6 review rounds. The final report includes 4 High, 65 Medium, 42 Low, 7 Informational findings.

AUDIT FIRM	REPOSITORY
Guardian	https://github.com/FastLane-Labs/fastlane-contracts-internal
CLIENT	METHODOLOGY
Fastlane	Smart contract · Manual · Main Review 1 · Remediation Review 1 · Remediation Review 2 · Main Review 2 · Remediation Review 3 · Remediation Review 4
PROJECT	MAIN REVIEW 1 COMMIT
ShMonad Updates	7910434ed048
PREPARED BY	REMEDATION REVIEW 1 COMMIT
OxCiphky ., Michael Lett, Minato7namikazi	14e40ee4491e
AUDIT WINDOW	REMEDATION REVIEW 2 COMMIT
April 30 – August 21, 2026	cd406b6f6127
FINAL REPORT	MAIN REVIEW 2 COMMIT
August 21, 2026	9f116de99a16

1.1 Confidence Ranking

3 / 5

Moderate Confidence

Guardian assigns a Confidence Ranking of 3 out of 5, combining finding severity, remediation outcome, and the reviewer’s assessment of latent risk. The full scale is on the next page.

1	2	3	4	5
VERY LOW CONFIDENCE	LOW CONFIDENCE	MODERATE CONFIDENCE	HIGH CONFIDENCE	VERY HIGH CONFIDENCE

1.2 Findings at a Glance

SEVERITY	TOTAL	PENDING	ACK	PARTIAL	RESOLVED	FIX REVIEW
Critical	0	0	0	0	0	0
High	4	2	0	0	0	2
Medium	65	21	11	1	22	10
Low	42	12	10	1	9	10
Informational	7	3	0	0	0	4
§01 - OVERVIEW				CONFIDENTIAL		P. 03 / 230
All severities	118	38	21	2	31	26

§ 02

Confidence Ranking.

Guardian assigns a project-level confidence score using the scale below. The ranking combines finding severity, remediation outcome, and the reviewer's assessment of latent risk.

CONFIDENCE RANKING	DEFINITION AND RECOMMENDATION	RISK PROFILE
5 Very High Confidence	Codebase is mature, clean, and secure. No High or Critical vulnerabilities were found. Follows modern best practices with high test coverage and thoughtful design. Recommendation: code is highly secure at time of audit.	0 High/Critical findings and few Low/Medium severity findings.
4 High Confidence	Code is clean, well-structured, and adheres to best practices. Design patterns are sound and test coverage is strong. Recommendation: suitable for deployment after remediations; consider periodic review with changes.	0-1 High/Critical findings per engagement week and little to no Medium severity findings.
3 Moderate Confidence ASSIGNED	Code is functional but shows gaps in defensive programming or test coverage. Recommendation: remediate the findings and re-review the changes before deployment.	Some High/Critical findings, with various Medium severity findings.
2 Low Confidence	Recurring issues suggest systemic gaps rather than isolated mistakes. Recommendation: remediate, then commission a follow-up review at a frozen commit.	Over 6 High and Critical findings across the engagement.
1 Very Low Confidence	The volume and severity of findings indicate the codebase is not ready for production. Recommendation: substantial rework, then a full re-review.	Over 12 High and Critical findings across the engagement.

§ 03

Scope & Classifications.

Guardian reviewed ShMonad Updates in <https://github.com/FastLane-Labs/fastlane-contracts-internal>. The scope includes 19,750 source lines across 29 files.

3.1 Severity Matrix

	IMP · HIGH	IMP · MED	IMP · LOW
LIK · HIGH	Critical	High	Medium
LIK · MED	High	Medium	Low
LIK · LOW	Medium	Low	Low

IMPACT · HIGH

Significant loss of assets, harm to a group of users, or core protocol functionality is disrupted.

IMPACT · MEDIUM

Small amount of funds can be lost, or ancillary functionality is affected. Users may experience reduced or delayed receipt.

IMPACT · LOW

Unexpected behavior in some functionality that is notable but does not meet the criteria for higher severity.

LIKELIHOOD · HIGH

Possible with reasonable assumptions that mimic on-chain conditions; cost is low compared to amount gained.

LIKELIHOOD · MEDIUM

Possible only in uncommon cases or requires a large amount of capital relative to disruption caused.

LIKELIHOOD · LOW

Unlikely to ever occur in production.

Files in Scope 1/3.

FILE	NSLOC	TOTAL
MAIN REVIEW 1		
script/deploy-shmonad-implementation.s.sol	26	45
script/safe/deploy-hmon-safe.s.sol	94	131
script/safe/deploy-shmonad-safe.s.sol	215	308
script/safe/prepare-shmonad-upgrade.s.sol	208	243
script/timelock/cancel-timelock-operation.s.sol	76	105
script/timelock/deploy-timelock.s.sol	84	120
script/timelock/execute-timelock-upgrade.s.sol	31	52
script/timelock/schedule-beacon-upgrade-through-timelock.s.sol	115	145
script/timelock/schedule-upgrade-through-timelock.s.sol	130	162
script/timelock/transfer-ownership-to-timelock.s.sol	77	104
src/shmonad/Coinbase.sol	316	447
src/shmonad/CoinbaseUpgradeable.sol	404	633
src/shmonad/Constants.sol	58	92
src/shmonad/Errors.sol	84	211
src/shmonad/Events.sol	197	310
src/shmonad/FLERC4626.sol	275	531
src/shmonad/HolisticMON.sol	129	239
src/shmonad/libraries/AccountingLib.sol	30	232
src/shmonad/libraries/CircuitBreakerLib.sol	9	16
src/shmonad/libraries/QueueLib.sol	100	147
src/shmonad/libraries/StakeAllocationLib.sol	113	170
src/shmonad/libraries/StorageLib.sol	33	44
src/shmonad/libraries/TradUnstakeLib.sol	150	210
src/shmonad/PrecompileHelpers.sol	232	295
src/shmonad/ShMonad.sol	204	509
src/shmonad/StakeTracker.sol	2,022	3,352
src/shmonad/Storage.sol	285	513
src/shmonad/Types.sol	178	232
src/shmonad/ValidatorRegistry.sol	529	816
REMEDIATION REVIEW 1		
script/deploy-shmonad-implementation.s.sol	26	45
script/safe/deploy-hmon-safe.s.sol	94	131
script/safe/deploy-shmonad-safe.s.sol	215	308
script/safe/prepare-shmonad-upgrade.s.sol	208	243
script/timelock/cancel-timelock-operation.s.sol	76	105

Files in Scope 2/3.

FILE	NSLOC	TOTAL
script/timelock/deploy-timelock.s.sol	84	120
script/timelock/execute-timelock-upgrade.s.sol	31	52
script/timelock/schedule-beacon-upgrade-through-timelock.s.sol	115	145
script/timelock/schedule-upgrade-through-timelock.s.sol	130	162
script/timelock/transfer-ownership-to-timelock.s.sol	77	104
src/shmonad/Coinbase.sol	316	447
src/shmonad/CoinbaseUpgradeable.sol	442	686
src/shmonad/Constants.sol	57	91
src/shmonad/Errors.sol	90	221
src/shmonad/Events.sol	195	308
src/shmonad/FLERC4626.sol	276	535
src/shmonad/HolisticMON.sol	129	239
src/shmonad/libraries/AccountingLib.sol	30	232
src/shmonad/libraries/CircuitBreakerLib.sol	13	21
src/shmonad/libraries/QueueLib.sol	169	230
src/shmonad/libraries/StakeAllocationLib.sol	190	259
src/shmonad/libraries/StorageLib.sol	33	44
src/shmonad/libraries/TradUnstakeLib.sol	151	208
src/shmonad/PrecompileHelpers.sol	201	257
src/shmonad/ShMonad.sol	208	526
src/shmonad/StakeTracker.sol	2,084	3,413
src/shmonad/Storage.sol	264	474
src/shmonad/Types.sol	186	241
src/shmonad/ValidatorRegistry.sol	531	821

REMEDIATION REVIEW 2

script/deploy-shmonad-implementation.s.sol	26	45
script/safe/deploy-hmon-safe.s.sol	94	131
script/safe/deploy-shmonad-safe.s.sol	215	308
script/safe/prepare-shmonad-upgrade.s.sol	208	243
script/timelock/cancel-timelock-operation.s.sol	76	105
script/timelock/deploy-timelock.s.sol	84	120
script/timelock/execute-timelock-upgrade.s.sol	31	52
script/timelock/schedule-beacon-upgrade-through-timelock.s.sol	115	145
script/timelock/schedule-upgrade-through-timelock.s.sol	130	162
script/timelock/transfer-ownership-to-timelock.s.sol	77	104
src/shmonad/Coinbase.sol	315	446

Files in Scope 3/3.

FILE	NSLOC	TOTAL
src/shmonad/CoinbaseUpgradeable.sol	467	723
src/shmonad/Constants.sol	57	91
src/shmonad/Errors.sol	92	225
src/shmonad/Events.sol	199	314
src/shmonad/FLERC4626.sol	263	527
src/shmonad/HolisticMON.sol	129	239
src/shmonad/libraries/AccountingLib.sol	30	232
src/shmonad/libraries/CircuitBreakerLib.sol	13	21
src/shmonad/libraries/QueueLib.sol	169	230
src/shmonad/libraries/StakeAllocationLib.sol	186	255
src/shmonad/libraries/StorageLib.sol	33	44
src/shmonad/libraries/TradUnstakeLib.sol	151	208
src/shmonad/PrecompileHelpers.sol	204	261
src/shmonad/ShMonad.sol	217	536
src/shmonad/StakeTracker.sol	2,148	3,526
src/shmonad/Storage.sol	266	476
src/shmonad/Types.sol	187	242
src/shmonad/ValidatorRegistry.sol	543	842

§ 04

Findings.

Grouped by round. Visible IDs reset by severity within each round, so a finding is named the same way in the report as it was in the channel it was written in. Status is where each finding ended up; its own page says how.

ID	TITLE	CATEGORY	SEVERITY	STATUS	PAGE
MAIN REVIEW 1					
H-01	Turnover Can Pin Validator Crank	DoS	HIGH	PENDING	p. 14
H-02	Stale denominator amplifies turnover	Logical Error	HIGH	PENDING	p. 16
M-01	Isolated Active-Stake Deficits Do Not Haircut Locked Redemptions	Logical Error	MEDIUM	PENDING	p. 19
M-02	Atomic Reserve Leaves Queue Live	Logical Error	MEDIUM	PENDING	p. 21
M-03	Stake allocation is still dominated by yield	Math	MEDIUM	PENDING	p. 23
M-04	Pending Delta Masks Stake Deficit	Logical Error	MEDIUM	PENDING	p. 26
M-05	Revenue Double-Books Atomic Liquidity	Logical Error	MEDIUM	PENDING	p. 28
M-06	Pending Sync Misattributes Stake	Logical Error	MEDIUM	PENDING	p. 31
M-07	Post-Roll Coinbase Rewrites Split	Logical Error	MEDIUM	PENDING	p. 33
M-08	Isolated Surplus Bypasses Yield Fees	Logical Error	MEDIUM	PENDING	p. 35
M-09	Zero Equity Bricks Global Crank	DoS	MEDIUM	PENDING	p. 37
M-10	Deficit Can Lose Unstake Demand	Logical Error	MEDIUM	PENDING	p. 39
M-11	Shortfall can leave excess isolated liq idle	Logical Error	MEDIUM	PENDING	p. 41
M-12	Isolated shortfall allocation depends on timing	Rewards	MEDIUM	PENDING	p. 43
M-13	Atomic reserve fills leave duplicate queue	Logical Error	MEDIUM	PENDING	p. 45
M-14	Queue netting consumes turnover first	Logical Error	MEDIUM	PENDING	p. 47
M-15	Deficit Can Lose Unstake Demand	Logical Error	MEDIUM	PENDING	p. 50
M-16	Split Exits Bypass Delay	Logical Error	MEDIUM	PENDING	p. 52
M-17	Excess netting underrepresents redemptions	Math	MEDIUM	PENDING	p. 54
M-18	Boosted high-rate state can make large deposits fail the share-rate breaker	Math	MEDIUM	PENDING	p. 57
M-19	Priority Fees Become Staking Yield	Logical Error	MEDIUM	PENDING	p. 58
M-20	Coinbase Unpaid Rewards Can Strand	DoS	MEDIUM	PENDING	p. 60
M-21	Pending isolated stake dilutes pool yield	Rewards	MEDIUM	PENDING	p. 62

Findings continued.

ID	TITLE	CATEGORY	SEVERITY	STATUS	PAGE
L-01	previewRequestUnstake Can Return a Success Quote for Requests That Later Revert on Slot Exhaustion	Best Practices	LOW	PENDING	p. 64
L-02	New Isolated Depositors Can Capture Surplus From Prior Pending Isolated Withdrawals	Best Practices	LOW	PENDING	p. 66
L-03	Stale externalReward Gas Constant	Configuration	LOW	PENDING	p. 68
L-04	Redundant second scan: unstake completion	Gas Optimization	LOW	PENDING	p. 69
L-05	Allocation split start semantics	Validation	LOW	PENDING	p. 70
L-06	Atomic unstake fees are immediately capturable by JIT deposits	MEV	LOW	PENDING	p. 72
L-07	UNKNOWN/inactive-proposer boost yield bypasses smoothing and enables JIT capture	Math	LOW	PENDING	p. 73
L-08	Validator crank can skip global epoch rolls	Math	LOW	PENDING	p. 75
L-09	Isolated deposit can mint zero shares	MEV	LOW	PENDING	p. 77
L-10	Isolated unstake can leave dust position	Logical Error	LOW	PENDING	p. 79
L-11	Late Isolated Deposit Captures Past Rewards	Logical Error	LOW	PENDING	p. 81
L-12	Pooled rate breaker has coverage gaps	Best Practices	LOW	PENDING	p. 83
I-01	recomputeCurrentAllocationTotals — validator cap not enforced at upgrade time	Warning	INFORMATIONAL	PENDING	p. 85
I-02	Informationals about Upgrade Plan	Best Practices	INFORMATIONAL	PENDING	p. 86
I-03	Deactivated Validator and Stake Allocation	Informational	INFORMATIONAL	PENDING	p. 88

REMEDIATION REVIEW 1

H-01	Sub-Dust Isolated Exit Freezes Protocol	DoS	HIGH	REPORTED	p. 89
M-01	Reward Retry Reuses Settled Buckets	Logical Error	MEDIUM	REPORTED	p. 91
M-02	Atomic Completion Leaves Paid Queue Live	Logical Error	MEDIUM	REPORTED	p. 93
M-03	Atomic Target Reuses Revenue Carryover	Logical Error	MEDIUM	REPORTED	p. 94
M-04	Inactive validator rewards bypass smoothing	MEV	MEDIUM	REPORTED	p. 96
M-05	Raw Breaker Misses Smoothed Rate Drop	Configuration	MEDIUM	REPORTED	p. 98
L-01	Hardcoded staking deadband limits	Configuration	LOW	REPORTED	p. 100
L-02	Delay Crank Skips Global Epoch Roll	Logical Error	LOW	REPORTED	p. 101
L-03	Dust Unstake Residual Drops Redemption Queue	Logical Error	LOW	REPORTED	p. 103
L-04	Onboarding Isolated Yield Routes Pooled	Rewards	LOW	REPORTED	p. 104
L-05	Reused Coinbase Keeps Stale Auth	Logical Error	LOW	REPORTED	p. 105
I-01	Gas-Dependent Coinbase Donation Attribution	Logical Error	INFORMATIONAL	REPORTED	p. 106

Findings continued.

ID	TITLE	CATEGORY	SEVERITY	STATUS	PAGE
REMEDIATION REVIEW 2					
H-01	Pooled unstake accounting can freeze protocol	DoS	HIGH	REPORTED	p. 107
M-01	Queue netting creates synthetic goodwill	Math	MEDIUM	REPORTED	p. 109
M-02	Atomic rebalance can reuse reserved revenue	Math	MEDIUM	REPORTED	p. 110
M-03	Inactive validator can affect allocation scoring	Math	MEDIUM	REPORTED	p. 112
M-04	Deadband-adjusted revenue can block allocations	Math	MEDIUM	REPORTED	p. 114
L-01	JIT Deposit Captures Orphan Sweep	Logical Error	LOW	REPORTED	p. 116
L-02	Inactive Reward Redirect Skips Smoothing	Logical Error	LOW	REPORTED	p. 117
L-03	MEV Commission Reprices Escrow	Logical Error	LOW	REPORTED	p. 119
L-04	Unpaid Rewards Skip Smoothing	MEV	LOW	REPORTED	p. 120
L-05	Share-Burn Boosts Skip Smoothing	MEV	LOW	REPORTED	p. 122
I-01	Forced Finalizers Forward All Gas	Gas Griefing	INFORMATIONAL	REPORTED	p. 124
I-02	Late Cranks Extend Revenue Smoothing	Informational	INFORMATIONAL	REPORTED	p. 125
I-03	Placeholder Event Reads New Epoch	Events	INFORMATIONAL	REPORTED	p. 126
MAIN REVIEW 2					
M-01	Onboarding Rewards Are Orphaned	Logical Error	MEDIUM	RESOLVED	p. 127
M-02	Late Cranks Mis-Split Rewards	Logical Error	MEDIUM	PARTIALLY RESOLVED	p. 128
M-03	Pooled Surplus Credits Isolated Holders	Logical Error	MEDIUM	RESOLVED	p. 130
M-04	Readded validator books old stake twice	Validation	MEDIUM	RESOLVED	p. 131
M-05	Atomic allocation shrink reprices withdrawals	Logical Error	MEDIUM	RESOLVED	p. 133
M-06	Pooled Pending Masks Isolated Stake	Rewards	MEDIUM	RESOLVED	p. 135
M-07	Mutable Signal Pins Crank	DoS	MEDIUM	RESOLVED	p. 137
M-08	Coinbase spoofing redirects validator rewards	Logical Error	MEDIUM	ACKNOWLEDGED	p. 139
M-09	Queued validator rewards dilute existing holders	Logical Error	MEDIUM	ACKNOWLEDGED	p. 141
M-10	Atomic withdrawals leak deferred holder yield	Logical Error	MEDIUM	RESOLVED	p. 143
M-11	Inactive cache misroutes active validator payout	Logical Error	MEDIUM	RESOLVED	p. 144
M-12	Queued unstake makes delay branch unreachable	Logical Error	MEDIUM	RESOLVED	p. 147
M-13	Reward split ignores pre exit stake mix	Logical Error	MEDIUM	RESOLVED	p. 149
M-14	Rolling pooled pending locks isolated redemption	Logical Error	MEDIUM	RESOLVED	p. 152

Findings continued.

ID	TITLE	CATEGORY	SEVERITY	STATUS	PAGE
M-15	Stale Cache Redirects Validator Rewards	Rewards	MEDIUM	RESOLVED	p. 155
M-16	Dust Sweep Strands Isolated Claims	Logical Error	MEDIUM	RESOLVED	p. 157
M-17	Boundary window permits unpaid reward diversion	Validation	MEDIUM	RESOLVED	p. 159
M-18	Temp capital creates permanent validator score	Logical Error	MEDIUM	ACKNOWLEDGED	p. 161
M-19	Unstake burns erase deferred reward claims	Logical Error	MEDIUM	ACKNOWLEDGED	p. 163
M-20	Failed rewards ignore validator lifecycle change	Logical Error	MEDIUM	ACKNOWLEDGED	p. 164
M-21	Coinbase auth bypasses reward snapshot ordering	Logical Error	MEDIUM	ACKNOWLEDGED	p. 166
M-22	Tiny validator slices stall redemptions	Logical Error	MEDIUM	ACKNOWLEDGED	p. 168
M-23	Final isolated sweep rewards late minter	Logical Error	MEDIUM	ACKNOWLEDGED	p. 170
M-24	Forced exit retries duplicate unstake demand	Logical Error	MEDIUM	RESOLVED	p. 172
M-25	Global rolls erase unvested revenue protection	Logical Error	MEDIUM	RESOLVED	p. 174
M-26	Recycled rewards collapse isolated shares	Logical Error	MEDIUM	ACKNOWLEDGED	p. 176
M-27	Pending isolated stake distorts allocation	Logical Error	MEDIUM	ACKNOWLEDGED	p. 178
M-28	withdrawal ownership erased during cleanup	Logical Error	MEDIUM	RESOLVED	p. 180
M-29	Zero snapshot misroutes pooled validator stake	Logical Error	MEDIUM	RESOLVED	p. 182
M-30	Reward Claims Mask Validator Exit	Rewards	MEDIUM	RESOLVED	p. 184
L-01	Frozen vault allows stale priced atomic exits	Logical Error	LOW	ACKNOWLEDGED	p. 186
L-02	Revoked validator auth still settles rewards	Validation	LOW	RESOLVED	p. 188
L-03	Public isolated claim can strand principal	Logical Error	LOW	ACKNOWLEDGED	p. 190
L-04	Raw signatures bypass ERC1271 permit checks	Validation	LOW	RESOLVED	p. 192
L-05	Zero snapshots credit attacker isolated stake	Logical Error	LOW	RESOLVED	p. 193
L-06	Utilized atomic pool prevents target shutdown	Logical Error	LOW	RESOLVED	p. 195
L-07	Small slope collapses atomic withdrawal fees	Math	LOW	ACKNOWLEDGED	p. 197
L-08	Queue planner preserves unrelated unstake demand	Logical Error	LOW	RESOLVED	p. 199
L-09	hMON Pending Stake Delays Pooled Exits	Logical Error	LOW	ACKNOWLEDGED	p. 201
L-10	Closed Vault Advertises Deposits	Unexpected Behavior	LOW	RESOLVED	p. 203
L-11	Deactivated Validator Duplicates Stake Queue	Logical Error	LOW	RESOLVED	p. 204
L-12	Toggle Reprices Historical Yield	Unexpected Behavior	LOW	ACKNOWLEDGED	p. 205
L-13	Pending Exit Inflates Yield Signal	Rewards	LOW	ACKNOWLEDGED	p. 206

Findings continued.

ID	TITLE	CATEGORY	SEVERITY	STATUS	PAGE
L-14	Early Activation States hMON History	Events	LOW	RESOLVED	p. 207
L-15	Fail stop state lacks recovery transition	Logical Error	LOW	ACKNOWLEDGED	p. 209
L-16	Late rewards buy closed allocation rounds	Validation	LOW	ACKNOWLEDGED	p. 210
L-17	ShMonad strikes ignore native epoch identity	Logical Error	LOW	PARTIALLY RESOLVED	p. 212
L-18	Isolated withdrawals can strand ownerless MON	Logical Error	LOW	RESOLVED	p. 214
L-19	Cursor lag prematurely finalizes pending deposit	Logical Error	LOW	ACKNOWLEDGED	p. 216

REMEDIATION REVIEW 3

M-01	Shared Epoch Prematurely Matures Deposits	Rewards	MEDIUM	RESOLVED	p. 218
M-02	Validator exit forgets queued withdrawal portion	Logical Error	MEDIUM	RESOLVED	p. 220
M-03	Mature withdrawals lose priority against atomics	Logical Error	MEDIUM	RESOLVED	p. 222
M-04	Sponsored operation can overcharge bonded stake	Validation	MEDIUM	REPORTED	p. 224
L-01	Dust Scores Dilute Stake Allocation	Logical Error	LOW	ACKNOWLEDGED	p. 225

REMEDIATION REVIEW 4

M-01	Unclaimed priority rewards underprice new deposits	Logical Error	MEDIUM	ACKNOWLEDGED	p. 227
------	--	---------------	--------	--------------	--------

HIGH **H-01**

PENDING

Turnover Can Pin Validator Crank

CATEGORY	DoS
LOCATION	https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/7910434ed048d9da1019b04e5684443e9c08ab4b/src/shmonad/StakeTracker.sol#L1424
REVIEW	Main Review 1

— DESCRIPTION

When incentive-alignment turnover routes stake away from an overallocated validator, the computed turnover withdrawal can exceed that validator's currently unstakable stake.

`StakeAllocationLib.calculateValidatorEpochStakeDelta()` caps normal non-turnover withdrawals by `validatorAmountAvailableToUnstake`, but turnover withdrawals are capped only by the validator's target stake and overallocation.

Later, `_crankValidator()` enters the forced-exit branch in either of two cases: the validator is inactive/deactivating, or the computed next target falls below `MIN_VALIDATOR_DEPOSIT`. That branch assumes `_netAmount <= _validatorUnstakableAmount` and computes `_validatorUnstakableAmount - uint256(_netAmount)` before overriding `_netAmount` to the available unstakable amount. If turnover made `_netAmount` larger than the currently unstakable stake, this subtraction underflows and `crank()` reverts. There are two realistic trigger paths. First, an active validator can have target stake slightly above `MIN_VALIDATOR_DEPOSIT`, most of that target still pending staking, small positive weighted revenue, and a larger turnover withdrawal because it is overallocated. The positive revenue makes `stakeAllocationIncrease > 0`, entering the vulnerable carryover branch, while the turnover withdrawal can push the computed `nextTargetStakeAmount` below `MIN_VALIDATOR_DEPOSIT`.

Second, the same underflow can occur for an inactive or deactivating validator. In that case, `_crankValidator()` enters the forced-exit branch regardless of whether `nextTargetStakeAmount` is below the minimum. If the validator has pending stake, `targetStakeAmount` can still exceed `validatorUnstakableAmount`, and a turnover-heavy computed withdrawal can again make `_netAmount > _validatorUnstakableAmount`.

Because the revert happens before the validator cursor is advanced, the same validator remains `s_nextValidatorToCrank`. Subsequent `crank()` calls keep reaching the same underflow and revert, so the validator round cannot complete. Since global epoch advancement requires the validator round to finish, `ShMonad` cannot advance normally while this state persists.

— RECOMMENDATION

H-01 Turnover Can Pin Validator Crank

CONTINUED

Cap turnover withdrawals by the validator's currently unstakable capacity, not only by target stake / overallocation, and requeue any unsatisfied turnover amount with the correct label. Also make the forced-exit carryover subtraction saturating or branch on `netAmount >= validatorUnstakableAmount` before computing `maxDelta`.

PROOF OF CONCEPT <https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/82e3fa1de75ab5f16f9a4ac96450430ce0462894/test/shmonad/TurnoverForcedExitAudit.t.sol#L12>

— RESOLUTION

Status: Pending.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/742>

HIGH **H-02**

PENDING

Stale denominator amplifies turnover

CATEGORY Logical Error
 LOCATION src/shmonad/StakeTracker.sol#L646
 REVIEW Main Review 1

DESCRIPTION

`_recomputeCurrentAllocationTotals` caches `totalPooledTargetStake` and `totalOverallocatedStake` for the upcoming validator crank round. However, it computes those totals from `validatorEpochPtr_N(-1)` before the global epoch pointer advances.

```
``solidity
totalPooledTargetStake += validatorEpochPtr_N(-1, cursorId).targetStakeAmount;
`
```

Later in the same global crank, `_advanceEpochPointersAndResetValidatorCursor()` increments `s_admin.internalEpoch`. During the following validator crank, `_computeStakeDelta` reads each validator's target from `validatorEpochPtr_N(-1)`, which now corresponds to the pre-advance `N(0)` slot.

As a result, turnover routing can combine:

- a cached `totalOverallocatedStake` denominator computed from pre-advance `N(-1)`, and
- per-validator `overallocatedStake` numerators computed from post-advance `N(-1)`, i.e. pre-advance `N(0)`.

This mixes different target-stake epochs in the same formula:

```
``solidity
turnoverAllocationDecrease =
uint256(turnoverQueueForUnstake_Last).mulDivUp(overallocatedStake, totalOverallocatedStake);
``
```

If the stale denominator is smaller than the current aggregate overallocated stake, turnover can be amplified beyond the intended turnover queue.

Example

Assume the protocol is globally cranking from epoch 5 into epoch 6.

Before the global pointer advances:

```
``plain text
N(0) = epoch 5 validator target
N(-1) = epoch 4 validator target
``
```

H-02 Stale denominator amplifies turnover

CONTINUED

`_recomputeCurrentAllocationTotals` currently caches totals using epoch 4 targets. After the pointer advances, the validator crank uses:

```
``plain text
```

```
post-advance N(-1) = pre-advance N(0) = epoch 5 validator target
```

```
`
```

So the denominator can be based on epoch 4 while the validator numerator is based on epoch 5.

Consider a less extreme drift:

```
``plain text
```

```
turnoverQueueForUnstake = 40 MON
```

```
cached totalOverallocated = 100 MON // computed from epoch 4 targets
```

```
current overallocated stake:
```

```
validator A = 80 MON
```

```
validator B = 70 MON
```

```
validator C = 50 MON
```

```
actual current total = 200 MON // based on epoch 5 targets
```

```
``
```

Because each validator's current overallocated stake is divided by the stale 100 MON denominator, the 40 MON turnover queue is over-assigned:

```
``plain text
```

```
A turnover decrease = 40 * 80 / 100 = 32 MON
```

```
B turnover decrease = 40 * 70 / 100 = 28 MON
```

```
C turnover decrease = 40 * 50 / 100 = 20 MON
```

```
Total assigned turnover = 80 MON
```

```
``
```

The intended turnover budget was 40 MON, but validators collectively receive 80 MON of turnover withdrawal pressure. With a correctly aligned denominator of 200 MON, the assignments would be:

```
``plain text
```

```
A turnover decrease = 40 * 80 / 200 = 16 MON
```

```
B turnover decrease = 40 * 70 / 200 = 14 MON
```

```
C turnover decrease = 40 * 50 / 200 = 10 MON
```

```
Total assigned turnover = 40 MON
```

```
...
```

So a normal-looking stale denominator can double the intended turnover withdrawal amount.

Impact

The issue can over- or under-route turnover withdrawals because the numerator and

H-02 Stale denominator amplifies turnover

CONTINUED

denominator are derived from inconsistent target-stake epochs.

In states where the stale denominator is smaller than current over-allocation, turnover can be amplified beyond the configured queue and cause excessive validator deallocation. This can produce large unstake/restake churn, delayed allocation convergence, and distorted validator incentives. In more extreme edge cases, the amplification can drive a validator below `MIN_VALIDATOR_DEPOSIT` and trigger forced-exit behavior.

— RECOMMENDATION

Compute the cached allocation totals from the same validator target epoch that the following validator crank will use.

Because `_recomputeCurrentAllocationTotals` runs before the global pointer advances, it should likely use `validatorEpochPtr_N(0)` for target stake inputs:

```
``diff
- validatorEpochPtr_N(-1, cursorId).targetStakeAmount
+ validatorEpochPtr_N(0, cursorId).targetStakeAmount
\
and:
`diff
- StakeAllocationLib.calculateOverallocatedStake(validatorEpochPtr_N(-1, cursorId), ...)
+ StakeAllocationLib.calculateOverallocatedStake(validatorEpochPtr_N(0, cursorId), ...)
``
```

The revenue windows should remain aligned with the intended lagged allocation windows; the issue is the target-stake epoch used for the cached stake denominators.

— RESOLUTION

Status: Pending.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/747>

MEDIUM **M-01**

PENDING

Isolated Active-Stake Deficits Do Not Haircut Locked Redemptions

CATEGORY Logical Error
LOCATION StakeTracker.sol:2457
REVIEW Main Review 1

DESCRIPTION

When `_handleStakeBalanceVerification()` detects an unexpected active-stake deficit, the isolated portion only reduces `isolatedState.activeStake` and `s_isolatedTotalAssets` (src/shmonad/StakeTracker.sol:2457). It does not reduce `state.redemptionsPayable` or clamp `state.queuedUnstake`.

This breaks isolated solvency accounting after a user has burned vShares into a fixed redemption. For example, a user can request a 100 MON isolated exit, then an unexpected 10 MON isolated deficit reduces backing active stake to 90 while the request still remains payable for 100. Completion later requires `state.idleAssets >= amountMon` (src/shmonad/StakeTracker.sol:3134), so the request can remain stuck at 90 idle assets.

A later isolated deposit into the same validator increases backing assets while stale `redemptionsPayable` remains outstanding. Since isolated equity is computed as `totalAssets - redemptionsPayable`, the new depositor can effectively fund the old redemption deficit and be diluted or wiped out.

The code already has a shortfall mechanism for isolated withdrawal settlement:

`_finalizeCompletedIsolatedWithdrawal()` calls `_applyIsolatedRedemptionShortfall()` when received principal is lower than expected (src/shmonad/StakeTracker.sol:1689), which haircuts `redemptionsPayable` and updates the redemption rate (src/shmonad/StakeTracker.sol:401). The balance-verification deficit path lacks equivalent handling.

Scenario:

- Bob locks a 100 ether isolated exit.
- A synthetic 5 ether isolated active-stake deficit is applied via balance verification.
- `redemptionsPayable` stays at 100 ether, while only 95 ether can be withdrawn and `queuedUnstake` remains stale at 5 ether.
- Bob's completion reverts with `InsufficientIsolatedIdleAssets`.

M-01 Isolated Active-Stake Deficits Do Not Haircut Locked Redemptions

CONTINUED

- Alice later deposits 5 ether into the same validator.
- That deposit is withdrawn to backfill Bob's stale exit, Bob receives the full 100 ether, and Alice's isolated shares are left redeemable for 0.

— RECOMMENDATION

When an isolated deficit is applied during balance verification, apply the same seniority model used during isolated withdrawal settlement: remaining vShare equity absorbs losses first, and any deficit exceeding share-backed equity haircuts redemptionsPayable through `_applyIsolatedRedemptionShortfall()`.

After changing redemptionsPayable, recompute queuedUnstake to the remaining uncovered redemption need:

— RESOLUTION

Status: Pending.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/748>

MEDIUM **M-02**

PENDING

Atomic Reserve Leaves Queue Live

CATEGORY	Logical Error
LOCATION	https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/7910434ed048d9da1019b04e5684443e9c08ab4b/src/shmonad/StakeTracker.sol#L2713
REVIEW	Main Review 1

— DESCRIPTION

`_settleGlobalNetMONAgainstAtomicUnstaking()` can reserve freed atomic liquidity for outstanding redemptions without cancelling the matching validator unstake queue.

When the atomic target shrinks, freed liquidity is represented as positive `_netStakeIn`. The function first uses this amount to cover uncovered liabilities by increasing `s_globalCapital.reservedAmount`. This makes the MON available for redemptionsPayable, but the corresponding `queueForUnstake` remains unchanged.

This differs from `_offsetLiabilitiesWithDeposits()`, where liquid MON used to cover withdrawals also reduces both `queueToStake` and `queueForUnstake` via `QueueLib.reducePreferNonTurnover()`.

Example: if the pool has 100 MON of redemptionsPayable and 100 MON of `queueForUnstake`, and an atomic target reduction frees 100 MON, the function moves 100 MON into reserves. The redemption is now liquid-backed, but the unstake queue can remain live. Depending on later crank netting/capacity, validators may still unstake another 100 MON for an already-backed redemption, causing avoidable unstake/restake churn and temporary loss of productive validator stake.

The atomic path also calculates uncovered liabilities as `currentLiabilities - reservedAmount`, unlike `_offsetLiabilitiesWithDeposits()`, which also accounts for pending unstaking.

— RECOMMENDATION

When freed atomic liquidity is moved into reserves for uncovered redemption liabilities, reduce the matching `queueForUnstake` by the newly backed redemption amount using

`QueueLib.reducePreferNonTurnover()` and update the turnover queue. Only the remaining `_netStakeIn` should then be netted or routed to staking. Apply this only to redemption coverage, not reserves used for `rewardsPayable`.

— RESOLUTION

Status: Pending.

M-02 Atomic Reserve Leaves Queue Live

CONTINUED

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/745>

MEDIUM **M-03**

PENDING

Stake allocation is still dominated by yield

CATEGORY Math
LOCATION StakeAllocationLib.sol
REVIEW Main Review 1

DESCRIPTION

The protocol's stated intent, per the whitepaper, is to assign stake proportional to smoothed earned revenue, treating staking yield, MEV fee share, and boost donations identically.

The allocation formula in StakeAllocationLib.sol:77 implements this as:

```
``plain text
weightedRevenue = avg(stakingYield) + multiplierRay × avg(nonStakingRevenue)
`
```

However, staking yield is mechanically proportional to existing stake ($\text{stakingYield} \propto \text{stake}$), so a validator's revenue signal is dominated by a term that reflects their current allocation rather than their marginal contribution to protocol yield. In equilibrium, each validator's revenue share approximates their stake share, allocation deltas approach zero, and non-staking revenue becomes a small additive term that rarely changes rankings – directly contradicting the whitepaper's intent of differentiating validators by operational quality.

Using the whitepaper's own example: a validator with 20B staked earns ~10M in staking yield and 500K in MEV (MEV is <5% of total revenue). A validator with 50M staked earns ~2.5M in staking yield and 2M in MEV – generating 1600× more MEV per unit of stake. The larger validator still receives 2.3× the allocation weight despite producing 4× less MEV per unit of stake.

The v1.2 `nonStakingRevenueMultiplierRay` partially addresses this by amplifying the MEV signal.

However, since it operates on absolute revenue values and is a single global constant, the correct multiplier to equalize any given pair of validators depends on their relative sizes and MEV rates — no fixed value works correctly across the entire validator set simultaneously. The dominance of staking yield is structural, not a tuning problem.

RECOMMENDATION

Consider replacing the purely absolute non-staking revenue signal with a size-aware scoring function that smoothly transitions between absolute revenue and per-unit revenue.

The goal is to avoid two extremes:

M-03 Stake allocation is still dominated by yield

CONTINUED

- Pure absolute revenue favors large incumbent validators because larger validators naturally capture more total revenue.

- Pure per-unit revenue can over-amplify small validators and make the allocation sensitive to donation spikes.

A possible formulation is:

S_i = validator i pooled stake

R_i = validator i smoothed non-staking revenue

K = transition stake parameter

$\text{nonStakingScore}_i = R_i / (K + S_i)$

When S_i is much smaller than K , the denominator is approximately constant, so the score behaves closer to absolute revenue:

$\text{nonStakingScore}_i \approx R_i / K$

When S_i is much larger than K , the denominator is dominated by validator stake, so the score behaves closer to per-unit revenue:

$\text{nonStakingScore}_i \approx R_i / S_i$

This creates a spectrum: smaller validators are not excessively penalized for having low stake, while larger validators are increasingly evaluated by revenue efficiency per unit of stake.

To limit reward/donation manipulation, the score should also be bounded:

$\text{nonStakingScore}_i = \min(R_i / (K + S_i), \text{maxRate})$

This keeps the allocation signal sensitive to validator performance without allowing an arbitrarily large non-staking payment to buy unlimited stake allocation.

Example:

```
```javascript
```

Let:

$S_i$  = validator pooled stake

$R_i$  = smoothed non-staking revenue

$K$  = transition stake size

Example with  $K = 1B$ :

Small validator:

$S = 50M$

$R = 2M$

$\text{score} = 2M / 1.05B \approx 0.190\%$

Large validator:

$S = 20B$

$R = 500K$

$\text{score} = 500K / 21B \approx 0.00238\%$

**M-03** Stake allocation is still dominated by yield

CONTINUED

So small validators are mostly judged by absolute MEV, while large validators are judged closer to MEV efficiency.

$$\begin{aligned} \text{absoluteTarget}_i &= \\ &\text{totalStake} * \text{stakingYield}_i / \text{sum}(\text{stakingYield}) \\ \\ \text{nonStakingTarget}_i &= \\ &\text{totalStake} * \text{adjustedNonStakingScore}_i / \text{sum}(\text{adjustedNonStakingScore}) \\ \\ \text{finalTarget}_i &= \\ &\alpha * \text{absoluteTarget}_i \\ &+ (1 - \alpha) * \text{nonStakingTarget}_i \end{aligned}$$

Where:

$$\begin{aligned} \text{adjustedNonStakingScore}_i &= \\ &\min(R_i / (K + S_i), \text{maxRate}) \end{aligned}$$

This gives you:

- small validators: non-staking signal is closer to absolute revenue,
- large validators: non-staking signal is closer to per-unit revenue,
- maxRate prevents huge fake MEV/donation spikes,
- alpha controls how much staking yield inertia remains.

So the compact proposal is:

$$\text{score}_i = \min(R_i / (K + S_i), \text{maxRate})$$

then allocate a bounded portion of stake by score<sub>i</sub>, not by raw R<sub>i</sub>.  
...

**— RESOLUTION**

Status: Pending.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/753>

MEDIUM **M-04**

PENDING

## Pending Delta Masks Stake Deficit

CATEGORY	Logical Error
LOCATION	<a href="https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/7910434ed048d9da1019b04e5684443e9c08ab4b/src/shmonad/StakeTracker.sol#L2266">https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/7910434ed048d9da1019b04e5684443e9c08ab4b/src/shmonad/StakeTracker.sol#L2266</a>
REVIEW	Main Review 1

### — DESCRIPTION

`_handleStakeBalanceVerification()` can erase a real stake deficit when expected pending deposits differ from actual pending deposits. This can prevent ShMonad from updating validator accounting and can bypass the freeze/close check for unexpected stake loss.

The function computes expected total stake as pooled target stake plus isolated active and pending stake, then reads `actualActiveStake` and `actualPendingDeposits` from the staking precompile.

To avoid treating normal pending-deposit timing as a loss, the function computes `expectedActiveStake = expectedTotalStake - actualPendingDeposits`. This already normalizes the active-stake comparison using the precompile's actual pending amount.

However, after computing the active delta, the function subtracts the pending mismatch a second time. In the deficit branch, if `actualPendingDeposits > expectedPendingDeposits`, it subtracts `actualPendingDeposits - expectedPendingDeposits` from the deficit.

This double-discounts pending timing. A real stake deficit that remains after the first normalization can be reduced to zero before accounting reconciliation and the freeze/close threshold are applied.

Example: ShMonad expects a validator to have 150 MON total stake and no pending deposits. The precompile reports 50 MON pending and 80 MON active. The actual pending amount is already handled by the normalized comparison: `expectedActiveStake = 150 - 50 = 100`. Since actual active stake is only 80 MON, there is a real 20 MON deficit.

The function initially detects this correctly: `expectedActiveStake = 100`, `actualActiveStake = 80`, so `delta = 100 - 80 = 20` MON. But then it subtracts the pending mismatch again: `actualPendingDeposits - expectedPendingDeposits = 50 - 0 = 50`. The code applies `delta = delta > pendingDelta ? delta - pendingDelta : 0`, so because `20 < 50`, delta is set to 0. This masks the real 20 MON deficit.

As a result, the 20 MON deficit is not applied by the reconciliation branch. The validator's target stake and global staked accounting remain overstated, and the freeze/close threshold is checked against zero instead of the real deficit. The function still returns `unexpectedDeficit` from the earlier total-stake check, so the caller can reduce unstakable capacity, but the accounting and circuit-breaker updates are skipped.

The same double-discount exists in the surplus direction.

**M-04** Pending Delta Masks Stake Deficit

CONTINUED

**— RECOMMENDATION**

Do not subtract the pending-deposit mismatch after expectedActiveStake has already been calculated using actualPendingDeposits. Either keep the current normalized comparison and apply the resulting active delta directly, or compute expected active stake from expected pending deposits and perform exactly one timing adjustment. The freeze/close threshold should be evaluated against the real deficit remaining after a single pending-timing normalization.

**— RESOLUTION**

Status: Pending.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/749>

MEDIUM **M-05**

PENDING

# Revenue Double-Books Atomic Liquidity

CATEGORY Logical Error

LOCATION <https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/7910434ed048d9da1019b04e5684443e9c08ab4b/src/shmonad/StakeTracker.sol#L878>

REVIEW Main Review 1

## — DESCRIPTION

**M-05** Revenue Double-Books Atomic Liquidity

CONTINUED

Current-epoch revenue can be routed into redemption reserves and then reused in the same global crank to release atomic liquidity.

During global crank, `_offsetLiabilitiesWithDeposits()` runs before `_carryOverAtomicUnstakeIntoQueue()`. If current-epoch revenue has been queued to stake and there is an outstanding redemption liability, `_offsetLiabilitiesWithDeposits()` can move that liquid MON into `reservedAmount` and reduce the existing `queueForUnstake`.

However, this reserve fill does not mark the revenue as consumed in `globalRevenue.allocatedRevenue`. Later in the same crank, `_carryOverAtomicUnstakeIntoQueue()` computes available revenue as `earnedRevenue - allocatedRevenue`. Since the reserved revenue was not added to `allocatedRevenue`, the function treats the same MON as still available and reduces `atomic.distributedAmount`.

Example: Alice has an outstanding 10 MON delayed redemption with `redemptionsPayable = 10` and `queueForUnstake = 10`. The protocol also has enough other pooled equity, for example 500 MON of pooled staked capital at the default 2% atomic target, so `allocatedAmount = 10` remains the correct target during the crank. The contract receives 10 MON of current-epoch revenue, recorded as `earnedRevenue = 10` and `queueToStake = 10`. Atomic accounting is fully utilized with `allocatedAmount = 10` and `distributedAmount = 10`.

When the global crank runs, `_offsetLiabilitiesWithDeposits()` reserves the 10 MON for Alice and clears the original unstake queue. `reservedAmount` is now 10, but `allocatedRevenue` is still zero. The crank then reaches `_carryOverAtomicUnstakeIntoQueue()`, sees `earnedRevenue - allocatedRevenue = 10`, and reduces `distributedAmount` by 10.

Atomic liquidity is now shown as available even though the only newly received 10 MON native balance is already reserved for Alice. A later atomic withdraw can consume this apparent liquidity, net of fees. If there is no extra idle native MON, Alice's `completeUnstake()` can pass the accounting reserve check because `reservedAmount` says her redemption is backed, but the final native MON transfer can fail until a later validator withdrawal replenishes the contract.

The comments above `_carryOverAtomicUnstakeIntoQueue()` explicitly say already-earmarked revenue must not be reused. Current tracking only treats `_accountForWithdraw()` usage as earmarked, but ignores revenue earmarked into redemption reserves by `_offsetLiabilitiesWithDeposits()`.

**— RECOMMENDATION**

When `_offsetLiabilitiesWithDeposits()` uses current-epoch revenue or revenue-backed `queueToStake` to fill redemption reserves, mark that portion unavailable to `_carryOverAtomicUnstakeIntoQueue()` in the same crank. This can be done by increasing `allocatedRevenue` for only the reserve-filled amount that is actually backed by `earnedRevenue`, tracking a crank-local consumed-revenue amount, or splitting `queueToStake` into revenue-backed and deposit-backed buckets so the consumed source is explicit.

**M-05** Revenue Double-Books Atomic Liquidity

CONTINUED

**— RESOLUTION**

Status: Pending.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/750>

MEDIUM **M-06**

PENDING

## Pending Sync Misattributes Stake

CATEGORY	Logical Error
LOCATION	<a href="https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/7910434ed048d9da1019b04e5684443e9c08ab4b/src/shmonad/StakeTracker.sol#L507">https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/7910434ed048d9da1019b04e5684443e9c08ab4b/src/shmonad/StakeTracker.sol#L507</a>
REVIEW	Main Review 1

### — DESCRIPTION

`_syncIsolatedPendingStake()` can promote isolated pendingStake into isolated activeStake even when the isolated native delegation is still pending.

Monad's staking precompile exposes ShMonad's delegation as aggregate active stake plus aggregate scheduled stake. It does not identify whether pending stake came from pooled shMON deposits or isolated validator-local deposits.

ShMonad resolves this ambiguity by subtracting locally expected pooled pending deposits from aggregate native pending deposits. This is unsafe when pooled and isolated deposits overlap across a boundary-delayed staking window.

If a pooled delegation has already activated natively while ShMonad still tracks it as local pooled pending, and a newer isolated delegation to the same validator is still native-pending, `_syncIsolatedPendingStake()` attributes the aggregate native pending amount to pooled first. The isolated pending amount is then treated as activated and promoted into isolated activeStake.

This breaks the pooled/isolated accounting boundary: an isolated exit can be funded by pooled active stake while the isolated deposit remains native-pending. The protocol may still have the same aggregate validator balance, but the wrong accounting domain owns the active and withdrawing stake. Any yield, withdrawal surplus/deficit, or slashing exposure during this mismatch window can therefore be assigned to the wrong domain. Balance verification may therefore emit only a pending mismatch or fail to repair the domain-level misattribution.

Example: ShMonad locally tracks a 20 MON pooled deposit as boundary-delayed pending, but Monad has already activated it natively. A newer 10 MON isolated deposit is still pending natively, so Monad reports 20 MON active and 10 MON aggregate pending.

`_syncIsolatedPendingStake()` computes isolated pending as zero because aggregate pending is less than locally expected pooled pending. It promotes the 10 MON isolated pendingStake into isolated activeStake. If the isolated user has a queued exit, `_queueAndInitiateIsolatedUnstake()` can initiate a 10 MON isolated withdrawal.

After the crank, local isolated pendingStake is zero and local isolated pendingUnstake is 10 MON, but native pending stake is still 10 MON and native active stake has dropped from 20 MON to 10 MON.

**M-06** Pending Sync Misattributes Stake

CONTINUED

**— RECOMMENDATION**

Do not promote isolated pendingStake into activeStake based only on aggregate pending subtraction. Keep isolated stake pending until isolated activation can be proven, or prevent isolated withdrawals while active stake may be ambiguously sourced by boundary-delayed pooled pending.

**— RESOLUTION**

Status: Pending.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/751>

MEDIUM **M-07**

PENDING

# Post-Roll Coinbase Rewrites Split

CATEGORY	Logical Error
LOCATION	<a href="https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/7910434ed048d9da1019b04e5684443e9c08ab4b/src/shmonad/StakeTracker.sol#L1350">https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/7910434ed048d9da1019b04e5684443e9c08ab4b/src/shmonad/StakeTracker.sol#L1350</a>
REVIEW	Main Review 1

## — DESCRIPTION

`_crankValidator()` processes an existing contract coinbase balance only after `_rollValidatorEpochForwards()` rewrites `snapshotPooledStake` and `snapshotIsolatedStake`. To the extent that balance produces a ShMonad donation, validator-attributed value that already existed before the crank can be classified using post-crank stake state.

The crank first settles staking yield, past epoch edges, completed isolated withdrawals, reward liabilities, balance verification, isolated restakes and unstakes, and pooled stake delta. It then calls `_rollValidatorEpochForwards()`, which refreshes the isolated split snapshots. Only after that, if the coinbase is a contract with balance, `_settleCoinbaseContract()` calls `Coinbase.process()` and routes the ShMonad balance delta through `_handleValidatorRewards()`, which splits value using `_splitIsolatedAndPooledBySnapshot()`.

Example: Alice is the only isolated holder on validator V with 10 MON active isolated stake. Before V is cranked, its coinbase contract already holds priority-fee balance that will produce a 3 MON ShMonad donation when `process()` runs. Alice then requests an isolated unstake for all 10 MON before V is cranked. During the crank, `_queueAndInitiateIsolatedUnstake()` removes isolated active stake and `_rollValidatorEpochForwards()` snapshots isolated stake as zero. The same crank then processes the pre-existing coinbase balance. `Coinbase.process()` sends the 3 MON donation to ShMonad, and `_settleCoinbaseContract()` routes that balance delta through `_handleValidatorRewards()`. Because the split now sees zero isolated stake, the full 3 MON donation is routed to pooled non-staking allocation revenue instead of isolated redemption backing, even though the donation-producing priority-fee balance was already sitting in the coinbase before Alice's exit was applied.

This conflicts with the snapshot-splitting design documented in code and docs. The split basis is supposed to prevent the same crank from mutating stake first and then using fresh state to decide who receives validator-attributed value. It also conflicts with `_creditIsolatedRewards()`, which keeps isolated-attributed value liquid for exit requests when all isolated shares have already been burned.

The affected value is the ShMonad donation slice of the coinbase balance, not the entire priority-fee balance. The rest of the balance is routed by `Coinbase.process()` to commission or staking-precompile reward paths.

**M-07** Post-Roll Coinbase Rewrites Split

CONTINUED

**— RECOMMENDATION**

Process `_settleCoinbaseContract()` before `_rollValidatorEpochForwards()`, or cache the pre-roll pooled and isolated split basis at the start of `_crankValidator()` and use that cached basis when processing coinbase balances that existed before the roll.

**— RESOLUTION**

Status: Pending.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/754>

MEDIUM **M-08**

PENDING

# Isolated Surplus Bypasses Yield Fees

CATEGORY	Logical Error
LOCATION	<a href="https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/7910434ed048d9da1019b04e5684443e9c08ab4b/src/shmonad/StakeTracker.sol#L1654">https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/7910434ed048d9da1019b04e5684443e9c08ab4b/src/shmonad/StakeTracker.sol#L1654</a>
REVIEW	Main Review 1

## — DESCRIPTION

`_finalizeCompletedIsolatedWithdrawal()` credits isolated withdrawal surplus directly into isolated accounting instead of treating it as staking yield.

When an isolated precompile withdrawal returns more than `expectedAmount`, the surplus is added to `s_isolatedTotalAssets` and `s_isolatedLiquidAssets`. If the validator is active and still has isolated `vShares`, the surplus is added to `queuedRestake`; otherwise it is added to `idleAssets`.

This keeps the value inside the isolated domain, but bypasses the normal staking-yield path in `_handleEarnedStakingYield()`. As a result, the surplus does not pay `s_admin.stakingCommission`, does not mint zero-yield commission to `OWNER_COMMISSION_ACCOUNT`, and does not update `validatorAllocationRevenue.stakingYield` / `globalAllocationRevenue.stakingYield`.

This differs from pooled withdrawal settlement. In `_handleCompleteDecreasedAllocation()`, the code explicitly states that Monad undelegated balances can continue to accrue rewards during the withdrawal period and treats withdrawal surplus as additional staking yield by calling `_handleEarnedStakingYield()`. The same Monad-side premise applies to isolated withdrawals. Monad's staking API pays withdrawal principal plus withdrawal rewards on `withdraw()`, so an isolated withdrawal returning more than its expected principal is staking yield, not raw principal.

Example: staking commission is 10%, isolated rewards are included in the allocation signal, and an isolated withdrawal for 100 MON settles for 102 MON. The 2 MON surplus is staking yield earned during the withdrawal delay. The pooled path would charge 0.2 MON commission and record staking-yield allocation signal. The isolated path instead credits the full 2 MON to isolated accounting and records zero owner commission and zero allocation signal.

## — RECOMMENDATION

**M-08** Isolated Surplus Bypasses Yield Fees

CONTINUED

Preserve the policy that isolated withdrawal surplus remains isolated, but apply staking-yield treatment to the surplus. Charge `s_admin.stakingCommission`, credit only the net surplus into isolated accounting, and update validator/global staking-yield allocation signal according to `s_includelockedRewardsInStakeAllocation`.

Avoid calling `_handleEarnedStakingYield()` directly unless it is adapted, because its generic snapshot split may route part of an isolated-only surplus to pooled accounting.

**— RESOLUTION**

Status: Pending.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/755>

MEDIUM **M-09**

PENDING

# Zero Equity Bricks Global Crank

CATEGORY DoS

LOCATION <https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/7910434ed048d9da1019b04e5684443e9c08ab4b/src/shmonad/StakeTracker.sol#L1654>

REVIEW Main Review 1

## DESCRIPTION

`_checkSetNewAtomicLiquidityTarget()` can divide by zero during global crank when pooled equity is zero but an old atomic liquidity allocation remains.

When no pending atomic target update exists, i.e. `s_pendingTargetAtomicLiquidityPercent == FLOAT_PLACEHOLDER`, `_checkSetNewAtomicLiquidityTarget()` compares the current atomic allocation percentage to the target percentage by calling `_scaledPercentFromAmounts(oldAllocatedAmount, _totalEquity)`. That helper divides by the denominator and documents that callers must ensure the denominator is nonzero. `_checkSetNewAtomicLiquidityTarget()` does not enforce that precondition. This is reachable through normal pooled exit flow. A user can deposit, the protocol can crank until the default atomic allocation exists, and then all outstanding pooled shares can be burned into traditional redemptions. At that point pooled `totalAssets()` is zero, while the old atomic allocation can remain nonzero because `_afterRequestUnstake()` only clears utilized atomic liquidity and can leave `allocatedAmount` unchanged.

On the next global crank, `_totalEquity` is zero and `oldAllocatedAmount` is nonzero, so the percentage calculation reverts with division by zero. The revert happens before global epoch advancement and before validator withdrawals for outstanding redemptions can progress.

This can brick public `crank()` in a full-exit state until positive pooled equity is introduced or governance/admin repairs the atomic target state. The state is recoverable, but recovery depends on external liquidity or intervention rather than the protocol self-settling the existing redemptions.

Example: Alice deposits pooled MON and the protocol initializes a nonzero atomic allocation. Alice and any other remaining pooled holders then request traditional unstake for all outstanding pooled shares. Pooled equity becomes zero, but atomic `allocatedAmount` remains nonzero. The next `crank()` enters global crank, calls `_checkSetNewAtomicLiquidityTarget()`, and reverts while computing `oldAllocatedAmount / _totalEquity`.

## RECOMMENDATION

**M-09** Zero Equity Bricks Global Crank

CONTINUED

Handle zero pooled equity explicitly in atomic target rebalancing. If `_totalEquity` is zero, avoid percentage division, set or defer the atomic target safely, and continue global crank so outstanding redemption withdrawals can settle.

**— RESOLUTION**

Status: Pending.

MEDIUM **M-10**

PENDING

# Deficit Can Lose Unstake Demand

CATEGORY	Logical Error
LOCATION	<a href="https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/7910434ed048d9da1019b04e5684443e9c08ab4b/src/shmonad/StakeTracker.sol#L1350">https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/7910434ed048d9da1019b04e5684443e9c08ab4b/src/shmonad/StakeTracker.sol#L1350</a>
REVIEW	Main Review 1

## — DESCRIPTION

A validator-side balance-verification deficit can reduce a validator's pooled unstakable capacity after the global redemption queue for the round has already been fixed. When this happens, ShMonad can initiate less validator unstake than the locked redemption liability requires, without preserving the missing remainder in either the current or next-epoch `queueForUnstake`.

`_handleStakeBalanceVerification()` is ShMonad's reconciliation path for unexpected validator stake mismatches: it reduces target/staked accounting for active-stake deficits and can freeze/close the protocol for slash-sized losses. The issue is that this reconciliation happens after the global unstake queue for the validator round has already been fixed, and the resulting lost unstake demand is not preserved.

During the validator round, `_computeStakeDelta()` allocates each validator's share of the previously fixed unstake queue using frozen round inputs. But earlier in the same validator crank, `_handleStakeBalanceVerification()` can discover an unexpected active-stake deficit and reduce the validator's target accounting. `_crankValidator()` also subtracts the detected deficit from the validator's unstakable capacity before calling `_computeStakeDelta()`.

This means the validator can initiate less unstake than its original share of the global redemption queue. The missing amount is not requeued anywhere.

Example: Alice requests a 200 MON pooled redemption. This creates `redemptionsPayable = 200` and a 200 MON unstake queue. Two validators each have 100 MON target stake. During validator A's crank, balance verification detects a 10 MON deficit, so A initiates only 90 MON of unstake. Validator B initiates 100 MON. After both validators crank, only 190 MON of validator withdrawals are pending, while `redemptionsPayable` remains 200 and both current and next `queueForUnstake` are zero.

The missing 10 MON redemption demand is now no longer represented by pending validator withdrawals or live queue state.

The shortfall can roll forward to later users or unrelated liquidity. If a later redemption creates fresh unstake demand, the resulting validator withdrawal can be consumed by the older under-backed redemption first, leaving the later user without their own reserve, pending validator withdrawal, or live queue demand.

**M-10** Deficit Can Lose Unstake Demand

CONTINUED

**— RECOMMENDATION**

When `_handleStakeBalanceVerification()` reduces pooled target or unstakable capacity during a validator round, preserve any lost unstake demand explicitly. One approach is to requeue the difference between the validator's pre-deficit requested decrease and its post-deficit initiated decrease.

**— RESOLUTION**

Status: Pending.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/748>

MEDIUM **M-11**

PENDING

# Shortfall can leave excess isolated liq idle

CATEGORY	Logical Error
LOCATION	<a href="https://github.com/FastLane-Labs/fastlane-contracts-internal/blob/shmonad-v1.2/src/shmonad/StakeTracker.sol#L3141">https://github.com/FastLane-Labs/fastlane-contracts-internal/blob/shmonad-v1.2/src/shmonad/StakeTracker.sol#L3141</a>
REVIEW	Main Review 1

## DESCRIPTION

`_reserveQueuedIsolatedLiquidityForRedemptions()` moves `queuedRestake` into `idleAssets` when current redemption coverage is below `redemptionsPayable`.

```plain text

```
uint256 covered = uint256(state.idleAssets) + state.pendingUnstake + state.queuedUnstake;
```

```
uint256 redemptionsPayable = state.redemptionsPayable;
```

```
if (covered >= redemptionsPayable || state.queuedRestake == 0) return;
```

```
uint120 amountToMove =
```

```
Math.min(redemptionsPayable - covered, uint256(state.queuedRestake)).toUint120();
```

```
state.queuedRestake -= amountToMove;
```

```
state.idleAssets += amountToMove;
```

```
s_isolatedIdleAssets += amountToMove;
```

```
``
```

If a later isolated withdrawal settles short, `_applyIsolatedRedemptionShortfall()` reduces `redemptionsPayable`. However, any liquidity previously moved from `queuedRestake` into `idleAssets` is not rebalanced back into `queuedRestake`, even when it is no longer needed to cover exits.

Example:

```
redemptionsPayable = 100
```

```
queuedUnstake = 80
```

```
queuedRestake = 50
```

```
idleAssets = 0
```

The reservation helper moves 20 from `queuedRestake` into `idleAssets`.

Later, a short settlement reduces:

```
redemptionsPayable = 70
```

The isolated bucket now has excess idle liquidity. That MON still belongs to isolated claimants through isolated equity, but it remains liquid and does not earn staking yield. There is no active-validator path that proactively moves the excess back into `queuedRestake`.

M-11 Shortfall can leave excess isolated liq idle

CONTINUED

— RECOMMENDATION

After applying isolated redemption shortfalls, or before isolated restake processing, rebalance excess idleAssets back into queuedRestake when:

validator is active

isolated vShares remain

idle/pending/queued coverage exceeds redemptionsPayable

This ensures liquidity reserved for exits can resume earning yield once it is no longer needed for redemptions.

— RESOLUTION

Status: Pending.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/746>

MEDIUM **M-12**

PENDING

Isolated shortfall allocation depends on timing

CATEGORY Rewards

LOCATION <https://github.com/FastLane-Labs/fastlane-contracts-internal/blob/shmonad-v1.2/src/shmonad/StakeTracker.sol#L349>

REVIEW Main Review 1

— DESCRIPTION

In v1.2, isolated exits are priced and locked when vShares are burned. If `idleAssets`, `pendingUnstake`, `queuedUnstake`, or reserved `queuedRestake` are available when a user requests isolated unstake, that user's redemption can become partially or fully covered before a later validator withdrawal shortfall is realized.

This means isolated shortfalls are not necessarily shared pro-rata across the full set of users who had isolated exposure before the shortfall event. Users who request exit while coverage exists may be less exposed than users who remain in vShares.

Example:

1. Alice and Bob each hold 100 MON of isolated exposure to validator V.
2. V has 20 MON in `queuedRestake`.
3. Bob requests isolated unstake. The system can reserve the 20 MON toward Bob's locked redemption.
4. Later, validator V's actual withdrawal completes with a 20 MON shortfall.
5. Bob's already-covered redemption is less exposed to the shortfall, while Alice absorbs more of the loss through reduced residual isolated equity.

Impact is mainly economic clarity/fairness rather than an immediate solvency break. The docs do describe locked isolated exits as senior claims, so this may be intended. However, they do not clearly call out that loss exposure can depend on request timing and available isolated coverage.

— RECOMMENDATION

Explicitly document whether isolated shortfalls are intended to be request-time/coverage-based, or whether they should be shared pro-rata across an epoch/cohort of isolated stakers.

— RESOLUTION

Status: Pending.

M-12 Isolated shortfall allocation depends on timing

CONTINUED

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/746>

MEDIUM **M-13**

PENDING

Atomic reserve fills leave duplicate queue

CATEGORY Logical Error
 LOCATION StakeTracker.sol#L2713
 REVIEW Main Review 1

DESCRIPTION

When `_settleGlobalNetMONAgainstAtomicUnstaking` frees atomic liquidity and routes it to `reservedAmount`, it does not reduce the corresponding `queueForUnstake`.

This creates duplicate fulfillment pressure for the same redemption demand. The redemption is partially covered by reserves, but the validator unstake queue still asks validators to withdraw the original amount. Later validator cranks can therefore initiate unnecessary withdrawals. When those withdrawals complete, the returned MON is treated as surplus/general liquidity instead of being needed for the already-covered redemption.

This is most visible in the `_stakeIn > _unstakeOut` branch:

```
```plain text
```

```
if (_currentLiabilities > _reservedAssets) {
 uint256 _uncoveredLiabilities = _currentLiabilities - _reservedAssets;
 uint120 _toReserves = _uncoveredLiabilities > _netStakeIn ? _netStakeIn : uint120(_uncoveredLiabilities);

 s_globalCapital.reservedAmount += _toReserves;
 _netStakeIn -= _toReserves;
}
```

The reserve fill covers liabilities, but no matching amount is removed from `globalCashFlows.queueForUnstake`.

## Example

Initial state:

```
`vb.net
redemptionsPayable = 80 MON
reservedAmount = 0
queueForUnstake = 80 MON
```

Atomic settlement shrinks the atomic pool and frees 60 MON.

Current result:

```
```vb.net
```

M-13 Atomic reserve fills leave duplicate queue

CONTINUED

```
reservedAmount = 60 MON
queueForUnstake = 80 MON
```

```
``[
The user redemption is now covered for 60 MON , but the system still carries the full 80
MON unstake queue. A later validator crank may initiate 80 MON of validator withdrawals
even though only 20 MON is still needed to cover the redemption.
```

```
Expected result:
```

```
` vb.net
reservedAmount = 60 MON
queueForUnstake = 20 MON
``
```

This can cause avoidable validator unstake/restake churn, delayed allocation convergence, and reduced staking efficiency. It can also distort turnover behavior because the remaining queue overstates actual redemption-driven unstake demand.

— RECOMMENDATION

When freed atomic liquidity is routed to reserves, reduce `queueForUnstake` by the portion that actually covers redemption liabilities.

The reduction should apply only to redemption reserve deficit, not validator rewards.

PROOF OF CONCEPT <https://gist.github.com/GuardianAudits/998241d9fb5b6f8d035de74415063d7d>

— RESOLUTION

Status: Pending.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/745>

MEDIUM **M-14**

PENDING

Queue netting consumes turnover first

CATEGORY Logical Error
 LOCATION StakeTracker.sol#L2807
 REVIEW Main Review 1

DESCRIPTION

Several queue-netting paths reduce the turnover portion of `queueForUnstake` before reducing the non-turnover portion.

In `_settleGlobalNetMONAgainstAtomicUnstaking`, when atomic settlement frees liquidity and has remaining `_netStakeIn` after reserve coverage, it nets that liquidity against `queueForUnstake` using `QueueLib.reducePreferTurnover`.

In `_offsetExcessQueueCapacityWithNet`, the same policy is applied indirectly through `QueueLib.planExcessQueueOffset`:

In both cases, netting consumes allocation-driven rebalancing before consuming user/liability-style pro-rata withdrawals.

This is inconsistent with the economic meaning of these netting paths:

- atomic settlement can free liquid MON that substitutes for validator withdrawals that would otherwise cover redemption/liability demand;
- excess-capacity netting cancels opposite stake and unstake queues to make the remaining queues fit current stakable/unstakable capacity.

In both cases, reducing the non-turnover portion first would preserve targeted turnover that moves stake away from overallocated validators while achieving the same aggregate queue reduction.

This is different from `_clampQueuesToCapacityOrRoll`, where `reducePreferTurnover` is more defensible. That function handles true unstakable-capacity scarcity. If the protocol cannot execute the whole unstake queue, prioritizing user/liability-style withdrawals over discretionary rebalancing is reasonable.

Example

Assume the current queue is:

```
``plain text
queueForUnstake = 100 MON
nonTurnover = 60 MON
turnover = 40 MON
`|
```

Atomic settlement frees 30 MON after reserve coverage, or excess-capacity netting determines that 30 MON of opposite stake/unstake queues should cancel.

M-14 Queue netting consumes turnover first

CONTINUED

Current behavior with `reducePreferTurnover(30)` :

```
``plain text
queueForUnstake = 70 MON
nonTurnover = 60 MON
turnover = 10 MON
`
```

The same total amount remains queued, but most of the remaining withdrawal is now non-turnover. During validator crank, that portion is distributed stake-proportionally, not according to over-allocation.

If the freed liquidity instead consumed non-turnover first:

```
`plain text
queueForUnstake = 70 MON
nonTurnover = 30 MON
turnover = 40 MON
`
```

Users remain covered by the freed liquidity, while the remaining validator withdrawal pressure continues to target overallocated validators.

For excess-capacity netting, the aggregate result is identical:

```
`plain text
queueForUnstake is reduced by 30 MON
queueToStake is reduced by 30 MON
``
```

Only the remaining queue classification changes. Preserving turnover leaves more of the executable validator withdrawal pressure on the overallocated validators instead of the stake-proportional path.

****Impact****

The total amount withdrawn may be unchanged, and user redemptions can still be paid. However, the validator-level withdrawal distribution changes:

- turnover withdrawals are routed by over-allocation;
- non-turnover withdrawals are routed stake-proportionally.

By consuming turnover first, these netting paths can weaken the protocol's incentive-alignment turnover and leave more withdrawal pressure in the stake-proportional path. This can delay allocation convergence and create unnecessary pro-rata churn instead of targeted rebalancing.

— RECOMMENDATION

When freed liquidity or opposite stake-side queue capacity is netted against `queueForUnstake`, prefer reducing the non-turnover portion first:

```
```plain text
```

**M-14** Queue netting consumes turnover first

CONTINUED

```
(_globalCashFlows.queueForUnstake, _currentTurnoverQueueForUnstake,) =
QueueLib.reducePreferNonTurnover(
 _currentQueueForUnstake, _currentTurnoverQueueForUnstake, _netStakeIn
);
`
```

Similarly, `QueueLib.planExcessQueueOffset` should use `reducePreferNonTurnover` for the planned offset if the goal is to preserve incentive-alignment turnover when aggregate capacity constraints can still be met.

Keep `reducePreferTurnover` for true unstakable-capacity scarcity paths such as `_clampQueuesToCapacityOrRoll``, where preserving user/liability withdrawals over rebalancing is a reasonable priority.

If the protocol intentionally wants all capacity netting to cancel turnover before user/liability-style withdrawal demand, document that policy explicitly and clarify that these paths may reduce incentive-alignment turnover for the epoch.

**— RESOLUTION**

Status: Pending.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/756>

MEDIUM **M-15**

PENDING

# Deficit Can Lose Unstake Demand

CATEGORY	Logical Error
LOCATION	<a href="https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/7910434ed048d9da1019b04e5684443e9c08ab4b/src/shmonad/StakeTracker.sol#L1350">https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/7910434ed048d9da1019b04e5684443e9c08ab4b/src/shmonad/StakeTracker.sol#L1350</a>
REVIEW	Main Review 1

## — DESCRIPTION

A validator-side balance-verification deficit can reduce a validator's pooled unstakable capacity after the global redemption queue for the round has already been fixed. When this happens, ShMonad can initiate less validator unstake than the locked redemption liability requires, without preserving the missing remainder in either the current or next-epoch `queueForUnstake`.

`_handleStakeBalanceVerification()` is ShMonad's reconciliation path for unexpected validator stake mismatches: it reduces target/staked accounting for active-stake deficits and can freeze/close the protocol for slash-sized losses. The issue is that this reconciliation happens after the global unstake queue for the validator round has already been fixed, and the resulting lost unstake demand is not preserved.

During the validator round, `_computeStakeDelta()` allocates each validator's share of the previously fixed unstake queue using frozen round inputs. But earlier in the same validator crank,

`_handleStakeBalanceVerification()` can discover an unexpected active-stake deficit and reduce the validator's target accounting. `_crankValidator()` also subtracts the detected deficit from the validator's unstakable capacity before calling `_computeStakeDelta()`.

This means the validator can initiate less unstake than its original share of the global redemption queue. The missing amount is not requeued anywhere.

Example: Alice requests a 200 MON pooled redemption. This creates `redemptionsPayable = 200` and a 200 MON unstake queue. Two validators each have 100 MON target stake. During validator A's crank, balance verification detects a 10 MON deficit, so A initiates only 90 MON of unstake. Validator B initiates 100 MON. After both validators crank, only 190 MON of validator withdrawals are pending, while `redemptionsPayable` remains 200 and both current and next `queueForUnstake` are zero.

The missing 10 MON redemption demand is now no longer represented by pending validator withdrawals or live queue state.

The shortfall can roll forward to later users or unrelated liquidity. If a later redemption creates fresh unstake demand, the resulting validator withdrawal can be consumed by the older under-backed redemption first, leaving the later user without their own reserve, pending validator withdrawal, or live queue demand.

**M-15** Deficit Can Lose Unstake Demand

CONTINUED

**— RECOMMENDATION**

When `_handleStakeBalanceVerification()` reduces pooled target or unstakable capacity during a validator round, preserve any lost unstake demand explicitly. One approach is to requeue the difference between the validator's pre-deficit requested decrease and its post-deficit initiated decrease.

**— RESOLUTION**

Status: Pending.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/748>

MEDIUM **M-16**

PENDING

# Split Exits Bypass Delay

CATEGORY	Logical Error
LOCATION	<a href="https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/7910434ed048d9da1019b04e5684443e9c08ab4b/src/shmonad/ShMonad.sol#L310">https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/7910434ed048d9da1019b04e5684443e9c08ab4b/src/shmonad/ShMonad.sol#L310</a>
REVIEW	Main Review 1

## — DESCRIPTION

requestUnstake() applies the extra large-exit completion delay to each individual request slice before same-epoch requests are merged.

Large traditional exits are supposed to receive a two-epoch delay when the requested amount exceeds the current safe redemption envelope. This accounts for cases where pending stake must activate before it can be unstaked.

However, \_calcUnstakeRequestCompletionEpoch() checks only the current request amount. Afterward, TradUnstakeLib.enqueuePendingRequest() merges requests with the same completion epoch into one bucket.

A user can split one large exit into smaller same-epoch chunks. Each chunk is below the delay threshold, so each receives the base completion epoch. The chunks then merge into a single large request bucket at that earlier epoch.

This lets a large redemption compete for shared reserves and atomic fallback liquidity two epochs earlier than the same aggregate request submitted normally. Since completeUnstake() pays mature requests from shared backing without FIFO protection across accounts, the split large request can drain early backing before smaller requests that were already mature.

Example: Alice first requests a small unstake and receives completion epoch  $N + 5$ . Bob has an 810 MON exit that would receive  $N + 7$  as one request. By splitting it into nine 90 MON requests, Bob gets all chunks assigned to  $N + 5$ ; they merge into one 810 MON bucket. At  $N + 5$ , Bob can complete first and consume the available backing, causing Alice's earlier mature request to revert until later backing arrives.

## — RECOMMENDATION

Apply the delay decision to cumulative redemption demand, not individual calls. Track global new redemption demand for the current internal epoch and assign the base completion epoch only up to the safe envelope. Any amount above the envelope should receive the delayed epoch.

**M-16** Split Exits Bypass Delay

CONTINUED

**— RESOLUTION**

Status: Pending.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/757>

MEDIUM **M-17**

PENDING

# Excess netting underrepresents redemptions

CATEGORY Math

LOCATION StakeTracker.sol#L1160-L1210, QueueLib.sol#L88-L138

REVIEW Main Review 1

## DESCRIPTION

`_offsetExcessQueueCapacityWithNet` can reduce the aggregate `queueForUnstake` by more than it increases `reservedAmount`, even while redemption liabilities remain under-reserved.

`planExcessQueueOffset` computes:

```
``plain text
plan.targetOffsetAmount = targetOffsetAmount.toUint120();
plan.reducedQueueToStake = (queuedToStakeAmount - targetOffsetAmount).toUint120();
(plan.reducedQueueForUnstake, plan.reducedTurnoverQueueForUnstake,) =
reducePreferTurnover(queuedForUnstakeAmount, turnoverQueueForUnstake, targetOffsetAmount);
`
```

Separately, reserve fill is capped:

```
`plain text
if (reserveIncrease > globalStakableAmount) reserveIncrease = globalStakableAmount;
if (reserveIncrease > targetOffsetAmount) reserveIncrease = targetOffsetAmount;
plan.reserveIncrease = reserveIncrease.toUint120();
`
```

This means:

```
`vb.net
queueForUnstake reduction = targetOffsetAmount
reservedAmount increase = reserveIncrease
`
```

and `reserveIncrease` can be smaller than `targetOffsetAmount`.

If liabilities remain above reserves after the reserve increase, the function can remove redemption funding demand from the live unstake queue without creating equivalent reserved MON.

**Example**

Initial state:

```
```vb.net
redemptionsPayable = 100 MON
```

M-17 Excess netting underrepresents redemptions

CONTINUED

```
reservedAmount = 20 MON
```

```
pendingUnstaking = 0
```

```
queueForUnstake = 100 MON
```

```
turnoverQueue = 0 MON
```

```
queueToStake = 50 MON
```

```
pooled liquid MON available to reserve = 10 MON
```

```
`
```

planExcessQueueOffset can reduce both queues by 40 MON due to capacity netting, but only 10 MON can be moved into reserves.

After _offsetExcessQueueCapacityWithNet :

```
` vb.net
```

```
redemptionsPayable = 100 MON
```

```
reservedAmount = 30 MON
```

```
queueForUnstake = 60 MON
```

```
`
```

The redemption liability remains 100 MON , but represented funding is now only:

```
` vb.net
```

```
reservedAmount + live queueForUnstake = 30 + 60 = 90 MON
```

```
`
```

The missing 10 MON` remains part of the reserve target, but there is no automatic mechanism that recreates the cancelled unstake queue. The redemption may later depend on unrelated deposits, atomic liquidity, or future withdrawals rather than the original unstake demand created for that exit.

****Impact****

The protocol can underrepresent redemption funding demand during global queue netting. This can delay traditional unstake completion, force reliance on atomic liquidity, or shift funding pressure to later unrelated liquidity.

— RECOMMENDATION

When liabilities remain under-reserved, do not reduce redemption-backed unstake demand by more than the amount actually moved into reserves.

One approach:

1. Compute the reserve-backed offset:

```
` vb.net
```

```
reserveBackedOffset = min(targetOffsetAmount, reserveIncrease)
```

```
`
```

1. Apply reserveBackedOffset` against non-turnover unstake demand, because it directly

M-17 Excess netting underrepresents redemptions

CONTINUED

covers redemption liabilities.

2. Any remaining pure capacity/churn offset should only consume turnover. If turnover is insufficient, leave the remaining non-turnover queue live or roll it forward.

More generally, preserve the invariant:

```
`` vb.net
```

```
redemption liabilities <= reservedAmount + pendingUnstaking + live redemption-backed queue +  
available atomic cover
```

```
``
```

unless the protocol intentionally documents that redemptions may depend on unrelated future liquidity after queue netting.

— RESOLUTION

Status: Pending.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/758>

MEDIUM **M-18**

PENDING

Boosted high-rate state can make large deposits fail the share-rate breaker

CATEGORY Math

LOCATION <https://github.com/GuardianAudits/FastlaneScan/blob/7910434ed048d9da1019b04e5684443e9c08ab4b/src/shmonad/FLERC4626.sol#L286-L300>

REVIEW Main Review 1

DESCRIPTION

Deposits mint shares via the virtual-offset formula `assets*(realSupply+1)/(equity+1)`, while `cbPooledShareRate` validates the real post-state pooled share rate `equity/realSupply` with only a 1-wei rounding tolerance. Once yield or a permissionless boost raises equity above supply, the virtual formula mints more shares than the real pre-deposit rate permits for sufficiently large deposits, so the post-state real rate decreases beyond the 1-wei tolerance and the circuit breaker reverts despite `previewDeposit()` and `maxDeposit()` indicating the deposit is valid. The updated PoC uses the production-style 20 MON bootstrap, applies a permissionless 100 MON boost, and shows a 1000 MON deposit reverts purely due to the pricing/breaker mismatch.

Impact: After organic yield growth or a permissionless boost creates a high-rate state, large but otherwise-valid `deposit()/depositAndCommit()` calls can become unexecutable.

RECOMMENDATION

Align deposit minting and the circuit breaker: either mint deposits using the real pre-deposit exchange rate when real supply exists, or update the breaker to use the same virtual offsets used for pricing. Optionally cap per-deposit size based on the largest amount that cannot decrease the real share rate.

PROOF OF CONCEPT <https://gist.github.com/GuardianAudits/daaec615fc1c2a876be15a5af9e28387>

RESOLUTION

Status: Pending.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/760>

MEDIUM **M-19**

PENDING

Priority Fees Become Staking Yield

CATEGORY	Logical Error
LOCATION	https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/7910434ed048d9da1019b04e5684443e9c08ab4b/src/shmonad/StakeTracker.sol#L2640
REVIEW	Main Review 1

— DESCRIPTION

CoinbaseUpgradeable.process() can route coinbase-origin priority fees through Monad's staking precompile and later cause ShMonad to account them as native staking yield.

CoinbaseUpgradeable treats its raw MON balance, after unpaid buffers, as accrued priority fees. It splits the balance into validator commission, ShMonad donation, and priorityStakers. The priorityStakers portion is sent through sendRewardsToDelegates(), which calls externalReward() for the validator.

After that transfer, source information is lost. Monad records the value as validator delegator rewards. On a later validator crank, ShMonad calls claimRewards(), measures the received balance delta, and passes it to _handleEarnedStakingYield().

That function books the amount as staking yield: it can charge stakingCommission, mint owner zero-yield hMON, increase validator/global stakingYield allocation buckets, and queue the pooled portion for staking. This creates routing-dependent accounting for the same priority-fee source. Priority fees routed as a ShMonad donation are handled by _handleValidatorRewards() as non-staking revenue. Priority fees routed through priorityStakers instead return through claimRewards() and are booked as staking yield.

This conflicts with the documented revenue split, where staking yield and non-staking revenue are separate allocation buckets and MEV / boost / donation-style revenue belongs in the non-staking bucket. It also lets priority-fee revenue pay stakingCommission even though it was not native staking yield.

Example: a coinbase contract receives 2 MON of priority fees. With priorityCommissionRate = 10%, donationRate = 10%, and stakingCommission = 10%, process() pays 0.2 MON to the validator commission recipient, sends 0.2 MON to ShMonad as donation revenue, and routes 1.6 MON through externalReward() for priorityStakers. The donation is booked as non-staking revenue, but the 1.6 MON later returns through claimRewards(), enters _handleEarnedStakingYield(), is booked as staking yield, and mints 0.16 MON of owner zero-yield hMON.

— RECOMMENDATION

M-19 Priority Fees Become Staking Yield

CONTINUED

Preserve source metadata for coinbase-origin rewards sent through `externalReward()`, or avoid sending priority-fee revenue through the generic staking-yield claim path. The `priorityStakers` portion should either be accounted as non-staking revenue before forwarding, with the later claim offset from staking-yield buckets and `stakingCommission`, or claimed through a separate accounting path that does not treat `externalReward`-derived priority fees as native staking yield.

— RESOLUTION

Status: Pending.

MEDIUM **M-20**

PENDING

Coinbase Unpaid Rewards Can Strand

CATEGORY	DoS
LOCATION	https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/7910434ed048d9da1019b04e5684443e9c08ab4b/src/shmonad/StakeTracker.sol#L1784
REVIEW	Main Review 1

— DESCRIPTION

Coinbase reward buffers have no normal recovery path once the validator can no longer receive `externalReward()` payments from Monad's staking precompile.

When ShMonad settles validator rewards through a contract Coinbase, `_settleValidatorRewardsPayable()` sends `rewardsPayable` to `Coinbase.handleMEVPayable()`. Coinbase records the delegate portion, after any configured Coinbase MEV commission, in `s_unpaid.rewards` until `process()` can forward it through Monad's staking precompile.

The only normal exit path for `s_unpaid.rewards` is `Coinbase.process()` calling `sendRewardsToDelegates()`, which calls `externalReward(validatorId)`. Monad only allows `externalReward()` for validators currently in the consensus validator set, and the call reverts otherwise. If the call fails, Coinbase catches the failure, stores the amount back in `s_unpaid.rewards`, returns `false`, and keeps the MON in the Coinbase balance.

This can naturally happen when a validator has queued ShMonad `rewardsPayable` and then leaves Monad's consensus validator set before those rewards are processed. ShMonad may still treat the validator as locally active at reward-settlement time, so `_settleValidatorRewardsPayable()` routes the queued MON into the Coinbase contract and clears ShMonad's own reward liability and reserve. Once the value is inside Coinbase's unpaid reward buffer, ShMonad's inactive-validator redirect path can no longer recover it.

Example: validator V has 3 MON of `rewardsPayable`, a Coinbase proxy, and zero Coinbase MEV commission. V then undelegates enough stake, or otherwise leaves Monad's consensus validator set. On the next validator crank, ShMonad sends the 3 MON to Coinbase because V is still locally active at settlement time, and ShMonad clears its own liability and reserve. Coinbase records the 3 MON as unpaid delegate rewards. `Coinbase.process()` then attempts `externalReward(V)`, but the precompile reverts because V is not currently consensus-rewardable. Coinbase catches the failure and leaves the same 3 MON in `s_unpaid.rewards` and in its contract balance.

The funds can only move again if V becomes `externalReward()`-eligible in the future or governance upgrades the Coinbase implementation. Otherwise, the MON can remain stranded in the Coinbase contract indefinitely.

M-20 Coinbase Unpaid Rewards Can Strand

CONTINUED

— RECOMMENDATION

Add a recovery path for Coinbase unpaid rewards when `externalReward()` cannot succeed, such as redirecting them back to ShMonad yield, allowing owner-controlled rescue after validator deactivation, or requiring unpaid reward buffers to be zero before validator removal. Alternatively, prevent ShMonad from transferring `rewardsPayable` into Coinbase unless the validator is currently `externalReward()`-eligible.

— RESOLUTION

Status: Pending.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/763>

MEDIUM **M-21**

PENDING

Pending isolated stake dilutes pool yield

CATEGORY Rewards
 LOCATION StakeTracker.sol#L527
 REVIEW Main Review 1

DESCRIPTION

Staking-yield settlement is split between pooled and isolated stake using the validator's stored isolated snapshot:

```
``plain text
(uint256 pooledAmount, uint256 isolatedAmount) =
_splitIsolatedAndPooledBySnapshot(valId, amount, validatorEpochPtr_N(-1,
valId).targetStakeAmount);
```

The snapshot is updated at the end of `_crankValidator`:

```
`plain text
state.snapshotPooledStake = pooledStakeForNextEpoch.toUint120();
state.snapshotIsolatedStake = (uint256(state.activeStake) + state.pendingStake).toUint120();
```

This includes `pendingStake`. However, isolated deposits are first delegated to the Monad staking precompile and recorded as pending:

```
`plain text
state.pendingStake += assets.toUint120();
```

In the Monad staking precompile, delegated stake does not necessarily become productive immediately. Delegations activate at `epoch + 1`, or `epoch + 2` during the boundary window. Until activation, the delegated amount is pending delta stake and does not accrue staking rewards as active delegator stake.

As a result, an isolated depositor can deposit shortly before the ShMonad validator snapshot is taken. ShMonad will include that pending isolated deposit in the next staking-yield split, even though the precompile rewards being claimed may have been earned only by the existing active stake. This can route a portion of pooled staking yield to newly pending isolated stake.

This is different from depositing after the snapshot. A deposit after `_setIsolatedSnapshots()` does not affect the current split. The issue is that a deposit before the snapshot can be counted in the next snapshot before it is productive at the precompile.

M-21 Pending isolated stake dilutes pool yield

CONTINUED

— RECOMMENDATION

Consider using a staking-yield snapshot that only includes productive isolated stake.

— RESOLUTION

Status: Pending.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/767>

LOW **L-01**

PENDING

previewRequestUnstake Can Return a Success Quote for Requests That Later Revert on Slot Exhaustion

CATEGORY Best Practices

LOCATION <https://github.com/FastLane-Labs/fastlane-contracts-internal/blob/shmonad-v1.2/src/shmonad/ShMonad.sol#L283>

REVIEW Main Review 1

— DESCRIPTION

`ShMonad.previewRequestUnstake` only previews amount conversion and global-liability timing by calling `_calcUnstakeRequestCompletionEpoch()` on simulated liabilities. It does not inspect the caller's traditional unstake request state.

However, `requestUnstake()` later calls `TradUnstakeLib.prepare()` and `TradUnstakeLib.enqueuePendingRequest()`, which may revert with `NoUnstakeRequestSlotsAvailable` when the caller's bounded pending request slots are full and the new completion epoch cannot be merged into an existing bucket. Legacy request migration can also consume/require a slot before the new request is enqueued.

This makes the `NatSpec` inaccurate because it claims the preview uses the same queue-capacity logic and returns the epoch that `requestUnstake(shares)` would return if executed now. In reality, the preview can quote an epoch even though the state-changing request reverts.

Impact is not funds loss, since the revert rolls back the burn, but integrations may show users a request as available/successfully quotable when submission cannot succeed.

— RECOMMENDATION

Either make `previewRequestUnstake()` account-aware and simulate the caller's queue feasibility, including matured compaction, legacy migration, and merge/enqueue checks, or weaken the `NatSpec`/name to clarify that it only previews timing and does not guarantee `requestUnstake()` acceptance. A separate `canRequestUnstake(account, shares)` or `previewRequestUnstakeFor(account, shares)` would also make the integration contract clearer.

L - previewRequestUnstake Can Return a Success Quote for Requests That **CONTINUED**
01 Later Revert on Slot Exhaustion

— **RESOLUTION**

Status: Pending.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/769>

LOW **L-02**

PENDING

New Isolated Depositors Can Capture Surplus From Prior Pending Isolated Withdrawals

CATEGORY Best Practices
LOCATION StakeTracker.sol:2457
REVIEW Main Review 1

DESCRIPTION

When an isolated user burns all `vShares` into a locked exit, they stop being a shareholder and become a fixed redemption creditor. However, `depositIsolated()` still allows a new user to deposit into the same validator while the prior isolated withdrawal is pending.

If that old withdrawal later settles with surplus, `_finalizeCompletedIsolatedWithdrawal()` routes the surplus based on the current `s_isolatedVTotalShares[validatorId]`. If a new depositor has recreated non-zero isolated shares, the surplus is added to `queuedRestake`, which is counted as isolated equity. The new depositor's fresh shares therefore capture surplus generated by the previous user's already-pending withdrawal.

Without the new depositor, the surplus would remain idle and could be swept out once the old isolated claimants clear. The issue is that settlement surplus is attributed to the current shareholder set instead of the cohort that existed when the withdrawal was initiated.

Impact: value misallocation between isolated users. A new depositor can receive upside from a prior user's pending withdrawal, while the protocol's isolated accounting no longer preserves cohort ownership across settlement.

RECOMMENDATION

Route isolated withdrawal surplus according to the claimant/ shareholder state at withdrawal initiation, not current `s_isolatedVTotalShares`. In the simple fixed-creditor model, surplus from an ownerless settled withdrawal should remain idle and be swept rather than credited to newly created shares.

RESOLUTION

Status: Pending.

L - New Isolated Depositors Can Capture Surplus From Prior Pending
02 Isolated Withdrawals

CONTINUED

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/776>

LOW **L-03**

PENDING

Stale externalReward Gas Constant

CATEGORY	Configuration
LOCATION	https://github.com/FastLane-Labs/fastlane-contracts-internal/blob/shmonad-v1.2/src/shmonad/Constants.sol#L30
REVIEW	Main Review 1

— DESCRIPTION

The staking precompile by monad defines `externalReward` gas as `66_575`, while ShMonad uses `62_300` in `STAKING_GAS_EXTERNAL_REWARD`.

Because ShMonad adds a 10_000 gas buffer, calls currently still have enough gas, but the stale base constant reduces safety margin and can become brittle if the buffer is lowered or precompile gas increases.

— RECOMMENDATION

Consider updating `STAKING_GAS_EXTERNAL_REWARD` to `66_575`

— RESOLUTION

Status: Pending.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/768>

LOW **L-04**

PENDING

Redundant second scan: unstake completion

CATEGORY	Gas Optimization
LOCATION	https://github.com/FastLane-Labs/fastlane-contracts-internal/blob/shmonad-v1.2/src/shmonad/libraries/TradUnstakeLib.sol#L146
REVIEW	Main Review 1

— DESCRIPTION

`completeUnstake()` first calls `TradUnstakeLib.prepare()`, which compacts all matured pending requests into the dedicated matured

slot and migrates any legacy request.

After this normalization, the expected invariant is that pending slots contain only future requests and the matured slot contains all currently claimable amount.

However, `consumeMaturedRequests()` then scans all request slots again, rechecking every slot's

`completionEpoch` to aggregate

matured amount and find the oldest pending epoch.

This repeats maturity classification work already performed by `prepare()`.

Impact is limited because the request set is bounded to 8 slots, so this is a small gas inefficiency rather than a correctness issue.

— RECOMMENDATION

have `prepare()` return the oldest pending completion epoch after compaction/migration, and make the consume path drain only the dedicated matured slot. This would avoid the second full scan while preserving the same behavior.

— RESOLUTION

Status: Pending.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/771>

LOW **L-05**

PENDING

Allocation split start semantics

CATEGORY Validation
 LOCATION StakeTracker.sol#L578
 REVIEW Main Review 1

DESCRIPTION

The allocation bridge uses a single boolean to decide whether an entire two-epoch smoothing window should be read from legacy `earnedRevenue` or from the new split allocation buckets:

```
``plain text
bool useLegacyWeighting = _useLegacyAllocationWeightingForCurrentRound();
`
```

Both revenue entries in the smoothing window are then interpreted the same way:

```
`plain text
_globalAllocationRevenueWindow(-2, useLegacyWeighting)
_globalAllocationRevenueWindow(-1, useLegacyWeighting)
`
```

This means the implementation does not support a mixed boundary window where one epoch is read from legacy revenue and the other is read from split `stakingYield` / `nonStakingRevenue` buckets.

This is safe if `s_allocationSplitStartEpoch` means:

```
`vb.net
the first allocation round that may use split weighting
`
```

rather than:

```
`vb.net
the first revenue epoch recorded using split buckets
``
```

Example

Assume the upgrade occurs at internal epoch `E`.

```
``vb.net
splitStartEpoch = E + 3
`
```

The first split-weighted allocation round is `E + 3`. That round reads the two completed revenue windows:

L-05 Allocation split start semantics

CONTINUED

```
`` vb.net
```

```
E + 1
```

```
E + 2
```

```
`
```

Both were recorded after the v1.2 upgrade, so both should have split buckets populated. No mixed legacy/split window is needed.

However, if `splitStartEpoch` were interpreted as "the first split revenue epoch", then a boundary window such as:

```
` vb.net
```

```
epoch 4 = legacy revenue only
```

```
epoch 5 = split buckets populated
```

```
``
```

would ideally be read as a mixed window. The current all-or-nothing boolean would instead read both entries as legacy or both entries as split, depending on the round.

— RECOMMENDATION

Document `s_allocationSplitStartEpoch` explicitly as the first allocation round that uses split weighting, not the first epoch that records split revenue.

If future migration code allows this value to be configured manually, enforce a sufficient delay so both smoothing-window entries are populated under split accounting before the bridge switches away from legacy weighting.

— RESOLUTION

Status: Pending.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/772>

LOW **L-06**

PENDING

Atomic unstake fees are immediately capturable by JIT deposits

CATEGORY	MEV
LOCATION	https://github.com/GuardianAudits/FastlaneScan/blob/7910434ed048d9da1019b04e5684443e9c08ab4b/src/shmonad/StakeTracker.sol#L2860-L2888
REVIEW	Main Review 1

DESCRIPTION

Atomic unstake paths calculate a fee and pass it to `_accountForWithdraw()`, but `_accountForWithdraw()` only increments `s_atomicAssets.distributedAmount` by the net amount paid out and ignores the fee argument. The fee is not added to global earnedRevenue, the revenue smoother, or any recent-revenue offset. Because the victim's shares are burned against gross value while only net MON leaves, the fee remains as immediate pooled equity. A searcher can deposit immediately before a large atomic withdrawal at the pre-fee price, then hold shares whose `convertToAssets` value includes a pro-rata share of the victim-paid fee; the PoC realizes that gain through the public traditional unstake path.

Impact: Searchers can redirect atomic unstake fee revenue from long-term shMON holders/liquidity providers to short-lived JIT capital. This defeats the protocol's anti-JIT revenue smoothing for an ordinary protocol fee stream and dilutes passive holders whenever large atomic exits or policy-agent instant withdrawals are predictable. The value at risk scales with atomic withdrawal volume, utilization, and fee-curve parameters; even though the searcher must wait for traditional unstake completion to avoid paying an atomic exit fee, the extracted value is paid by other users and is not earned by the JIT capital's historical liquidity provision.

RECOMMENDATION

In `_accountForWithdraw()`, account the fee portion as earned/recent revenue and include it in `_recentRevenueOffset()` smoothing, or allocate atomic fee revenue via a snapshot/accumulator that excludes shares minted during the smoothing window. Apply the same accounting to both ERC4626 withdraw/redeem and agentWithdrawFromCommitted paths.

RESOLUTION

Status: Pending.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/773>

LOW **L-07**

PENDING

UNKNOWN/inactive-proposer boost yield bypasses smoothing and enables JIT capture

CATEGORY Math

LOCATION <https://github.com/GuardianAudits/FastlaneScan/blob/7910434ed048d9da1019b04e5684443e9c08ab4b/src/shmonad/StakeTracker.sol#L2154-L2186>

REVIEW Main Review 1

DESCRIPTION

`boostYield` attributes donations to the current proposer validator. If the proposer lookup returns zero/UNKNOWN or returns a validator that is not active/known in `ShMonad`, `_handleBoostYield` credits UNKNOWN or validator-specific buckets and queues the MON to stake, but does not increment `globalRevenue` or `globalAllocationRevenue`. Share conversions only deduct `_recentRevenueOffset`, which is based on `globalRevenue`, so inactive/UNKNOWN-proposer boost yield is immediately included in pooled equity instead of being smoothed like active-validator boost yield. Production triggers include (a) consensus validators not yet onboarded into ShMonad's ValidatorRegistry, (b) ShMonad-deactivated validators still proposing at the native layer during exit/unstaking lag, and (c) transient precompile zero/revert cases mapped to UNKNOWN.

Impact: A just-in-time depositor in a block proposed by an unregistered, ShMonad-inactive/deactivated, or UNKNOWN proposer can deposit before a large permissionless `boostYield` (including paymaster-triggered boosts) and redeem immediately after to capture donation value that should have been smoothed to existing holders. Under default fee/curve parameters, the PoC profits ~16.10 MON net of atomic-unstake fees from a 100 MON JIT deposit around a 100 MON boost. The omission also distorts global allocation revenue signals used for stake allocation.

RECOMMENDATION

Record pooled boost yield in `globalRevenue` and the appropriate `globalAllocationRevenue` bucket regardless of whether the current proposer is active/known. If validator-specific attribution must remain UNKNOWN, separate validator attribution from global revenue smoothing and allocation accounting so all pooled revenue is included in `_recentRevenueOffset`.

PROOF OF CONCEPT <https://gist.github.com/GuardianAudits/fc345f834c96891c0fca7cc461547be1>

L - UNKNOWN/inactive-proposer boost yield bypasses smoothing and
07 enables JIT capture

CONTINUED

— **RESOLUTION**

Status: Pending.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/775>

LOW **L-08**

PENDING

Validator crank can skip global epoch rolls

CATEGORY Math

LOCATION <https://github.com/GuardianAudits/FastlaneScan/blob/7910434ed048d9da1019b04e5684443e9c08ab4b/src/shmonad/StakeTracker.sol#L763-L794>

REVIEW Main Review 1

— DESCRIPTION

When validator work is pending, `crank()` skips `_crankGlobal()` and enters `_crankValidators()`. `_crankValidators()` synchronizes `globalEpochPtr_N(0).epoch` to `_getEpochBarrierAdj()` even though `ShMonad` has not executed the global roll for that `Monad` epoch and has not incremented `s_admin.internalEpoch`. After the remaining validators are cranked, a later `_crankGlobal()` compares this stamped value to the unadjusted `_getEpoch()` and returns true at L820, so the global accounting path is skipped while `crank()` reports success. The PoC repeats this with only public `crank()` calls: a low-gas crank leaves validators pending after a global roll, `Monad` advances, validator cranking stamps the unrolled slot to the new native epoch, and the next crank skips the global roll.

Impact: Each successful repetition consumes two native `Monad` epochs while advancing `ShMonad` internal accounting by only one epoch. Traditional and isolated unstake completion, queue processing, atomic-unstake carry-over, and revenue smoothing are keyed to the skipped internal global rolls. The PoC demonstrates a traditional unstake request that remains immature and cannot be completed even after enough native epochs have elapsed for its quoted completion delay. A continued attacker can extend the redemption/cash-flow delay by repeating the timing pattern, although honest full-gas cranking during the pending-validator window can restore normal progress.

— RECOMMENDATION

Do not mutate the current global epoch marker from `_crankValidators()`. Only update `globalEpochPtr_N(0).epoch` as part of a successful `_crankGlobal()` roll, or track validator-crank synchronization in separate state that `_crankGlobal()` does not use as proof the global epoch was processed. Additionally, `_crankGlobal()` should verify that the internal/global accounting roll for the `Monad` epoch actually executed before returning true.

PROOF OF CONCEPT <https://gist.github.com/GuardianAudits/fc345f834c96891c0fca7cc461547be1>

L-08 Validator crank can skip global epoch rollsCONTINUED

— RESOLUTION

Status: Pending.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/774>

LOW **L-09**

PENDING

Isolated deposit can mint zero shares

CATEGORY MEV
 LOCATION StakeTracker.sol#L3002
 REVIEW Main Review 1

DESCRIPTION

`_depositIsolated()` does not enforce a minimum amount of isolated vShares to mint. It prices deposits with the virtual-offset formula:

```
``plain text
vShares = assets * (s_isolatedVTotalShares[valId] + 1) / (_isolatedValidatorEquity(valId) + 1);
`
```

This can mint zero shares when the validator has positive isolated equity but no outstanding isolated vShares. That state can occur after all isolated vShares have been burned into locked redemption requests, while `redemptionsPayable` keeps the isolated validator state alive, and additional isolated-attributed rewards or withdrawal surplus later increase isolated assets. In that state, there is no real share supply, but `_isolatedValidatorEquity()` can be positive. A fresh isolated depositor is then priced against only the one virtual share in the numerator. If the residual equity exceeds the deposit amount, integer division can round the minted vShares down to zero. The deposit still succeeds, delegates the user's MON, and increments isolated assets, but the receiver receives no claim.

Example

- Bob is the only isolated staker for validator `V` with 100 MON and later requests a full isolated unstake.
- Bob's vShares are burned immediately:

```
``vb.net
s_isolatedVTotalShares[V] = 0
state.redemptionsPayable = 100 MON
isolated equity = totalAssets - redemptionsPayable = 0
`
```

- Before Bob completes, 20 MON of isolated-attributed rewards or isolated withdrawal surplus is credited to `V`.

```
``vb.net
s_isolatedVTotalShares[V] = 0
```

L-09 Isolated deposit can mint zero shares

CONTINUED

```
state.redemptionsPayable = 100 MON
```

```
isolated totalAssets = 120 MON
```

```
isolated equity = 20 MON
```

```
``
1. Alice deposits 1 MON isolated into V.
`vb.net
vShares = 1 MON * (0 + 1) / (20 MON + 1) = 0
``
```

Alice's MON is accepted and staked, but she receives zero isolated vShares.

This could be used by malicious users to cause fund loss for newly enabled isolated validators for their first deposits.

— RECOMMENDATION

Consider adding slippage protection to isolated deposits, and reject zero-share mints.

For example:

```
``plain text
function depositIsolated(
uint64 validatorId,
uint256 assets,
address receiver,
uint256 minVSharesOut
) external payable returns (uint256 vShares) {
vShares = _depositIsolated(_msgSender(), validatorId, assets, receiver);
require(vShares >= minVSharesOut, IsolatedDepositSlippageExceeded(vShares, minVSharesOut));
}
``
```

— RESOLUTION

Status: Pending.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/777>

LOW **L-10**

PENDING

Isolated unstake can leave dust position

CATEGORY	Logical Error
LOCATION	StakeTracker.sol#L3056
REVIEW	Main Review 1

— DESCRIPTION

`depositIsolated()` requires each isolated deposit to be at least `MIN_VALIDATOR_DEPOSIT`, but `_requestUnstakeIsolated()` allows a user to burn any nonzero amount of vShares as long as the burned shares convert to a nonzero MON amount.

This means the minimum deposit threshold is only an entry-time rule. It does not ensure that isolated accounts maintain a minimum validator-specific position after partial unstaking. A user can deposit at least `MIN_VALIDATOR_DEPOSIT`, then immediately request an isolated unstake for most of the position, leaving a very small residual vShare balance.

Small residual isolated positions still participate in validator-local accounting, reward splitting, snapshots, deactivation cleanup, and direct-claim flows. This creates unnecessary state and crank complexity that the minimum deposit rule appears intended to avoid.

Example

1. Alice deposits `1 MON` isolated into validator `V`, satisfying `MIN_VALIDATOR_DEPOSIT`.
2. Alice immediately calls `requestUnstakeIsolated()` for shares worth `0.999999999 MON`.
3. The request succeeds because `vShares != 0` and the burned shares convert to a nonzero `amountMon`.
4. Alice is left with an isolated position worth roughly `1 wei`.

That residual position remains in the isolated share supply and must continue to be handled by the protocol.

— RECOMMENDATION

L-10 Isolated unstake can leave dust position

CONTINUED

When processing an isolated unstake request, enforce that the user's remaining isolated position for that validator is either zero or at least `MIN_VALIDATOR_DEPOSIT`.

For example:

```
``plain text
uint256 remainingShares = availableShares - vShares;
if (remainingShares != 0) {
uint256 remainingAssets = _convertToIsolatedAssets(validatorId, remainingShares);
require(remainingAssets >= MIN_VALIDATOR_DEPOSIT, IsolatedRemainderBelowMinimum());
}
``
```

— RESOLUTION

Status: Pending.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/778>

LOW **L-11**

PENDING

Late Isolated Deposit Captures Past Rewards

CATEGORY	Logical Error
LOCATION	https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/7910434ed048d9da1019b04e5684443e9c08ab4b/src/shmonad/StakeTracker.sol#L370
REVIEW	Main Review 1

— DESCRIPTION

ShMonad uses validator-level isolated stake snapshots to decide how much staking yield belongs to isolated stake versus pooled stake, but the isolated portion is later distributed over the validator's current isolated share supply. This allows isolated shares minted after the snapshot to capture rewards attributable to the prior isolated cohort.

`_setIsolatedSnapshots()` records only aggregate stake amounts: `snapshotPooledStake` and `snapshotIsolatedStake`. It does not checkpoint the isolated share supply or the users who held isolated shares at the snapshot. Later, `_handleEarnedStakingYield()` calls `_splitIsolatedAndPooledBySnapshot()` to classify the reward between pooled and isolated stake using that snapshot.

The classified isolated reward is then passed to `_creditIsolatedRewards()`. That function increases current isolated validator equity, usually by adding the amount to `queuedRestake` when the validator is active and `s_isolatedVTotalShares[validator]` is nonzero. Since isolated redemptions use current `vShares` pro-rata against current isolated equity, any shares minted after the snapshot but before reward settlement receive part of the prior reward. This also applies when pending staking rewards are visible before settlement: an attacker can observe pending rewards, deposit isolated stake after the snapshot but before ShMonad settles them, and then share in the isolated portion once it is credited. Unlike pooled shMON, isolated `vShares` have no equivalent per-validator recent-revenue offset or reward-debt accounting to prevent this.

For example, Bob deposits 10 MON isolated into a validator and becomes the only isolated stake in the snapshot. Alice then deposits 10 MON isolated after the snapshot but before staking yield is settled. When a 4 MON of staking yield is handled, the snapshot correctly classifies the reward as isolated. However, the 4 MON is added to current isolated equity, where Bob and Alice each own roughly half the current isolated shares. Alice's redeemable isolated assets increase by about 2 MON despite not being part of the reward-earning snapshot cohort.

— RECOMMENDATION

L-11 Late Isolated Deposit Captures Past Rewards

CONTINUED

Do not distribute snapshot-attributable isolated rewards over the current isolated share supply. Maintain per-validator isolated reward index accounting with user reward debt, checkpoint isolated share supply at snapshot time, or stage post-snapshot isolated deposits so they become reward-eligible only after the next snapshot.

— RESOLUTION

Status: Pending.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/776>

LOW **L-12**

PENDING

Pooled rate breaker has coverage gaps

CATEGORY Best Practices

LOCATION FLERC4626.sol#L390

REVIEW Main Review 1

DESCRIPTION

`cbPooledShareRate` checks that an external action does not decrease the real pooled shMON share rate:

```
`plain text
```

```
uint256 previousEquity = _pooledShareRateEquitySnapshot();
```

```
uint256 previousSupply = _realTotalSupply();
```

```
_;
```

```
_assertPooledShareRateNotDecreased(previousEquity, previousSupply);
```

```
`
```

The breaker is applied to the main pooled mint/burn paths, including `deposit`, `mint`, `atomic withdraw / redeem`, `traditional requestUnstake`, `committed-balance instant withdrawal`, `share-burn boost yield`, and `zero-yield conversion to shMON`.

However, several external accounting paths that move MON through shared accounting buckets or mutate isolated carve-outs are not protected by the same invariant. Examples include:

- `native boostYield`
- `sendValidatorRewards`
- `depositToZeroYieldTranche`
- `completeUnstake`
- `depositIsolated`
- `requestUnstakeIsolated`
- `completeUnstakeIsolated`
- `claimDeactivatedIsolated`

Some of these paths are intended to be pooled-rate increasing, and isolated paths should be pooled-rate neutral after `s_isolatedTotalAssets` and `s_isolatedLiquidAssets` are carved out.

Still, the absence of the circuit breaker means those assumptions are not enforced at the external boundary. This increases the risk that a future accounting change or edge-case interaction silently decreases pooled share rate without being caught immediately.

The isolated paths are especially worth considering because they mutate accounting variables used by pooled equity calculations:

```
`plain text
```

```
_totalEquity(false) = totalEquity(..., s_isolatedTotalAssets)
```

L-12 Pooled rate breaker has coverage gaps

CONTINUED

...

An isolated operation should not reduce pooled equity after the isolated carve-out. Applying the pooled share-rate breaker would make that invariant explicit.

— RECOMMENDATION

Apply `cbPooledShareRate` consistently to external functions that mutate pooled accounting, shared liabilities/reserves, or isolated accounting values that feed the pooled equity carve-out.

At minimum, consider adding the breaker to:

```
``plain text
boostYield(address)
sendValidatorRewards(uint64,uint256)
depositToZeroYieldTranche(uint256,address)
completeUnstake()
depositIsolated(uint64,uint256,address)
requestUnstakeIsolated(uint64,uint256)
completeUnstakeIsolated(uint64)
completeUnstakeIsolatedFor(uint64,address)
claimDeactivatedIsolated(uint64,uint256,address)
claimDeactivatedIsolatedFor(uint64,uint256,address)
``
```

— RESOLUTION

Status: Pending.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/780>

INFORMATIONAL I-01

PENDING

recomputeCurrentAllocationTotals — validator cap not enforced at upgrade time

CATEGORY Warning
LOCATION StakeTracker.sol:3562.
REVIEW Main Review 1

— DESCRIPTION

`_recomputeCurrentAllocationTotals` at StakeTracker.sol:646 walks the full validator linked list with no gas break or pagination. For a fresh deployment this is bounded by `MAX_ACTIVE_VALIDATORS_FOR_ALLOCATION`, enforced in `_assertValidatorCanBeAdded` at StakeTracker.sol:3562.

For an upgrade path (existing deployment → v1.2), the cap is only applied to validators added after the upgrade. If the live deployment already had more active validators than `MAX_ACTIVE_VALIDATORS_FOR_ALLOCATION` at the time of the upgrade, the prepass will iterate beyond the intended limit on every crank call. At a sufficient count this could approach block gas limits and make `crank()` uncallable, stalling epoch advancement.

This is distinct from the per-validator crank path, which is explicitly designed to be incremental and gas-bounded.

— RECOMMENDATION

For this specific deployment the risk depends on the active validator count at upgrade time. If it is already within the new cap it is a non-issue. The finding is worth noting as a deployment checklist item rather than a code defect: verify `s_activeValidatorCount <= MAX_ACTIVE_VALIDATORS_FOR_ALLOCATION` holds before executing the upgrade, or add an assertion to `initialize()` that enforces this invariant at upgrade time.

— RESOLUTION

Status: Pending.

INFORMATIONAL I-02

PENDING

Informationals about Upgrade Plan

CATEGORY Best Practices
LOCATION SHMONAD_V1_2_UPGRADE_PLAN.md
REVIEW Main Review 1

DESCRIPTION

1. Time-Sensitive Allocation Split Validation

The plan currently verifies:

```
`plain text
getAllocationSplitStartEpoch() == internalEpoch + 3
`
```

This condition only holds immediately at upgrade execution. Since `crank()` is permissionless, `internalEpoch` may advance before operators read it.

****Recommended approach:****

Validate against the epoch at execution time:

```
`plain text
getAllocationSplitStartEpoch() == internalEpoch_at_upgrade_execution + 3
`
```

****Execution artifact should capture:****

- `internalEpoch_at_upgrade_execution`
- Expected split start epoch
- Actual split start epoch
- Transaction hash
- Block number

2. Pre-Flight Should Require Crank Idleness

The plan verifies that ShMonad is not frozen, but should also require:

```
`plain text
getNextValidatorToCrank() == address(0)
`
```

****When to enforce:****

- Before upgrade execution
- Before validator migration batches

This ensures no validator crank round is partially processed during critical operations.

3. Coinbase Migration Batch Validation per Validator

Before calling:

```
`plain text
```

I-02 Informationals about Upgrade Plan

CONTINUED

```
updateCoinbaseForExistingValidator(validatorId)
```

```
`
```

```
Require the following checks:
```

```
`plain text
```

```
isValidatorActive(validatorId) == true
```

```
getValidatorCoinbase(validatorId) != address(0)
```

```
previewCoinbaseAddress(validatorId) matches expected operator target
```

```
`
```

```
**Benefit:**
```

```
Reduces risk of incorrect validator IDs in Safe transaction batches.
```

```
## 4. hMON holder snapshot can go stale before activation.
```

Phase 1 captures the holder manifest, Phase 2 generates the Safe batch, and Phase 4 executes it. During that gap, new zero-yield holders can appear or balances can change unless the relevant state-expansion paths are closed.

The plan does require manifest total equals totalZeroYieldPayable at the same block height, but it does not require a final rescan immediately before activation or closing zero-yield deposits/conversions around the snapshot. This can leave late holders without the discovery `Transfer(address(0), holder, balance)`` event.

— RECOMMENDATION

Beware of these considerations during upgrade.

— RESOLUTION

Status: Pending.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/779>

INFORMATIONAL I-03

PENDING

Deactivated Validator and Stake Allocation

CATEGORY Informational
LOCATION StakeAllocationLib
REVIEW Main Review 1

— DESCRIPTION

Deactivated validators are excluded from active-set stake/allocation totals immediately via `isActive == false`, but their previously recorded revenue remains in the global allocation revenue windows. This can temporarily inflate the global weighted revenue denominator while the validator itself is no longer eligible to receive allocation. Impact is limited: the inactive validator cannot receive new stake because its computed allocation is overridden into forced withdrawal/requeue, but active validators may receive less stake allocation for the smoothing window, causing temporary queue carry-forward or slower redeployment.

— RECOMMENDATION

Beware of this consideration, if you intend to resolve this, we dont see easy way to do so without introducing new state which may not be worth considering the likelihood of this case.

— RESOLUTION

Status: Pending.

HIGH **H-01**

Sub-Dust Isolated Exit Freezes Protocol

CATEGORY	DoS
LOCATION	https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/14e40ee4491eb0b7454258b2488f43fbe48e424b/src/shmonad/StakeTracker.sol#L2348
REVIEW	Remediation Review 1

— DESCRIPTION

Full isolated exits can leave a sub-dust native stake remainder that Monad rounds into the withdrawal, while ShMonad still accounts for the remainder as active stake.

Isolated redemptions are priced with virtual-share rounding. If an isolated position receives a tiny validator-attributed reward before the holder exits, `requestUnstakeIsolated()` can lock slightly less than the validator's total isolated assets. The reward is reserved as idle liquidity, and only the remaining principal is queued for native unstaking.

When that queued un stake would leave less than Monad's `DUST_THRESHOLD` on the validator, the Monad staking precompile rounds the undelegation up and withdraws the full remaining native stake. ShMonad, however, records only the requested amount as pending un stake and leaves the dust remainder in `activeStake`.

On the next validator crank, balance verification compares ShMonad's expected active stake against the real precompile stake. Since the real stake is zero but ShMonad still expects dust, `_handleStakeBalanceVerification()` treats this as a principal deficit and sets the global epoch to frozen and closed.

This is permissionlessly triggerable once an active registered validator has no pooled ShMonad allocation. That state is reachable through normal flows: a newly added validator starts with pooled `targetStakeAmount == 0`, and any user can make an isolated deposit into it. The attacker can then send a 1 wei validator reward through `sendValidatorRewards(validatorId, SCALE)`, fully exit the isolated position, and call the public `crank()` function themselves across the required epoch transitions. No keeper, proposer control, or privileged role is needed to drive the protocol into the freeze.

Owner unfreezing/reopening is insufficient by itself, because the stale isolated `activeStake` dust remains and the next crank re-freezes the protocol.

— RECOMMENDATION

H-01 Sub-Dust Isolated Exit Freezes Protocol

CONTINUED

Prevent isolated native unstake requests that would leave a nonzero remainder below Monad's dust threshold. When queueing a full isolated exit, either round the ShMonad-accounted unstake amount up to the full active stake if the remainder would be dust, or keep isolated stake accounting synchronized with the precompile's dust-rounding behavior. Balance verification should not be the first place this mismatch is handled.

PROOF OF CONCEPT <https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/870a8a78c5ee0897ee420c1d1c37214163c0165a/test/shmonad/SubDustIsolatedExitFreezePoC.t.sol#L8>

— RESOLUTION

Status: Reported during the fix review period.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/788>

MEDIUM **M-01**

Reward Retry Reuses Settled Buckets

CATEGORY	Logical Error
LOCATION	https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/14e40ee4491eb0b7454258b2488f43fbe48e424b/src/shmonad/StakeTracker.sol#L1872
REVIEW	Remediation Review 1

— DESCRIPTION

As part of the fix for `M-16: Deficit Can Lose Unstake Demand`, validators that halt after balance verification were made retryable by clearing `wasCranked`. This recovery behavior is not idempotent for reward settlement.

During `_crankValidator()`, ShMonad first marks the previous validator epoch as cranked, then calls `_settleValidatorRewardsPayable()` before balance verification. Reward settlement reads `validatorRewardsPtr_N(-1, valId).rewardsPayable` and may pay or redirect that amount. When settlement succeeds or redirects, ShMonad decreases the global reward liability and reserve, but it does not clear the settled amount from the validator-local historical bucket.

If the crank halts after reward settlement but before the validator epoch is completed, the fix clears `wasCranked`. After recovery, the same validator can be cranked again for the same epoch. The retry re-enters `_settleValidatorRewardsPayable()` and reads the same stale historical reward bucket, so rewards that were already settled can be processed again. This leaves validator-local reward payable state larger than the global reward liability and reserve backing it.

This creates accounting drift and can also pin crank progress. In the partial direct-payment path, stale rewards can be booked or paid again if ShMonad has available balance. In the inactive-validator redirect path, the first crank can clear global accounting while leaving the stale local bucket, and a later retry can revert from underflow until manual correction or upgrade.

— RECOMMENDATION

Clear or decrement `validatorRewardsPtr_N(-1, valId).rewardsPayable` whenever rewards are paid or redirected. Alternatively, move retryable halt checks before reward settlement.

— RESOLUTION

Status: Reported during the fix review period.

M-01 Reward Retry Reuses Settled Buckets

CONTINUED

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/783>

MEDIUM **M-02**

Atomic Completion Leaves Paid Queue Live

CATEGORY	Logical Error
LOCATION	https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/14e40ee4491eb0b7454258b2488f43fbe48e424b/src/shmonad/StakeTracker.sol#L3010
REVIEW	Remediation Review 1

— DESCRIPTION

Traditional unstake requests create two linked records: `redemptionsPayable` and `queueForUnstake`. If the validator unstake queue is rolled forward because stake is temporarily unavailable, the request can later mature while its queue entry is still live.

During `completeUnstake()`, `_beforeCompleteUnstake()` can satisfy the matured redemption by pulling unutilized atomic liquidity into reserves. It then decreases `reservedAmount` and `redemptionsPayable`, but does not reduce the corresponding `queueForUnstake`.

As a result, a user's redemption can be fully paid while the validator unstake queue for that same redemption remains executable. An active validator can have stake temporarily unavailable due to pending staking, causing the unstake queue to roll. If the pending stake later activates before the stale queue is netted away, the next validator crank can initiate a native validator withdrawal even though `redemptionsPayable == 0`.

Atomic liquidity can be replenished separately from fresh current MON while the stale paid redemption queue remains executable in the validator round. This creates duplicate withdrawal pressure for an already-paid redemption and can unnecessarily move stake out of productive validator allocation.

— RECOMMENDATION

When `_beforeCompleteUnstake()` newly backs a traditional redemption using unutilized atomic liquidity, reduce the live aggregate non-turnover `queueForUnstake` by the newly atomic-backed redemption amount, capped to the portion not already backed by redemption reserves or live pending unstaking. Preserve turnover labels and avoid double-reducing queue already backed by live pending unstaking.

— RESOLUTION

Status: Reported during the fix review period.

MEDIUM **M-03**

Atomic Target Reuses Revenue Carryover

CATEGORY	Logical Error
LOCATION	https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/14e40ee4491eb0b7454258b2488f43fbe48e424b/src/shmonad/StakeTracker.sol#L2713
REVIEW	Remediation Review 1

— DESCRIPTION

During global crank, current-epoch revenue can be used twice in atomic liquidity accounting. `_settleGlobalNetMONAgainstAtomicUnstaking()` runs before `_carryOverAtomicUnstakeIntoQueue()`. When the atomic target increases, `_settleGlobalNetMONAgainstAtomicUnstaking()` can use current native assets to increase `s_atomicAssets.allocatedAmount`. If those assets are current-epoch revenue, the function does not mark the revenue as consumed in `globalRevenue.allocatedRevenue`.

Later in the same crank, `_carryOverAtomicUnstakeIntoQueue()` computes available revenue as `earnedRevenue - allocatedRevenue`. Since the revenue used to increase atomic allocation was not marked allocated, carryover treats the same MON as still available and reduces `s_atomicAssets.distributedAmount`. This can increase atomic liquidity through both sides of the formula. The same native revenue can raise `allocatedAmount` and reduce `distributedAmount`, causing `getCurrentLiquidity()`, `maxWithdraw()`, and `maxRedeem()` to report more atomic liquidity than is backed by current native MON after the same revenue has already been consumed.

The reused amount can also remain as live validator unstake pressure, so the protocol has inflated atomic availability and outstanding unstake demand for the same MON. A later atomic withdrawal can be quoted within `maxWithdraw()` / `maxRedeem()` and pass accounting checks, then fail at the native MON transfer if no other idle balance is available. This breaks the reliability of the ERC4626 liquidity views and can create temporary withdrawal failures until later validator withdrawals or deposits replenish native balance.

— RECOMMENDATION

When current-epoch revenue-backed assets are consumed by `_settleGlobalNetMONAgainstAtomicUnstaking()` to increase atomic allocation, mark that portion unavailable to `_carryOverAtomicUnstakeIntoQueue()` by increasing `allocatedRevenue` or by tracking crank-local consumed revenue. Alternatively, reorder or combine atomic target settlement and revenue carryover so one unit of revenue can affect only one side of the atomic liquidity formula.

M-03 Atomic Target Reuses Revenue Carryover

CONTINUED

— RESOLUTION

Status: Reported during the fix review period.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/785>

MEDIUM **M-04**

Inactive validator rewards bypass smoothing

CATEGORY MEV

LOCATION `src/shmonad/StakeTracker.sol::_handleValidatorRewards`

REVIEW Remediation Review 1

DESCRIPTION

The audit-fix branch updated `boostYield(address)` so rewards received while the proposer is unknown or inactive are still recorded in `globalRevenuePtr_N(0).earnedRevenue`. This ensures the ERC4626 recent-revenue smoothing offset applies to pooled shMON yield.

However, `sendValidatorRewards()` still has an inactive/unregistered validator branch that credits pooled yield only to the `UNKNOWN_VAL_ID` validator bucket:

```
``plain text
validatorRewardsPtr.earnedRevenue += pooledYield.toUint120();
validatorAllocationRevenuePtr_N(0, UNKNOWN_VAL_ID).nonStakingRevenue +=
pooledYield.toUint120();
```

```
`
It does not increment:
```

```
`plain text
globalRevenuePtr_N(0).earnedRevenue;
`
```

As a result, MON sent through `sendValidatorRewards()` for an inactive, deactivated, unregistered, or `UNKNOWN_VAL_ID` validator immediately increases pooled equity without entering `_recentRevenueOffset()`.

ERC4626 exit pricing uses `_totalEquity(true)`, which subtracts recent revenue. Since this reward is not recorded as global revenue, it bypasses the smoothing defense intended to prevent JIT capture of fresh yield.

This is inconsistent with the fixed `boostYield()` inactive-proposer branch, which now treats inactive or unknown proposer revenue as pooled shMON revenue that must still be smoothed.

Example

1. The vault has 1000 MON equity and 1000 shMON supply.
2. An attacker sees a pending 1000 MON reward routed through `sendValidatorRewards(UNKNOWN_VAL_ID, 0)`.
3. The attacker deposits 1000 MON, receiving about 1000 shMON.
4. The 1000 MON reward lands through the inactive/unknown branch.
5. Because `globalRevenuePtr_N(0).earnedRevenue` is not incremented, smoothed equity

M-04 Inactive validator rewards bypass smoothing

CONTINUED

immediately becomes `3000 MON`, not `2000 MON`.

6. The attacker redeems/unstakes their `1000 shMON` against the unsmoothed rate and captures roughly half of the fresh reward.

If the same value had entered through the fixed `boostYield()` path, `globalRevenuePtr_N(0).earnedRevenue` would increase and `_recentRevenueOffset()` would prevent immediate capture.

Fresh pooled yield can bypass recent-revenue smoothing and become immediately claimable by short-lived depositors. This creates a JIT deposit/redeem value extraction vector around inactive or unknown validator rewards.

— RECOMMENDATION

In the inactive/unregistered branch of `_handleValidatorRewards()`, increment `globalRevenuePtr_N(0).earnedRevenue` by the pooled amount so ERC4626 smoothing applies.

— RESOLUTION

Status: Reported during the fix review period.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/789>

MEDIUM **M-05**

Raw Breaker Misses Smoothed Rate Drop

CATEGORY	Configuration
LOCATION	src/shmonad/FLERC4626.sol#L382
REVIEW	Remediation Review 1

— DESCRIPTION

The pooled share-rate circuit breaker checks the raw pooled equity rate, while user-facing ERC4626 exit pricing uses the smoothed equity rate.

`cbPooledShareRate` snapshots and verifies `_totalEquity(false)`. However, `convertToAssets()`, `previewRedeem()`, and `previewUnstake()` call `_totalEquity(true)`, which subtracts `_recentRevenueOffset()`.

During an existing revenue smoothing window, `boostYield(shares, from, originator)` values the burned shares against raw equity with `deductRecentRevenue=false`. For an active proposer with atomic liquidity headroom, `_handleBoostYieldFromShares()` then increases current `earnedRevenue`. Raw equity per share can increase because supply was burned, so the breaker passes. But the smoothed equity used by exits can decrease because the new `earnedRevenue` is added to the recent-revenue offset.

The result is a guarded operation that passes the pooled share-rate breaker while lowering `previewRedeem()`, `previewUnstake()`, and other smoothed exit quotes for existing users. The omitted value remains in pooled equity and is redistributed to remaining holders as smoothing decays.

This leaves a gap in the recent share-rate breaker coverage: the guard enforces the virtual rate on the raw equity basis, while ERC4626 exit quotes are computed on the smoothed equity basis.

— RECOMMENDATION

Check the same equity basis used by user-facing exit pricing. Either make `cbPooledShareRate` also enforce non-decrease of `_totalEquity(true)` per share, or prevent shares-based boost accounting from increasing the recent-revenue offset in a way that reduces smoothed exit quotes.

— RESOLUTION

Status: Reported during the fix review period.

M-05 Raw Breaker Misses Smoothed Rate Drop

CONTINUED

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/793>

LOW **L-01**

Hardcoded staking deadband limits

CATEGORY	Configuration
LOCATION	/shmonad/libraries/StakeAllocationLib.sol:145
REVIEW	Remediation Review 1

— DESCRIPTION

The staking-yield deadband is hardcoded as `STAKING_YIELD_DEADBAND_BPS = 8000`.

This value directly controls how much stake-proportional staking yield is filtered out before allocation scoring.

Because the correct deadband depends on live validator commission rates, reward variance, MEV distribution, and whether isolated rewards are included in allocation signals, a fixed value can become economically miscalibrated.

If too high, staking yield is mostly ignored; if too low, incumbent validators continue to dominate allocation through stake-proportional yield.

— RECOMMENDATION

Move the deadband into `StakeAllocationConfig`, expose a bounded owner/governance setter, and snapshot the value per allocation round like `nonStakingRevenueMultiplierRay` so global totals and validator crank calculations use the same parameter

— RESOLUTION

Status: Reported during the fix review period.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/792>

LOW **L-02**

Delay Crank Skips Global Epoch Roll

CATEGORY	Logical Error
LOCATION	https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/14e40ee4491eb0b7454258b2488f43fbe48e424b/src/shmonad/StakeTracker.sol#L1081
REVIEW	Remediation Review 1

— DESCRIPTION

The L-08 remediation removed validator-side global epoch stamping, but `_crankGlobal()` can still future-stamp the global epoch marker during the Monad staking delay window.

`_crankGlobal()` checks availability against the unadjusted `_getEpoch()`. However, when the roll executes, `_primeNextGlobalEpoch()` stores `_getEpochBarrierAdj()`, which returns `reportedEpoch + 1` while `inEpochDelayPeriod` is true. After `s_admin.internalEpoch` is bumped, this future value becomes the current `globalEpochPtr_N(0).epoch`.

The next finalized Monad epoch is then treated as already processed because `_crankGlobal()` compares the future-stamped marker against raw `_getEpoch()` and returns success without running the global accounting roll. Repeating this timing at boundaries advances ShMonad's internal epoch counter slower than finalized Monad epochs.

This preserves the same core invariant break as L-08: the global epoch marker can indicate that an epoch was processed even though the corresponding internal/global accounting roll did not execute.

A permissionless caller can extend traditional and isolated withdrawal maturity protocol-wide by repeatedly cranking during delay windows. Queue processing, atomic carry-over, and revenue smoothing are also delayed because they are keyed to skipped internal global rolls.

— RECOMMENDATION

Make global crank readiness and marker storage use the same epoch basis. Prevent `_crankGlobal()` from running while `_inEpochDelayPeriod()` is true. Alternatively, store only the raw finalized epoch marker when deciding global-roll completion, and keep barrier-adjusted timing in separate state that is not used as proof that the global accounting roll executed.

— RESOLUTION

Status: Reported during the fix review period.

L-02 Delay Crank Skips Global Epoch Roll

CONTINUED

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/794>

LOW L-03

Dust Unstake Residual Drops Redemption Queue

CATEGORY	Logical Error
LOCATION	https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/14e40ee4491eb0b7454258b2488f43fbe48e424b/src/shmonad/libraries/StakeAllocationLib.sol#L215
REVIEW	Remediation Review 1

DESCRIPTION

Current-epoch deposits can net almost all of a traditional unstake queue, leaving a non-turnover `queueForUnstake` residual at or below `DUST_THRESHOLD`.

`_offsetLiabilitiesWithDeposits()` moves available deposit MON into `reservedAmount` and reduces both `queueToStake` and `queueForUnstake`. If the remaining redemption-backed unstake demand is exactly dust-sized, validator allocation does not assign it to any validator because `calculateValidatorEpochStakeDelta()` only processes non-turnover withdrawals when `nonTurnoverQueueForUnstake > DUST_THRESHOLD`.

The resulting validator `netAmount` is zero. `_initiateStakeAllocationSkip()` handles `netAmount == 0` by only emitting `LowValidatorStakeDeltaNetZero`; it does not requeue or reserve the aggregate residual. The dust withdrawal demand can therefore age out while `redemptionsPayable` and the user's fixed traditional unstake request still include the full amount.

In liquidity-tight states, a dust-sized `queueForUnstake` residual can be dropped instead of backed, withdrawn, or requeued. This can leave the protocol under-reserved by dust and make a matured `completeUnstake()` revert until later liquidity or atomic capacity covers the gap.

RECOMMENDATION

Preserve aggregate dust residuals when no validator receives a withdrawal delta. Requeue the unallocated `queueForUnstake`, reserve it explicitly, or enforce an invariant that redemption liabilities cannot age out unless backed by reserves, pending unstaking, or available atomic liquidity.

RESOLUTION

Status: Reported during the fix review period.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/796>

LOW **L-04**

Onboarding Isolated Yield Routes Pooled

CATEGORY	Rewards
LOCATION	https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/14e40ee4491eb0b7454258b2488f43fbe48e424b/src/shmonad/StakeTracker.sol#L347
REVIEW	Remediation Review 1

— DESCRIPTION

The productive-staking-yield snapshot fix can misattribute the first staking reward for isolated deposits into a newly added validator.

`_addValidator()` marks the validator's current epoch as `wasCranked = true` while still allowing `depositIsolated()`. If a user deposits isolated stake during that onboarding epoch, the MON is delegated and can become productive, but the first validator crank is skipped and `_setIsolatedSnapshots()` does not initialize `snapshotProductiveIsolatedStake`.

On the next real validator crank, `_settleEarnedStakingYield()` runs before isolated pending stake is synced active. The staking-yield splitter sees `snapshotProductiveIsolatedStake == 0` and routes the first reward to pooled shMON instead of the isolated vShare holder.

Impact is bounded to newly added validators and the first reward window before the first successful snapshot roll. Principal and withdrawals are unaffected, but isolated users can lose a small first-epoch yield slice to pooled holders.

— RECOMMENDATION

Block isolated deposits until a new validator completes its first snapshot roll, or initialize/sync productive isolated snapshots before settling the first staking yield.

— RESOLUTION

Status: Reported during the fix review period.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/795>

LOW **L-05**

Reused Coinbase Keeps Stale Auth

CATEGORY Logical Error

REVIEW Remediation Review 1

— DESCRIPTION

Re-adding a fully removed validator id with `addValidator(uint64)` reuses the same deterministic Coinbase proxy, but does not reset or reinitialize that proxy's storage.

Full validator removal only unlinks the validator id from its Coinbase. The deterministic Coinbase salt is derived from the validator id, so re-adding the same id returns the old proxy. Because the proxy is already deployed, `initialize()` is skipped and the new lifecycle inherits the previous cached `authAddress`, commission recipient, commission rates, donation rate, and unpaid balances.

Until `updateAuthAddress()` is called, the previous cached auth remains authorized for Coinbase config and `processCoinbaseByAuth(uint64)`. New lifecycle priority-fee or MEV settlement can therefore use stale settings or be routed to the previous recipient. Even when auth is refreshed, stale rates and unpaid buffers remain unless separately corrected.

— RECOMMENDATION

Reset Coinbase config and unpaid balances when reusing an already deployed deterministic proxy, or include a lifecycle nonce in the CREATE2 salt. At minimum, refresh `authAddress` from the staking precompile during re-add and clear unpaid balances.

— RESOLUTION

Status: Reported during the fix review period.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/797>

INFORMATIONAL I-01

Gas-Dependent Coinbase Donation Attribution

CATEGORY Logical Error

REVIEW Remediation Review 1

DESCRIPTION

`_settleCoinbaseContract()` classifies all MON received during `Coinbase.process()` as revenue for the currently cranked validator using a raw ShMonad balance delta.

A nested callback can theoretically call another Coinbase's `donateUnpaidRewardsToShMonad()` during this window, causing that donation to be included in the current validator's reward delta instead of being queued as goodwill.

Under current constants, the standard production path appears gas-blocked: commission recipients receive only `50,000` gas, while `donateUnpaidRewardsToShMonad()` calls `_canReceiveExternalReward()`, which budgets `107,200` gas for `getValidator()`. If Monad later reduces this precompile cost or changes under-gassed failure behavior, the accounting boundary could become reachable.

RECOMMENDATION

Do not rely on callback gas limits for reward attribution safety. Bind the transient classifier to the expected coinbase sender, or account only explicit funds received from the coinbase being processed.

RESOLUTION

Status: Reported during the fix review period.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/798>

HIGH **H-01**

Pooled unstake accounting can freeze protocol

CATEGORY	DoS
LOCATION	src/shmonad/StakeTracker.sol:1600
REVIEW	Remediation Review 2

— DESCRIPTION

When ShMonad initiates a pooled stake decrease, `_initiateStakeAllocationDecrease()` calls `_initiateWithdrawal(valId, netAmount, withdrawalId)` and then records only `netAmount` as pending unstake. This assumes the native staking precompile withdraws exactly the requested amount.

However, the real Monad precompile does not always withdraw exactly the requested amount. In `handle_undelegate`, if the requested undelegation would leave a nonzero active-stake remainder below `DUST_THRESHOLD (1 gwei)`, Monad rounds the undelegation up to the delegator's full active stake:

```
``plain text
let remaining = checked_sub_u256(del.stake, amount)?;
if !remaining.is_zero() && remaining < DUST_THRESHOLD {
amount = del.stake; // collect all including dust
}
``
```

This creates an accounting mismatch in pooled unstake flows. Example:

- Validator active pooled stake is 100 MON.
- Validator also has 1 MON pending deposit.
- ShMonad computes a pooled withdrawal of 100 MON - (DUST_THRESHOLD - 1).
- The post-withdrawal target remains above MIN_VALIDATOR_DEPOSIT because of the pending deposit, so the forced full-exit path does not trigger.
- Monad rounds the undelegation up and withdraws the full 100 MON active stake.
- ShMonad records only 100 MON - (DUST_THRESHOLD - 1) as pending unstake and keeps the dust remainder in the validator target.

On the next balance verification, ShMonad expects the dust remainder to still exist as principal, but the native precompile has already moved it into the withdrawal request. This can produce a principal deficit and freeze/fail-stop the protocol even though the mismatch is only dust-sized.

H-01 Pooled unstake accounting can freeze protocolCONTINUED

— RECOMMENDATION

Before initiating a pooled withdrawal, mirror Monad's dust-rounding rule in ShMonad accounting. If `activeStake - netAmount` is nonzero and below `DUST_THRESHOLD`, increase `netAmount` to the full active stake and reduce `nextTargetStakeAmount` by the same additional amount before recording pending unstake.

This should be applied to pooled stake decreases similarly to the existing isolated-stake dust handling.

— RESOLUTION

Status: Reported during the fix review period.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/808>

MEDIUM **M-01**

Queue netting creates synthetic goodwill

CATEGORY Math
LOCATION `src/shmonad/StakeTracker.sol:1321`
REVIEW Remediation Review 2

DESCRIPTION

During global crank, `_offsetExcessQueueCapacityWithNet()` computes `_globalStakableAmount` from live pooled current assets and passes both that value and `queueToStake` into `QueueLib.planExcessQueueOffset()`.

When the queue planner applies capacity-only netting, it reduces both `queueForUnstake` and `queueToStake`. However, the liquid MON corresponding to the reduced `queueToStake` is not moved into reserves, atomic liquidity, revenue, or a future stake queue. The native MON remains in the contract as pooled current assets, but it is no longer represented by any explicit queue or accounting destination. Afterward, `AccountingLib.goodwill()` computes goodwill as pooled current assets minus current `queueToStake`, so the unqueued MON is classified as goodwill even though it did not come from an external donation or other expected goodwill source.

This is not a direct solvency break because the MON remains in pooled equity, but it weakens the bucket model and can leave meaningful capital idle/unpurposed until a later crank reclassifies it as unexpected goodwill.

RECOMMENDATION

Do not clear `queueToStake` through capacity-only netting unless the corresponding liquid MON is explicitly moved into a tracked destination.

Possible fixes:

- Route the matched liquid MON into reserves when it backs redemption demand.
- Carry the matched amount forward in `queueToStake` if it is still intended for staking.

PROOF OF CONCEPT <https://github.com/GuardianOrg/fastlane-contracts-internal-fuzz-1777384878281/blob/fuzz-suite-06-12/test/fuzzing/FoundryBreakDebugging.sol#L75>

RESOLUTION

Status: Reported during the fix review period.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/812>

MEDIUM **M-02**

Atomic rebalance can reuse reserved revenue

CATEGORY Math
LOCATION `src/shmonad/StakeTracker.sol:2919`
REVIEW Remediation Review 2

DESCRIPTION

During `crank()`, `_settleGlobalNetMONAgainstAtomicUnstaking()` first computes adjusted atomic utilization using current unallocated revenue via `_getAdjustedAmountsForAtomicUnstaking()`. Later in the same function, freed atomic liquidity can be routed into `s_globalCapital.reservedAmount` to cover outstanding liabilities.

The reserve-fill branch increases `reservedAmount` and reduces `_netStakeIn`, but does not mark any current-epoch revenue as consumed with `_allocateCurrentEpochRevenue()`.

As a result, the same current-epoch revenue can be treated as available for two effects in the same crank flow:

1. Reducing adjusted atomic utilization / improving atomic liquidity.
2. Backing redemption reserves.

A fuzz replay demonstrates this mismatch (POC Attached)

```
``vb.net
reserveIncrease = 15,766,000,863 wei
atomicDistributedDecrease = 45,774,728,229 wei
allocatedRevenueIncrease = 11,154,531,365 wei
``
```

The accounting remains balanced at a high level, but source tagging is wrong: revenue that should be consumed by reserve coverage can also make atomic utilization look healthier. This can overstate liquidity and redemption-backing health during crank.

RECOMMENDATION

Do not let unallocated current-epoch revenue support both reserve fills and atomic utilization adjustment. Either mark reserve-filled revenue as consumed:

```
``plain text
_allocateCurrentEpochRevenue(_toReserves);
s_globalCapital.reservedAmount += _toReserves;
```

Or, more robustly, split `_netStakeIn` by source and allocate only the revenue-backed

M-02 Atomic rebalance can reuse reserved revenue

CONTINUED

reserve portion. The atomic rebalance math should use revenue net of any amount needed for reserve coverage.

PROOF OF CONCEPT <https://github.com/GuardianOrg/fastlane-contracts-internal-fuzz-1777384878281/blob/fuzz-suite-06-12/test/fuzzing/FoundryBreakDebugging.sol#L45>

— RESOLUTION

Status: Reported during the fix review period.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/811>

MEDIUM **M-03**

Inactive validator can affect allocation scoring

CATEGORY Math

LOCATION `src/shmonad/libraries/StakeAllocationLib.sol:142`

REVIEW Remediation Review 2

DESCRIPTION

The adjusted staking-yield deadband computes each validator's expected-yield floor from global staking yield:

```
``plain text
globalStakingYield * validatorSignalStake / totalAllocationSignalStake
`
```

However, `_recomputeCurrentAllocationTotals()` only includes active validators in `totalAllocationSignalStake` and `totalAdjustedWeightedRevenue`.

At the same time, `_handleEarnedStakingYield()` records staking yield into both the validator allocation bucket and `globalAllocationRevenue.stakingYield` without checking whether the validator is still active.

This means yield from inactive or deactivating validators can remain in the global staking-yield numerator, while those same validators are excluded from the active-set denominator. That creates an inconsistent deadband calculation: active validators are charged a deadband floor based on yield earned by validators that are no longer eligible for allocation.

Example

- Active validator A earns 5 MON staking yield.
- Active validator B earns 5 MON staking yield.
- Inactive validator C earns or settles 90 MON staking yield.
- `globalStakingYield` = 100 MON.
- `totalAllocationSignalStake` only includes A and B.

If A and B have equal signal stake, each active validator may be assigned an expected-yield floor around 50 MON, despite only earning 5 MON.

Their adjusted staking yield can be saturated to zero. As a result,

`totalAdjustedWeightedRevenue` can become zero or materially understated, even though active validators produced valid yield.

Impact

This can suppress stake allocation, roll stake queues forward, or distort turnover

M-03 Inactive validator can affect allocation scoring

CONTINUED

routing.

The issue is economic rather than a direct fund drain: capital may remain idle or be allocated inefficiently because inactive validator yield contaminates the scoring baseline for active validators.

— RECOMMENDATION

Consider not adding staking yield from inactive/deactivating validators to

`globalAllocationRevenue.stakingYield` used for active allocation scoring or add it to both numerator and denominator.

— RESOLUTION

Status: Reported during the fix review period.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/810>

MEDIUM **M-04**

Deadband-adjusted revenue can block allocations

CATEGORY Math

LOCATION `src/shmonad/StakeTracker.sol:1367`

REVIEW Remediation Review 2

DESCRIPTION

`_rollUnassignableStakingQueue()` rolls the full `queueToStake` forward when:

```
`plain text`
```

```
totalAdjustedWeightedRevenue < MIN_VALIDATOR_DEPOSIT
```

```
`
```

This check may have been reasonable before the new deadband-adjusted scoring, when the allocation denominator was closer to raw validator revenue or yield.

Under the new logic, however, `totalAdjustedWeightedRevenue` is no longer raw revenue. It is the sum of each active validator's adjusted score:

```
`plain text`
```

```
max(actualStakingYield - deadbandFloor, 0)
```

```
+ nonStakingRevenue * multiplier
```

```
`
```

With the default 80% staking-yield deadband, ordinary stake-proportional yield is intentionally filtered out. As a result, even large validators with meaningful raw staking yield can produce less than 1 MON of adjusted revenue signal.

Example

Assume:

- `queueToStake` = 1,000 MON

- Validator A adjusted revenue = 0.3 MON

- Validator B adjusted revenue = 0.2 MON

- `totalAdjustedWeightedRevenue` = 0.5 MON

- `MIN_VALIDATOR_DEPOSIT` = 1 MON

Current behavior:

```
`plain text`
```

```
0.5 MON < 1 MON
```

```
`
```

Therefore:

- The full 1,000 MON `queue` rolls to the next epoch.

M-04 Deadband-adjusted revenue can block allocations

CONTINUED

- Validator A receives 0 MON new stake.
- Validator B receives 0 MON new stake.

However, the allocation denominator is valid, and proportional allocation would work:

```
`` vb.net
A receives 1,000 * 0.3 / 0.5 = 600 MON
B receives 1,000 * 0.2 / 0.5 = 400 MON
`[
```

So the issue is not that the stake queue is actually unassignable. The issue is that a legacy-style MIN_VALIDATOR_DEPOSIT threshold is being applied to a new deadband-adjusted revenue denominator.

****Impact****

This can leave pooled MON idle for one or more epochs, reducing capital efficiency and delaying stake allocation to validators with small but valid adjusted revenue signals.

— RECOMMENDATION

Consider not comparing deadband-adjusted revenue against MIN_VALIDATOR_DEPOSIT. Replace the guard with a zero or dust-level denominator check, for example:

```
`` plain text
if (s_activeValidatorCount > 0 && _weightedRevenue == 0) {
...
}
```

Or, to align with existing per-validator dust behavior:

```
` plain text
if (s_activeValidatorCount > 0 && _weightedRevenue <= DUST_THRESHOLD) {
...
}
```

— RESOLUTION

Status: Reported during the fix review period.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/809>

LOW **L-01**

JIT Deposit Captures Orphan Sweep

CATEGORY	Logical Error
LOCATION	https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/43c356d35f00032c70a84abf472af5d7e8a77dff/src/shmonad/ShMonad.sol#L482
REVIEW	Remediation Review 2

— DESCRIPTION

Permissionless isolated finalization can make orphaned isolated value immediately claimable by a JIT pooled depositor.

A full isolated exit burns the user's vShares immediately, but leaves a fixed pending redemption. While that redemption is pending, isolated-attributed rewards or surplus can still accumulate as isolated idle assets. Once all vShares are gone, this creates residual isolated value with no remaining isolated share claimant. When anyone later calls `completeUnstakeIsolatedFor()`, ShMonad pays the fixed redemption amount and then sweeps the leftover isolated idle assets out of isolated accounting. Because pooled equity is total protocol equity minus isolated total assets, the sweep immediately increases the pooled shMON exchange rate.

An attacker can deposit into pooled shMON immediately before calling the permissionless finalizer, then redeem or request unstake after the sweep. This lets newly minted pooled shares capture part of orphaned isolated reward/surplus that accrued before the attacker held shMON.

— RECOMMENDATION

Do not make orphan-swept isolated value immediately claimable by newly minted pooled shares. Route swept value through recent-revenue smoothing, credit it using a pre-sweep pooled-supply accumulator, or otherwise exclude same-block/newly minted pooled shares from the sweep windfall.

— RESOLUTION

Status: Reported during the fix review period.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/801>

LOW **L-02**

Inactive Reward Redirect Skips Smoothing

CATEGORY	Logical Error
LOCATION	https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/43c356d35f00032c70a84abf472af5d7e8a77dff/src/shmonad/StakeTracker.sol#L2776
REVIEW	Remediation Review 2

— DESCRIPTION

The prior inactive/unknown validator reward smoothing fix updated `_handleValidatorRewards()` so pooled yield from inactive, unregistered, or `UNKNOWN_VAL_ID` validators is recorded in `globalRevenuePtr_N(0).earnedRevenue`. That makes ERC4626 recent-revenue smoothing apply before short-lived depositors can exit against the fresh yield.

However, the fix did not cover the prior-epoch `rewardsPayable` redirect path.

When `_settleValidatorRewardsPayable()` finds prior-epoch `rewardsPayable` for a validator that has since become inactive, it calls `_handleRewardsRedirect()`. That function consumes the validator-local payable bucket, reduces global `reservedAmount` and `rewardsPayable`, and adds the amount to `globalCashFlowsPtr_N(0).queueToStake`.

It does not increment `globalRevenuePtr_N(0).earnedRevenue`.

As a result, redirected inactive-validator reward escrow becomes pooled shMON yield without entering `_recentRevenueOffset()`. A JIT depositor can deposit before the inactive-validator redirect is cranked, then redeem or request unstake after the redirect and immediately share in reward value that was earned before they held shMON.

This is the same economic class as the previously fixed inactive/unknown `_handleValidatorRewards()` branch, but through a missed legacy escrow path.

Impact: Freshly released inactive-validator reward value can bypass smoothing and become immediately claimable by short-lived pooled depositors, diluting standing pooled shMON holders by the attacker's pro-rata share of the redirected amount.

— RECOMMENDATION

In `_handleRewardsRedirect()`, record redirected pooled reward escrow in `globalRevenuePtr_N(0).earnedRevenue` before it is included in user-facing ERC4626 exit pricing. If validator-level allocation attribution should remain excluded from active-validator routing weights, keep that distinction, but still classify the pooled shMON yield as recent global revenue for smoothing.

L-02 Inactive Reward Redirect Skips Smoothing

CONTINUED

— RESOLUTION

Status: Reported during the fix review period.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/800>

LOW **L-03**

MEV Commission Reprices Escrow

CATEGORY	Logical Error
LOCATION	https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/43c356d35f00032c70a84abf472af5d7e8a77dff/src/shmonad/StakeTracker.sol#L2776 (https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/43c356d35f00032c70a84abf472af5d7e8a77dff/src/shmonad/CoinbaseUpgradeable.sol#L225)
REVIEW	Remediation Review 2

DESCRIPTION

ShMonad records active-validator MEV as an unsplit `rewardsPayable` escrow, then applies the validator Coinbase's live `mevCommissionRate` only when that escrow is settled in a later crank.

A validator auth can keep a low MEV commission while rewards accrue through `sendValidatorRewards()`.

ShMonad stores the validator payout in `validatorRewardsPtr_N(0, valId).rewardsPayable`, increases global `rewardsPayable`, and reserves the MON. The applicable MEV commission rate is not snapshotted.

On the next validator crank, `_settleValidatorRewardsPayable()` forwards the prior-epoch escrow to the validator Coinbase through `handleMEVPayable()`. That function reads the current

`s_config.mevCommissionRate` and splits the already-accrued escrow at settlement time.

The validator auth can therefore raise `mevCommissionRate` immediately before settlement and apply the new rate to MEV that accrued while the advertised rate was lower. This can redirect part or all of the pending delegate reward escrow to the commission recipient.

RECOMMENDATION

Snapshot the MEV commission rate when `rewardsPayable` is created, or split the ShMonad escrow into commission and delegate-reward buckets at accrual time. Alternatively, force-settle existing

`rewardsPayable` before allowing commission changes, or make commission updates effective only for future epochs.

RESOLUTION

Status: Reported during the fix review period.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/806>

LOW **L-04**

Unpaid Rewards Skip Smoothing

CATEGORY	MEV
LOCATION	src/shmonad/CoinbaseUpgradeable.sol
REVIEW	Remediation Review 2

— DESCRIPTION

CoinbaseUpgradeable can hold unpaid delegate rewards after ShMonad has already cleared its rewardsPayable liability. This happens when MEV is sent to the validator coinbase through handleMEVPayable(), but the coinbase later cannot forward the delegate reward portion through the staking precompile, for example because the validator has left the rewardable set or the reward amount remains in the unpaid buffer.

donateUnpaidRewardsToShMonad() intentionally lets anyone recover that buffer once the validator is no longer rewardable. The function zeroes s_unpaid.rewards and transfers the MON directly to ShMonad. ShMonad receives that transfer through the generic goodwill path. The receive() function increments globalCashFlows.queueToStake, but does not increment globalRevenue.earnedRevenue.

ERC4626 exits and traditional unstake requests rely on _recentRevenueOffset() to prevent short-lived depositors from immediately capturing newly realized pooled value. That offset only includes globalRevenue.earnedRevenue, so recovered unpaid coinbase rewards become immediately visible in pooled exit pricing instead of being smoothed.

A short-lived depositor can deposit before calling the permissionless donation function, trigger donateUnpaidRewardsToShMonad(), then redeem or request unstake against the inflated post-donation share rate. The recovered MON was earned before the depositor held shMON, but is distributed over the current share supply.

Impact: recovered unpaid delegate rewards can be redistributed to short-lived pooled depositors instead of being smoothed to standing shMON holders. The issue occurs on the intended unpaid-reward recovery path after a validator leaves the rewardable set, so the value is already protocol-recognized yield being returned to ShMonad, but it is not classified as recent global revenue before affecting exit pricing.

— RECOMMENDATION

L-04 Unpaid Rewards Skip Smoothing

CONTINUED

Route unpaid coinbase reward donations through a ShMonad accounting hook that records the recovered pooled reward amount in `globalRevenue.earnedRevenue` before it affects exit pricing. Keep boost commission and validator allocation attribution excluded if desired, but classify the recovered delegate reward value as recent pooled revenue for smoothing.

— RESOLUTION

Status: Reported during the fix review period.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/814>

LOW **L-05**

Share-Burn Boosts Skip Smoothing

CATEGORY	MEV
LOCATION	src/shmonad/StakeTracker.sol
REVIEW	Remediation Review 2

— DESCRIPTION

boostYield(uint256 shares, address from, address yieldOriginator) burns shMON and leaves the corresponding pooled value in the vault. This reduces total supply while equity remains in the system, increasing the share price for remaining holders without adding new MON.

ERC4626 exits and traditional unstake requests rely on `_recentRevenueOffset()` to prevent short-lived depositors from immediately capturing newly-realized value. That offset only includes `globalRevenuePtr_N(0).earnedRevenue`.

In `_handleBoostYieldFromShares()`, the active-validator branch records the clamped burn value in `globalRevenuePtr_N(0).earnedRevenue`. However, the inactive / placeholder branch only credits `validatorRewardsPtr_N(0, UNKNOWN_VAL_ID).earnedRevenue` and `validatorAllocationRevenuePtr_N(0, UNKNOWN_VAL_ID).nonStakingRevenue`. It does not increment `globalRevenuePtr_N(0).earnedRevenue`. The MON-backed boost and inactive-reward paths already separate validator attribution from global smoothing: inactive or unknown rewards are kept out of active validator allocation weights, but pooled value is still recorded globally so immediate exits cannot capture it.

As a result, if the current proposer resolves to `UNKNOWN_VAL_ID`, inactive, deactivated, or otherwise not an active ShMonad validator, a share-burn boost immediately improves pooled exit pricing without entering `_recentRevenueOffset()`.

A short-lived depositor can deposit before a visible share-burn boost, wait for another account's approved/self-owned shares to be burned, then redeem or request unstake against the unsmoothed post-burn rate. Since pooled equity is unchanged while total supply decreases, the depositor captures part of the value donated by the burned shares.

The same class also exists in the active branch when burn value exceeds atomic headroom: only the clamped amount enters `globalRevenue`, while the residual burn value still improves the share rate immediately.

— RECOMMENDATION

Separate recent-revenue smoothing from validator attribution and atomic-liquidity accounting. Record the full net share-burn value in a smoothing-only offset for all proposer

L-05 Share-Burn Boosts Skip Smoothing

CONTINUED

branches, while keeping placeholder attribution out of active allocation weights. Only the liquidity-backed portion should affect atomic distributedAmount, available-revenue settlement, and carryover accounting.

— RESOLUTION

Status: Reported during the fix review period.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/813>

INFORMATIONAL I-01

Forced Finalizers Forward All Gas

CATEGORY	Gas Griefing
LOCATION	https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/43c356d35f00032c70a84abf472af5d7e8a77dff/src/shmonad/StakeTracker.sol#L3353
REVIEW	Remediation Review 2

— DESCRIPTION

Permissionless isolated finalizers can forward nearly all remaining gas to an untrusted withdrawal receiver before Solady falls back to force-send.

`completeUnstakeIsolatedFor()` and `claimDeactivatedIsolatedFor()` intentionally allow third parties to settle abandoned isolated positions. Both paths call `_sendIsolatedSettlement(..., true)`, which pays the original owner with `receiver.forceSafeTransferETH(amountMon, gasleft())`.

Solady's force-send variant first calls the receiver with the supplied gas stipend. Because ShMonad supplies `gasleft()`, a malicious contract receiver can accept the MON while consuming most of the caller's gas before returning.

This does not steal funds or permanently block settlement, because a sufficiently high-gas caller can still complete cleanup and Solady can force-send on failure. However, it makes third-party cleanup of malicious or abandoned isolated positions unnecessarily expensive, and can cause cleanup attempts to revert if too little gas remains for post-transfer execution.

— RECOMMENDATION

Use Solady's default `forceSafeTransferETH(amountMon)` or another bounded stipend large enough for ordinary receiver hooks, while preserving the force-send fallback. Do not replace this with a plain bounded `call`, because that could reintroduce receiver-controlled blocking.

— RESOLUTION

Status: Reported during the fix review period.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/803>

INFORMATIONAL I-02

Late Cranks Extend Revenue Smoothing

CATEGORY	Informational
LOCATION	https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/43c356d35f00032c70a84abf472af5d7e8a77dff/src/shmonad/FLERC4626.sol#L423
REVIEW	Remediation Review 2

DESCRIPTION

L-08 fixed validator cranks incorrectly marking global epochs as processed, but ERC4626 revenue smoothing remains freshness-blind when a valid global roll is simply overdue.

`_recentRevenueOffset()` always excludes `globalRevenuePtr_N(0).earnedRevenue` in full until `_crankGlobal()` rotates the revenue bucket. If the Monad epoch has advanced and a global roll is available, but no one calls `crank()`, already-aged current revenue remains treated as fresh for ERC4626 exit pricing.

When the late global crank finally runs, `_updateRevenueSmootherAfterEpochChange()` records `epochChangeBlockNumber = block.number`, so the decay starts from the late maintenance block rather than the actual epoch boundary or revenue age. This can extend smoothing by the stale-crank delay plus a fresh smoothing window.

The value is not lost and anyone can resolve the stale state by calling `crank()`. The risk is stale exit pricing for users or integrations that redeem, withdraw, or request unstake without first ensuring the global roll is fresh.

RECOMMENDATION

Document that ERC4626 exit quotes are only freshness-correct after pending global rolls are processed, and recommend checking `isGlobalCrankAvailable()` or calling `crank()` before relying on exit previews around epoch boundaries. If stronger behavior is desired, anchor smoothing decay to the actual epoch boundary or revenue age instead of the late crank block.

RESOLUTION

Status: Reported during the fix review period.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/804>

INFORMATIONAL I-03

Placeholder Event Reads New Epoch

CATEGORY Events

LOCATION <https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/43c356d35f00032c70a84abf472af5d7e8a77dff/src/shmonad/StakeTracker.sol#L1630>

REVIEW Remediation Review 2

— DESCRIPTION

The placeholder validator crank can emit `UnregisteredValidatorRevenue` with the completed epoch label but the new epoch's reward bucket.

After `_crankGlobal()` advances `s_admin.internalEpoch`, UNKNOWN validator revenue from the just-ended epoch shifts from `N(0)` to `N(-1)`. `_crankPlaceholderValidator()` correctly labels the event with `globalEpochPtr_N(-1).epoch`, but reads `validatorRewardsPtr_N(0, UNKNOWN_VAL_ID)` for `rewardsPayable` and `earnedRevenue`.

As a result, the event can report zero or unrelated current-epoch UNKNOWN revenue for the completed epoch. On-chain accounting is unaffected because the revenue is already booked when received and the event values are not consumed by settlement logic. Impact is limited to indexers and monitoring that rely on this event.

— RECOMMENDATION

In `_crankPlaceholderValidator()`, read `validatorRewardsPtr_N(-1, UNKNOWN_VAL_ID)` when emitting the completed epoch's `UnregisteredValidatorRevenue`, or cache the completed bucket before advancing the epoch pointer.

— RESOLUTION

Status: Reported during the fix review period.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/805>

MEDIUM **M-01**

RESOLVED

Onboarding Rewards Are Orphaned

CATEGORY	Logical Error
LOCATION	https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/9f116de99a16a35f1889e08880e0b50141d28701/src/shmonad/ValidatorRegistry.sol#L306
REVIEW	Main Review 2

DESCRIPTION

A newly added validator can receive validator rewards during its onboarding epoch, but those rewards can be skipped permanently by the crank logic.

`_addValidator()` initializes the validator's current epoch bucket with `wasCranked = true`. The `sendValidatorRewards()` entrypoint remains open during that epoch. If the validator is active and in ShMonad's active-set view, `_handleValidatorRewards()` treats the payment as a validator payout: it writes the net amount to `validatorRewardsPtr_N(0, valId).rewardsPayable` and increases both `s_globalLiabilities.rewardsPayable` and `s_globalCapital.reservedAmount`.

After the next global epoch roll, that current bucket becomes `N(-1)`, the bucket

`_settleValidatorRewardsPayable()` is supposed to settle. However, `_crankValidator()` first loads `validatorEpochPtr_N(-1, valId)` and returns immediately when `wasCranked` is already true. Because the onboarding bucket was pre-marked as cranked, the validator crank exits before calling `_settleValidatorRewardsPayable()`.

The local reward bucket then ages through the ring buffer without being paid, redirected, or requeued. Later rolls can clear the local bucket, while the global `rewardsPayable` liability and matching `reservedAmount` remain inflated. This strands submitted validator rewards and leaves persistent global accounting pollution until an upgrade.

RECOMMENDATION

Prevent reward writes into pre-cranked onboarding buckets, or initialize new validators so any current-epoch reward bucket that can receive rewards is also crankable. Alternatively, route onboarding-epoch validator rewards to the next settleable bucket or to pooled yield until the first validator crank completes.

RESOLUTION

Status: Resolved.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/816>

MEDIUM **M-02**

PARTIALLY RESOLVED

Late Cranks Mis-Split Rewards

CATEGORY	Logical Error
LOCATION	https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/9f116de99a16a35f1889e08880e0b50141d28701/src/shmonad/StakeTracker.sol#L1509
REVIEW	Main Review 2

— DESCRIPTION

Late validator cranks can split staking rewards using stale ShMonad pending-deposit state after the native delegation has already become productive.

During `_crankValidator()`, ShMonad calls `_settleEarnedStakingYield()` before `_settlePastEpochEdges()`. `_settleEarnedStakingYield()` claims rewards from the Monad staking precompile, then `_handleEarnedStakingYield()` splits them between pooled shMON and isolated vShares through `_splitStakingYieldByProductiveSnapshot()`.

The pooled side of that split is derived from `_productivePooledStakeSnapshot()`, which calls `getValidatorAmountAvailableToUnstakeSnapshot()`. If `validatorEpochPtr_N(-1, valld).hasDeposit` is still true, that helper subtracts `validatorPendingPtr_N(-1, valld).pendingStaking` from the validator target stake. That is correct while the deposit is genuinely pending, but not if the validator crank is late enough for the native epoch to advance first. In that case the staking precompile can already include the pooled delegation in productive stake and pay rewards on it while ShMonad has not yet cleared its old pending flag.

ShMonad clears the stale pending deposit in `_settlePastEpochEdges()`, but only after the reward claim and split. The reward is therefore split with an understated pooled denominator. If isolated stake exists on the same validator, part of the reward earned by pooled stake is credited to isolated accounting.

Example: a validator has 200 MON productive pooled stake and 100 MON productive isolated stake in the precompile, but ShMonad still has an old 100 MON pooled deposit marked pending in `N(-1)` because the validator crank is late. A 30 MON reward should split 20 MON pooled / 10 MON isolated. Current logic subtracts the stale pending deposit and uses 100 MON pooled versus 100 MON isolated, crediting 15 MON pooled / 15 MON isolated. The extra 5 MON is transferred from pooled shMON holders to isolated vShare holders.

— RECOMMENDATION

Settle or refresh matured pending deposit edges before claiming and splitting staking yield in late-crank/skipped-native-epoch cases. Alternatively, compute staking-yield attribution from a native productive-stake basis that reflects deposits already earning rewards.

M-02 Late Cranks Mis-Split Rewards

CONTINUED

— RESOLUTION

Status: Partially resolved.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/817>

MEDIUM **M-03**

RESOLVED

Pooled Surplus Credits Isolated Holders

CATEGORY	Logical Error
LOCATION	https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/9f116de99a16a35f1889e08880e0b50141d28701/src/shmonad/StakeTracker.sol#L2709
REVIEW	Main Review 2

— DESCRIPTION

Pooled withdrawal surplus is routed through the generic staking-yield splitter, allowing yield from a pooled withdrawal position to be credited to isolated vShare holders.

When ShMonad decreases pooled stake allocation for a validator, the withdrawal is recorded in the pooled pending bucket as `validatorPendingPtr.pendingUnstaking`. If the Monad staking precompile later returns more than the expected pooled principal, `_handleCompleteDecreasedAllocation()` treats the excess as staking yield.

However, the surplus is passed to `_handleEarnedStakingYield()`. That function is not source-aware and splits the amount between pooled and isolated accounting through `_splitStakingYieldByProductiveSnapshot()`. As a result, surplus generated by a pooled withdrawal can be partially, or even fully, credited to isolated accounting.

Example: validator V has a completed pooled withdrawal for 100 MON and active isolated stake. If the native withdrawal settles for 112 MON, the 12 MON surplus came from the pooled withdrawal position. If V's pooled target is now zero while isolated stake remains, the splitter sees zero pooled productive stake and nonzero isolated stake, so the full 12 MON is credited to `s_isolatedTotalAssets` and `queuedRestake` instead of `pooled earnedRevenue`.

This breaks the pooled/isolated accounting boundary and transfers yield from pooled shMON holders to isolated vShare holders.

— RECOMMENDATION

Handle pooled withdrawal surplus with source-aware accounting. Charge staking commission as usual, but credit the net surplus only to pooled `earnedRevenue` and pooled staking-yield allocation buckets. Do not call `_handleEarnedStakingYield()` for source-known pooled withdrawal surplus unless it accepts an explicit `source/domain` parameter.

— RESOLUTION

Status: Resolved.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/818>

MEDIUM **M-04**

RESOLVED

Readded validator books old stake twice

CATEGORY	Validation
LOCATION	completeDeactivatingValidator()
REVIEW	Main Review 2

— DESCRIPTION

This needs a validator removal where pooled stake is still left behind, then the owner re adds that same validator ID. After that, anyone can crank the system and make the bad accounting happen the vault tracks how much MON it has staked in `s_globalCapital.stakedAmount`. That number is trusted by `totalAssets()` when pricing shares

The wrong part is validator deletion. `_completeDeactivatingValidator()` deletes the validator local records after the timeout, but it does not first prove that the validator has zero pooled stake, zero pending pooled deposits, zero pending pooled withdrawals, or zero actual stake in the Monad staking precompile It deletes `s_validatorEpoch`, `s_validatorPending`, rewards, and allocation records, but does not reduce global staked accounting

There is a mitigation for isolated stake, but not for pooled stake. The override only blocks deletion while isolated crank work remains

The reuse check also only blocks residual isolated state, not residual pooled stake or live precompile stake when the same validator is added again, `_addValidator()` initializes its target stake to zero

If the staking precompile still has old stake for that validator, the next balance check sees

```
``plain text
Expected stake: 0
Actual stake: P
``
```

The code treats that difference as a new windfall and adds it to `s_globalCapital.stakedAmount`. But that MON was not new. It was already counted before deletion. So the same stake can be counted twice If validator unstaking fully succeeds before deletion, the issue does not happen. The dangerous path is failed or incomplete pooled unwind, then validator deletion, then owner re-registration, then permissionless crank

M-04 Readded validator books old stake twice

CONTINUED

— RECOMMENDATION

Add `_hasResidualPooledState(valId)` (true when `_getStakeInfo` is nonzero or any tracked epoch has nonzero `targetStakeAmount`, `hasDeposit`, or `hasWithdrawal`)
and return from `StakeTracker._completeDeactivatingValidator()` before `super while true`, so records are deleted only after the pooled position is fully unwound.

— RESOLUTION

Status: Resolved.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/820>

MEDIUM **M-05**

RESOLVED

Atomic allocation shrink reprices withdrawals

CATEGORY	Logical Error
LOCATION	beforeCompleteUnstake()
REVIEW	Main Review 2

— DESCRIPTION

ShMonad has two ways to exit

1) Instant exit: `withdraw()` / `redeem()`

User gets MON immediately, but pays a dynamic fee

2) Traditional exit: `requestUnstake()` then later `completeUnstake()`

User waits several epochs, then gets MON with no fee

The problem is that traditional completion can secretly use the instant-exit pool

When `completeUnstake()` runs, it calls `_beforeCompleteUnstake()`

If normal redemption reserves are not enough, the code pulls the missing amount from atomic liquidity

The dangerous line is that, It lowers `s_atomicAssets.allocatedAmount`, but does not lower `distributedAmount`

`allocatedAmount` = size of the instant-withdraw pool

`distributedAmount` = how much of that pool is already used

fee depends on: "how used is this pool?"

Atomic fee pricing uses available liquidity and allocation

And the docs describe the fee as:

$\text{utilization} = 1 - \text{available} / \text{allocated}$

So if `allocatedAmount` becomes smaller while `distributedAmount` stays the same, the pool looks more used.

More used pool means higher fee, assume

Before attacker

```
`` typescript
```

```
Pool size: 1000 MON
```

```
Already used: 500 MON
```

```
Utilization: 50%
```

```
Attacker calls completeUnstake() and takes 400 MON from atomic liquidity.
```

```
``
```

M-05 Atomic allocation shrink reprices withdrawals

CONTINUED

After attacker

```
`` typescript
```

```
Pool size: 600 MON
```

```
Already used: still 500 MON
```

```
Utilization: about 83%
```

```
``
```

Now a victim's instant withdrawal pays a much higher fee

Traditional unstaking is supposed to be delayed and fee free. Atomic unstaking is supposed to be instant and fee paying. The bug lets a traditional unstake consume atomic liquidity without paying the atomic fee, while still worsening the price for the next atomic user

So a victim using `withdraw/assets` can burn more shMON than expected, and a victim using `redeem(shares)` can receive less MON than expected

— RECOMMENDATION

Consider hard separating the exit paths, In `_beforeCompleteUnstake()`, replace the atomic liquidity fallback and with

```
require(reservedForRedemptions >= amount, InsufficientReservedLiquidity(amount, reservedForRedemptions));
```

leaving the existing reserve / liability decrements below unchanged

— RESOLUTION

Status: Resolved.

MEDIUM **M-06**

RESOLVED

Pooled Pending Masks Isolated Stake

CATEGORY Rewards

LOCATION <https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/9f116de99a16a35f1889e08880e0b50141d28701/src/shmonad/StakeTracker.sol#L522>

REVIEW Main Review 2

— DESCRIPTION

ShMonad can keep native-active isolated stake marked as pending when a later pooled deposit remains pending in the Monad staking precompile. Rewards earned by that active isolated stake are consequently routed to pooled shMON holders.

`_syncIsolatedPendingStake()` promotes isolated `pendingStake` only when aggregate native pending deposits are lower than the recorded isolated amount. This is conservative because Monad reports pending deposits at the delegator level without distinguishing pooled and isolated sources.

The issue occurs when deposits activate in different native epochs. An isolated deposit `I` is scheduled first for `E+1`. Before ShMonad synchronizes it, a pooled allocation increase `P` is scheduled during the boundary period for `E+2`. At `E+1`, `I` is native-active while `P` remains pending.

During the `E+1` validator crank, `_settleEarnedStakingYield()` runs before balance verification. Because ShMonad still has `snapshotProductiveIsolatedStake == 0`, any reward already earned by the native-active isolated stake is assigned to pooled shMON. Balance verification then calls

`_syncIsolatedPendingStake()` with aggregate pending deposits equal to `P`. If `P >= I`, the function returns because `actualPendingDeposits >= pendingStake`, so ShMonad leaves `I` locally pending despite it already being native-active.

The crank then records `snapshotProductiveIsolatedStake == 0` again. During the next reward window, the isolated stake remains native-active, but the next validator crank still splits rewards using the stale zero productive snapshot. `_splitStakingYieldByProductiveSnapshot()` therefore assigns the claimed amount to pooled shMON. Only afterward, once `P` has also activated and aggregate pending becomes zero, does balance verification finally promote `I`.

This can persist even after a validator crank completes. The completed crank observes native state, but the later pooled pending deposit still masks the earlier isolated activation, so ShMonad records another zero productive isolated snapshot. Any rewards settled against that snapshot are permanently credited to pooled shMON holders instead of the isolated vShare holders whose principal earned them.

M-06 Pooled Pending Masks Isolated Stake

CONTINUED

— RECOMMENDATION

Track source and activation-epoch metadata for pooled and isolated pending deposits. When an isolated cohort's activation epoch has passed while known later pooled stake remains pending, include that isolated cohort in the productive reward snapshot before splitting newly claimed rewards, while retaining conservative principal handling for withdrawals.

— RESOLUTION

Status: Resolved.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/833>

MEDIUM **M-07**

RESOLVED

Mutable Signal Pins Crank

CATEGORY	DoS
LOCATION	https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/9f116de99a16a35f1889e08880e0b50141d28701/src/shmonad/libraries/StakeAllocationLib.sol#L76
REVIEW	Main Review 2

— DESCRIPTION

When `includeIsolatedRewardsInStakeAllocation` is enabled, a validator with zero pooled and isolated snapshots can use mutable live isolated stake as its allocation signal.

`allocationSignalStake()` normally uses `snapshotIsolatedStake`, but when both snapshots are zero it falls back to `activeStake + pendingStake`. During the global crank prepass,

`_recomputeCurrentAllocationTotals()` evaluates this fallback and persists the global denominator.

That signal can change before the same validator's allocation is applied. An isolated deposit can still be locally recorded as `pendingStake` when the global prepass runs. Later in the validator crank, balance verification can promote it into `activeStake`, and an isolated full exit can move the same stake into

`pendingUnstake`. At that point `activeStake + pendingStake` becomes zero.

`_computeStakeDelta()` then recomputes the validator's signal from the now-zero fallback. This causes `_adjustedStakingYield()` to use a different basis than the cached global denominator. With the default 80% deadband, a validator that dominated the cached denominator can receive an allocation target far above its legitimate share of the pooled staking queue.

If ShMonad does not have enough raw MON to fund the inflated allocation, `_initiateStaking()` reaches `assert(stakingAmount <= address(this).balance)`. Because the global pass and validator cursor are already persisted, later public `crank()` attempts hit the same validator state and revert again, blocking epoch progress until liquidity or an upgrade repairs the state.

If enough raw MON exists only because ShMonad is holding protected reserves, redemption liquidity, atomic liquidity, or isolated liquid assets, the inflated stake can instead be delegated while those accounting buckets remain recorded as available.

— RECOMMENDATION

Cache each validator's allocation signal during `_recomputeCurrentAllocationTotals()` and reuse that exact value in `_computeStakeDelta()`. Also cap pooled staking increases by actual pooled-stakable capital, not raw `address(this).balance`.

M-07 Mutable Signal Pins Crank

CONTINUED

— RESOLUTION

Status: Resolved.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/833>

MEDIUM **M-08**

ACKNOWLEDGED

Coinbase spoofing redirects validator rewards

CATEGORY Logical Error
LOCATION `_legacyOperationalConfigOrDefault()`
REVIEW Main Review 2

— DESCRIPTION

ShMonad is moving validators from old coinbase addresses to new managed coinbase proxy contracts. During that move, the new proxy correctly asks the staking precompile: "who is the current validator auth?" But it also asks the old coinbase address: "where should commission go, and what rates should I use?" The old coinbase address may no longer be trustworthy. The repo even allows a custom validator coinbase only because it has no code at registration time, while warning that this must be checked offchain and could be an undeployed contract later. But during migration, if that old address now has code, ShMonad treats it as a legacy config source. The new proxy only checks that the old address says "yes, I am for validator X" via `VAL_ID()`. Then it copies the old address's reported commission recipient and rates. It does not verify that the old contract is a real ShMonad coinbase, that it belongs to this ShMonad, that its auth matches the current live auth, or that it is known-safe code.

So the attack is

1. A validator used an old custom coinbase address L
2. Later, the validator auth changes, so the old controller should no longer control rewards.
3. Before migration, the old controller makes L contain code, for example EIP-7702 or CREATE2
4. That code lies and says: "I am validator V, send 100% commission to attacker."
5. Owner runs the planned migration.
6. The new proxy gets the correct current auth, but poisoned reward settings
7. Anyone can call `crank()`, which is permissionless
8. The crank sends pending validator rewards into the new coinbase, and `process()` pays commission to the imported recipient

M-08 Coinbase spoofing redirects validator rewards

CONTINUED

The impact is theft of validator reward money, especially pending MEV rewards and priority-fee balance sitting on the new coinbase. former controller can steal funds after they should no longer have authority

— RECOMMENDATION

Consider removing automatic operational config import from arbitrary legacy addresses. Initialize the new proxy entirely from the current staking-precompile auth and require that auth to explicitly configure it. For atomic migration, accept a configuration signed by the current validator auth using EIP-712, or leave the proxy in an unconfirmed state in which `handleMEVPayable()` and `process()` cannot execute until the current auth confirms the settings.

— RESOLUTION

Status: Acknowledged.

MEDIUM **M-09**

ACKNOWLEDGED

Queued validator rewards dilute existing holders

CATEGORY	Logical Error
LOCATION	<code>_handleValidatorRewards()</code>
REVIEW	Main Review 2

— DESCRIPTION

shMON is like a ticket that owns a slice of the pooled MON vault. If a reward was already sent for old stakers, only the old tickets should benefit from it. But here, a new user can buy tickets after the reward is already known / queued, before it is actually settled, and then share in that old reward

`sendValidatorRewards()` does not immediately count the validator payout as shMON holder revenue. It stores it as `rewardsPayable` and also reserves the MON:

Deposit pricing uses current equity to decide how many shMON shares to mint. But equity subtracts `rewardsPayable`, so the queued reward is invisible to the new depositor's price

Later, validator crank sends that payable reward through `externalReward()` / coinbase settlement, When ShMonad later claims it, `_handleEarnedStakingYield()` adds it as pooled yield for the current shMON supply

So the bug is a timing mismatch

1. Reward is already funded and queued.
2. New depositor enters at a price that ignores that future reward.
3. Reward later becomes shMON yield.
4. The new depositor now owns part of the supply and receives part of the old reward.

Assume, old holders own 1,000 shMON backed by 1,000 MON. A 1,000 MON reward is queued. Attacker deposits 1,000 MON and gets about 1,000 shMON because the queued reward is excluded from pricing. When the reward returns as pooled yield, there are now 2,000 shMON, so the attacker owns 50% and captures about 500 MON of a reward that existed before they entered.

The recent-revenue smoother is not a full fix. It makes immediate exits less profitable by subtracting recent revenue from exit pricing, but it does not stop the attacker from entering before settlement and waiting until the smoothing window decays

this is not insolvency or direct theft of principal. It is economic dilution. Existing shMON holders lose part of validator rewards they already economically earned, and late depositors can capture that value. The captured amount depends on the queued reward size and

M-09 Queued validator rewards dilute existing holders**CONTINUED**

how much temporary MON the attacker deposits. If the queued reward is large, the value shift can be large.

— RECOMMENDATION

Snapshot reward entitlement when rewardsPayable is created, or otherwise prevent shares minted after sendValidatorRewards() from participating in that pending validator reward.

Another approach is to track queued validator payouts as pending recent revenue for deposit pricing as well as withdrawal pricing, but the full fix is reward-debt / accumulator accounting tied to the share supply at reward submission time

— RESOLUTION

Status: Acknowledged.

MEDIUM **M-10**

RESOLVED

Atomic withdrawals leak deferred holder yield

CATEGORY Logical Error
LOCATION `_previewWithdraw()`
REVIEW Main Review 2

— DESCRIPTION

shMON as a receipt for a slice of the vault MON. If the vault earns 100 MON of new rewards, the old shMON holders should own that reward.

A new person who enters after the reward was earned should not be able to claim it

The problem is that ShMonad uses two different prices

Deposits use full vault value: `previewDeposit()` uses `deductRecentRevenue: false`

Withdrawals use reduced vault value: `maxWithdraw()` and `_previewWithdraw()` use `deductRecentRevenue: true`

That reduced value comes from `_recentRevenueOffset()`, which temporarily hides recent revenue from exit pricing in `FLERC4626.sol`, line 410

This is wrong because, when the victim withdraws, `_withdraw()` burns their shares permanently, There is no separate "you still own your delayed reward" record.

Storage only tracks balances, global revenue, liabilities, and atomic pool state, not per-user deferred revenue ownership

So the victim's shares are destroyed, but the hidden reward stays in the vault. whoever still has shares later gets that reward. So an attacker can buy a tiny amount of shares right before the victim exits, then wait and collect part of the victim's old reward.

The victim can lose a large part of revenue that was earned while they were the holder. The attacker's profit is capped by available atomic withdrawal liquidity and fees, but the value transfer is real and repeatable around large exits

— RECOMMENDATION

Deferred revenue can have explicit ownership rather than being represented only by a global deduction from equity

— RESOLUTION

Status: Resolved.

MEDIUM **M-11**

RESOLVED

Inactive cache misroutes active validator payout

CATEGORY	Logical Error
LOCATION	<code>_handleValidatorRewards()</code>
REVIEW	Main Review 2

— DESCRIPTION

ShMonad can accidentally treat a live validator as inactive because it trusts an old cached flag, when that happens, money that should be saved for the validator / delegators is instead credited as yield to shMON or isolated vShare holders

When someone sends validator rewards through `sendValidatorRewards()`, the reward is supposed to be split like this

- a protocol fee goes to shMON/isolated holders

- an owner commission may be taken

- the remaining validator payout is stored as `rewardsPayable`

later, ShMonad sends that payout to Monad's staking precompile using `externalReward()`

The docs say this directly: active registered validators should have their validator facing payout escrowed as `rewardsPayable`

ShMonad has its own local memory of whether a validator is in Monad's active set

`inActiveSet_Current`

`inActiveSet_Last`

That local flag is only refreshed when that validator is cranked. A crank is basically ShMonad's maintenance loop

So this bad state can exist

```
``plain text
validator is actually live again in Monad
but ShMonad still has inActiveSet_Current = false
``
```

Then if `sendValidatorRewards()` is called before ShMonad cranks that validator, `_handleValidatorRewards()` takes the wrong branch

Instead of this

```
``plain text
validator payout -> rewardsPayable -> externalReward()
``
```

M-11 Inactive cache misroutes active validator payout

CONTINUED

it does this

```
``plain text
validator payout -> holder yield
``
```

That is wrong because the payout belongs to the validator / delegators, not to whoever currently owns shMON or isolated vShares.

If the validator has no pooled ShMonad stake but does have isolated stake,

_splitIsolatedAndPooledBySnapshot() can send the whole misclassified reward to isolated accounting

Then _creditIsolatedRewards() increases the isolated assets backing the current isolated vShares.

Lets isolated holders burn vShares into a fixed MON redemption claim.

So the attacker can own the isolated shares, wait for the stale false active-set window, receive the

misclassified reward as isolated value, then call requestUnstakeIsolated() to lock in the gain, example

Suppose 100 MON comes in as validator reward.

Correct result

```
``plain text
9 MON -> holder/protocol yield
1 MON -> owner commission
90 MON -> validator/delegator payout
``
```

Buggy stale cache result

```
``plain text
99 MON -> holder/isolated yield
1 MON -> owner commission
0 MON -> validator/delegator payout
``
```

If the attacker owns all isolated vShares for that validator, they can capture almost the whole 99 MON, including the 90 MON that should have gone to the validator/delegators

Later, ShMonad may crank the validator and set inActiveSet_Current = true again

But by then the damage is already done

No rewardsPayable was created, no reserve was created for the validator payout

isolated assets were already increased

the attacker may already have converted vShares into a fixed redemption claim

There is no accounting record saying this 90 MON was misrouted; claw it back

This is a direct value transfer from validator / delegator rewards to current shMON or isolated vShare holders.

M-11 Inactive cache misroutes active validator payout

CONTINUED

The worst case is isolated vShares because the attacker can concentrate ownership around one validator with only the minimum isolated position, they can capture a much larger incoming validator payout if the stale-cache window exists

— RECOMMENDATION

A registered lifecycle active validator payout should always be escrowed, regardless of the cached active set bit

— RESOLUTION

Status: Resolved.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/831>

MEDIUM **M-12**

RESOLVED

Queued unstake makes delay branch unreachable

CATEGORY Logical Error
LOCATION `_requestUnstakeIsolated()`
REVIEW Main Review 2

— DESCRIPTION

queuedUnstake is only a note saying withdraw this stake later. It is not liquid MON, and it is not a started withdrawal yet. but requestUnstakeIsolated() creates that note, then immediately counts it as coverage when deciding when the user can withdraw

The wrong part is that, the isolated unstake request first increases redemptionsPayable, then tops up queuedUnstake until the request looks covered, then calculates the completion epoch

Then the completion helper counts queuedUnstake together with real / near real liquidity idleAssets + pendingUnstake + queuedUnstake + activeStake

Because the function just added enough queuedUnstake, the "add 2 more epochs" branch cannot be reached through the normal request path.

That matters because the docs say pending-backed isolated exits should get the extra 2 epochs when pending stake must activate first

So a user can deposit isolated MON, while that deposit is still pending, and immediately request to exit. The protocol says "you can withdraw at N+5," but the user's own stake may only be truly withdrawable at N+7.

If no liquid MON exists, their completion just reverts. worse, if another older user's withdrawal has produced liquid MON in the shared idleAssets pot, the newer user can complete first and take that MON.

There is no per-user reservation in completion. The code only checks whether the shared validator-level idleAssets has enough MON, so it does not verify that the MON came from the completing user's own withdrawal.

the practical damage is queue jumping: a newer exit can get paid early using an older exit's settled MON. The older user then reverts and waits longer, carrying the delay / risk while the attacker already has liquid MON

— RECOMMENDATION

Calculate the completion epoch before creating or increasing queuedUnstake, and never count queuedUnstake as an asset

M-12 Queued unstake makes delay branch unreachable

CONTINUED

— RESOLUTION

Status: Resolved.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/833>

MEDIUM **M-13**

RESOLVED

Reward split ignores pre exit stake mix

CATEGORY	Logical Error
LOCATION	<code>_settleValidatorRewardsPayable()</code>
REVIEW	Main Review 2

— DESCRIPTION

ShMonad has two kinds of stake on the same validator

pooled stake: the shared shMON pool owned by normal shMON holders

isolated stake: a validator-specific position owned by isolated vShare holders

When a validator reward arrives, ShMonad must decide: how much belongs to pooled users, and how much belongs to isolated users?

The rule should be, split the reward using the stake that existed when the reward was actually created

Here, that does not happen

In `_crankValidator()`, the order is

1. Claim old staking rewards.
2. Settle validator rewards payable.
3. If direct fallback is used, `\_sendRewards()\` calls `Monad \externalReward()\`
4. Later in the same crank, ShMonad may remove pooled or isolated stake
5. Then `\_rollValidatorEpochForwards()\` saves the new stake snapshot

The bad part is step 3. `\_sendRewards()\` immediately calls `\externalReward()\` on the staking precompile

Monad records that reward using the current validator stake. But ShMonad does not immediately claim and split it

It waits until a later crank, then `\_handleEarnedStakingYield()\` splits it using the saved productive snapshot

So the issue is

- reward is earned using the old stake mix;
- stake is changed;
- snapshot is updated;
- reward is later split using the new stake mix

Assume before the crank

```plain text

pooled stake: 1,000,000 MON

**M-13** Reward split ignores pre exit stake mix

CONTINUED

isolated stake: 1 MON

reward: 100,000 MON

```
``
Correct split
`plain text
pooled gets almost all of it
isolated gets about 0.1 MON
``
```

But if the pooled stake exits in the same crank, the next snapshot can look like

```
``plain text
pooled stake: 0
isolated stake: 1 MON
``
```

Then ShMonad later claims the reward and says:

```
``plain text
pooled gets: 0
isolated gets: 100,000 MON
``
```

That is wrong because the isolated user only had 1 MON when the reward was created

The impact is a transfer of value from pooled shMON holders to an isolated vShare holder. The attacker does not need to provide the big reward;

they only need a small isolated position on a validator where a large direct validator reward is about to be settled and pooled stake is about to leave.

The accounting still balances globally, so this can look like normal isolated staking yield instead of an obvious insolvency

The normal contract-coinbase path mostly avoids this because `\Coinbase.process()\` happens after the stake delta and before the new snapshot.

The vulnerable case is the supported fallback path: EOA coinbase, 7702 coinbase, or failed coinbase contract handling.

**— RECOMMENDATION**

Consider claiming immediately after every successful direct `_sendRewards()` payment, before balance verification or either unstake path

**— RESOLUTION**

Status: Resolved.

**M-13** Reward split ignores pre exit stake mix

CONTINUED

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/837>

MEDIUM **M-14**

RESOLVED

# Rolling pooled pending locks isolated redemption

CATEGORY Logical Error  
 LOCATION `_syncIsolatedPendingStake()`  
 REVIEW Main Review 2

## — DESCRIPTION

ShMonad has pooled stake and isolated stake on the same validator. Monad only reports total pending stake, not whether it belongs to pooled or isolated users. So ShMonad guesses conservatively

```
``plain text
function _syncIsolatedPendingStake(uint64 valId, uint256 actualPendingDeposits) internal {
 IsolatedValidatorState storage state = s_isolatedValidatorState[valId];
 if (state.pendingStake == 0) return;
 if (actualPendingDeposits >= state.pendingStake) return;
 uint120 activatedAmount = (uint256(state.pendingStake) - actualPendingDeposits).toUint120();
 state.pendingStake -= activatedAmount;
 state.activeStake += activatedAmount;
}
``
```

That guess is safe in one way, but bad here. If Alice exits 1,000 MON, ShMonad burns her isolated shares and stores a fixed MON claim

```
``plain text
uint256 remainingShares = availableShares - vShares;
s_isolatedVShareBalances[account][validatorId] = remainingShares;
s_isolatedVTTotalShares[validatorId] -= vShares;
state.redemptionsPayable += amountMon120;
// Multiple requests from the same account and validator collapse into one fixed-principal
entry.
request.amountMon = (uint256(request.amountMon) + amountMon).toUint128();
request.completionEpoch = completionEpoch;
\`
```

If unrelated pooled pending stake keeps masking 10 MON, ShMonad may only withdraw 990 MON

```
``plain text
```

**M-14** Rolling pooled pending locks isolated redemption

CONTINUED

```

if (state.isolatedWithdrawalPending || state.queuedUnstake == 0 || state.activeStake == 0) return;
uint120 amountToWithdraw = Math.min(uint256(state.queuedUnstake),
uint256(state.activeStake)).toUint120();
bool success = _initiateWithdrawal(valId, amountToWithdraw, withdrawalId);
state.pendingUnstake += amountToWithdraw;
state.queuedUnstake = amountToWithdraw >= state.queuedUnstake ? 0 : state.queuedUnstake -
amountToWithdraw;
state.activeStake -= amountToWithdraw;
...

```

Then completion is all or nothing. If the request says 1,000 MON but only 990 MON is idle, It reverts

```

` ` plain text
IsolatedValidatorState storage state = s_isolatedValidatorState[validatorId];
amountMon = request.amountMon;
require(state.idleAssets >= amountMon, InsufficientIsolatedIdleAssets(validatorId, amountMon,
state.idleAssets));
...

```

The deeper reason this can persist is ordering: isolated sync and withdrawal happen before new pooled stake allocation is applied in the same validator crank

```

` ` plain text
bool haltedByPrincipalDeficit = _handleStakeBalanceVerification(_valId, validatorEpochPtr_N(-1,
_valId));
_restakeIsolatedRewards(_valId);
_queueAndInitiateIsolatedUnstake(_valId);
// Calculate and then handle the net staking / unstaking
...
(_nextTargetStakeAmount, _netAmount) = _applyStakeDelta();
...

```

And the pooled increase path can then schedule more pending stake

```

` ` plain text
// Deploy additional stake to the validator
(bool _success, uint128 _actualAmount) = _initiateStaking(valId, netAmount);
_handleInitiateIncreasedAllocation(valId, netAmount.toUint120());
...

```

**M-14** Rolling pooled pending locks isolated redemption

CONTINUED

So a small rolling pooled pending amount can block a much larger isolated exit. The victim's isolated shares are already burned, their full claim cannot complete, and there is no partial payout or cancel path to recover the already-returned principal

**— RECOMMENDATION**

Consider tracking pending cohorts by source and activation epoch, record pooled and isolated delegations separately for both the normal and boundary pending tracks.

And use the precompile's deltaEpoch and nextDeltaEpoch instead of discarding them, and promote an isolated cohort when its recorded activation epoch has passed, rather than inferring activation from aggregate pending value.

**— RESOLUTION**

Status: Resolved.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/833>

MEDIUM **M-15**

RESOLVED

## Stale Cache Redirects Validator Rewards

CATEGORY Rewards

LOCATION <https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/9f116de99a16a35f1889e08880e0b50141d28701/src/shmonad/StakeTracker.sol#L2273>

REVIEW Main Review 2

### — DESCRIPTION

`_handleValidatorRewards()` uses the cached `inActiveSet_Current` flag to decide whether a payment belongs to a registered validator or should become yield for ShMonad holders.

This flag can be false even after the validator becomes rewardable in Monad's current native epoch. For example, ShMonad can register a validator while its native delegation is pending. `addValidator()` records `isActive = true`, but initializes `inActiveSet_Current = false` because the validator has no current consensus stake.

The delegation can then activate at the next native epoch boundary. The validator now has consensus stake and can receive external rewards, but ShMonad's cached flag remains false until `_crankValidator()` processes that validator.

If `sendValidatorRewards()` is called between native activation and the validator crank, `_handleValidatorRewards()` enters the branch intended for inactive or unregistered validators. It sets `validatorPayout = 0` and converts `feeTaken + validatorPayout` into pooled and isolated ShMonad yield. The payment is therefore never recorded in the validator's `rewardsPayable` bucket. No matching global reward liability or reserve is created. Instead, the amount is credited to ShMonad holder equity and queued for staking, so a later cache refresh cannot restore the validator's claim.

With a 100 MON payment and `feeRate = 0`, the correct branch records a 100 MON validator liability and reserves 100 MON for settlement. Under the stale-false state, all 100 MON becomes ShMonad holder yield and the validator receives nothing.

The impact is an irreversible transfer of up to the entire validator payout from the validator and its native delegators to current shMON and isolated holders.

### — RECOMMENDATION

When `inActiveSet_Current` is false but the validator remains registered, query the native staking precompile before redirecting the payment. Alternatively, escrow payments for every protocol-active validator and redirect them only after settlement confirms native ineligibility.

**M-15** Stale Cache Redirects Validator Rewards

CONTINUED

**— RESOLUTION**

Status: Resolved.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/831>

MEDIUM **M-16**

RESOLVED

## Dust Sweep Strands Isolated Claims

|          |                                                                                                                                                                                                                                                                                                                                               |
|----------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| CATEGORY | Logical Error                                                                                                                                                                                                                                                                                                                                 |
| LOCATION | <a href="https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/9f116de99a16a35f1889e08880e0b50141d28701/src/shmonad/StakeTracker.sol#L2157">https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/9f116de99a16a35f1889e08880e0b50141d28701/src/shmonad/StakeTracker.sol#L2157</a> |
| REVIEW   | Main Review 2                                                                                                                                                                                                                                                                                                                                 |

### — DESCRIPTION

A full pooled withdrawal can sweep isolated dust into pooled accounting without preserving the isolated redemption obligations backed by that stake.

An older isolated withdrawal can occupy the validator's isolated withdrawal lane. During a boundary transition, a pooled deposit of  $P$  can be queued while a new isolated deposit of  $P + d$  is delegated before the boundary and scheduled for  $E+1$ . The validator crank can then resume during the delay period and delegate pooled  $P$  for  $E+2$ , where  $d$  is below the 1 gwei dust threshold.

At  $E+1$ , Monad reports  $P$  as aggregate pending delegation. `_syncIsolatedPendingStake()` therefore attributes only  $d$  as active isolated stake and leaves  $P$  as isolated pendingStake. If the validator is then deactivated, the forced full pooled exit reaches `_initiateStakeAllocationDecrease()`.

The function identifies  $d$  as isolated dust, adds it to the pooled withdrawal, clears isolated activeStake, and subtracts it from `s_isolatedTotalAssets`. However, the returned dust is processed entirely as pooled capital. The users' fixed isolated requests, `redemptionsPayable`, and `queuedUnstake` are not reduced or credited with the swept amount.

If no isolated-attributed reward or withdrawal surplus covers  $d$  before the isolated snapshots clear, all remaining isolated principal can settle while the validator has no active, pending, or withdrawing isolated principal and fixed redemptions exceed isolated `idleAssets` by  $d$ .

Isolated requests are completed atomically. Because `completeUnstakeIsolatedFor()` is permissionless, another matured request can be completed first, leaving a materially larger request reverting because it is short by less than 1 gwei. The residual `queuedUnstake` has no clearing path once no isolated active principal remains and also prevents normal validator removal and re-addition.

Although the accounting deficit is below 1 gwei, the amount rendered inaccessible can be the full value of an arbitrarily large isolated redemption until the accounting is repaired.

**M-16** Dust Sweep Strands Isolated Claims

CONTINUED

**— RECOMMENDATION**

Track the isolated portion included in a pooled withdrawal separately. Keep that amount within isolated asset accounting while the withdrawal is pending, reconcile the corresponding queuedUnstake, and credit it to isolated idleAssets when the native withdrawal completes. Process only the genuinely pooled portion through pooled settlement. Do not simply disable the sweep, as that could restore the native dust mismatch it was intended to remediate.

**— RESOLUTION**

Status: Resolved.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/838>

MEDIUM **M-17**

RESOLVED

# Boundary window permits unpaid reward diversion

CATEGORY Validation  
LOCATION `_canReceiveExternalReward()`  
REVIEW Main Review 2

## — DESCRIPTION

The contract mistakes cannot pay rewards right now for these rewards are abandoned forever during an epoch switch, that can be false: the validator may be temporarily unpayable now, but clearly payable again after the delay

coinbase holds unpaid validator / delegator rewards in `s_unpaid.rewards`

`process()` tries to send those rewards through `Monad.externalReward()`

If that cannot work, `donateUnpaidRewardsToShMonad()` lets anyone move the whole unpaid buffer to `ShMonad`

The problem is the check only asks: can this validator receive rewards at this exact moment?

`Monad` has a short epoch delay window, during that window, rewards use the old `snapshotStake`

After the window, rewards use the new `consensusStake`

So a validator can have `snapshotStake = 0` but `consensusStake > 0`: not payable now, but definitely payable soon

The donation function still allows deleting the reward buffer in that temporary state

Anyone can call the donation function during that window, the entire unpaid reward buffer is zeroed from `Coinbase` and sent to generic `ShMonad` yield

And the original intended delegators / validator reward group lose their claim, there is no later record to reconstruct who should have received it

## — RECOMMENDATION

Consider blocking donation whenever the validator appears in any current or upcoming validator view

## — RESOLUTION

Status: Resolved.

**M-17** Boundary window permits unpaid reward diversion

CONTINUED

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/834>

MEDIUM **M-18**

ACKNOWLEDGED

## Temp capital creates permanent validator score

CATEGORY Logical Error  
LOCATION `_handleValidatorRewards()`  
REVIEW Main Review 2

### — DESCRIPTION

when the protocol decides which validators should receive pooled stake. that decision is based partly on a validator revenue score, validators that look like they brought more rewards can receive more future pooled MON

The bug is that a validator can make fake looking revenue with the same MON over and over  
Example flow

1. The attacker controls a validator and waits until that validator is the current proposer.
2. They call `boostYield\{value: 100 MON\}`
3. The code treats that 100 MON as validator revenue signal in `_handleBoostYield()` if the isolated-reward allocation toggle is enabled.
4. The same 100 MON is also credited as isolated liquid value for the attacker's isolated shares
5. The attacker burns some isolated shares, which moves that just-added MON into `idleAssets`
6. Because the request is now fully backed by idle MON, the completion epoch becomes "now"
7. The attacker immediately completes the isolated unstake and receives almost all the 100 MON back.
8. The validator revenue score is not reduced

The system thinks the validator permanently contributed useful revenue, but economically the validator took the money back. The "scoreboard" says "this validator produced 100 MON of value," while the protocol no longer keeps that 100 MON

So the attacker can buy a lot of validator reputation for much less than the displayed amount. With enough repeated loops, their validator can look like the best revenue producer. Then future pooled stake can be routed to their validator, and turnover can move stake away from honest validators toward the attacker

**M-18** Temp capital creates permanent validator score

CONTINUED

**— RECOMMENDATION**

Consider excluding isolated rewards from pooled stake-allocation scoring. If isolated rewards must contribute to the pooled allocation signal, they need a retained-value vesting mechanism

**— RESOLUTION**

Status: Acknowledged.

MEDIUM **M-19**

ACKNOWLEDGED

# Unstake burns erase deferred reward claims

CATEGORY Logical Error  
LOCATION `_requestUnstake()`  
REVIEW Main Review 2

## — DESCRIPTION

when the pool earns fresh revenue, that revenue belongs to the people who held shMON  
The protocol temporarily hides fresh revenue from exit pricing. that is meant to stop someone from depositing right before revenue and instantly withdrawing it

But `requestUnstake()` does something dangerous

1. It calculates the user's MON payout using the discounted price.
2. It immediately burns the user shMON
3. It stores only a fixed MON amount for later withdrawal.
4. It does not store the user's claim to the fresh revenue that was excluded.

So the exiting user loses their "receipt" before the hidden revenue becomes claimable

An attacker can hold only a tiny amount of shMON, then a large holder exits. their shares are burned, but the fresh revenue stays in the pool

After the revenue smoothing period passes, the tiny attacker is one of the only remaining shMON holders, so they receive a huge part of the old revenue

Plainly: the victim receipt is destroyed too early, and the attacker's tiny receipt inherits the victim's delayed reward

The system is only supposed to delay access to fresh rewards, not change who owns them. but

`\requestUnstake()\` burns the victim's shares without storing any "this user still owns their part of recent revenue" record

So a user doing a normal unstake after recent revenue can lose most of their deferred revenue. and a front runner can capture that value with tiny capital

## — RECOMMENDATION

The protocol must preserve the exiting cohort ownership of deferred revenue

## — RESOLUTION

Status: Acknowledged.

MEDIUM **M-20**

ACKNOWLEDGED

## Failed rewards ignore validator lifecycle change

CATEGORY Logical Error  
LOCATION `process()`  
REVIEW Main Review 2

### — DESCRIPTION

ShMonad sends validator reward money into a validator coinbase contract. If Coinbase cannot forward the money to Monad staking, it stores it as `s_unpaid.rewards`,

The issue is that this stored money is just one bucket. It does not remember who was supposed to receive it or which validator / stake lifecycle it belongs to

The bad flow is

1. Old rewards fail to pay because the validator is not rewardable
2. The money stays inside Coinbase as unpaid rewards.
3. ShMonad has already cleared its own liability/reserve for that money.
4. Later, the validator becomes rewardable again.
5. `process()` adds the old unpaid rewards into the next `externalReward()` call.
6. Monad gives that reward to the current delegators, not necessarily the old ones.

The reward was historical money. It should not become free money for whoever stakes later. But

`process()` blindly does `_rewardPortion += _unpaidRewards`,

while `donateUnpaidRewardsToShMonad()` only checks whether the validator is rewardable right now, If the validator was reactivated, donation reverts

### — RECOMMENDATION

**M-20** Failed rewards ignore validator lifecycle change

CONTINUED

Recovery eligibility must be latched when the failed payment occurs, rather than recomputed when someone eventually calls the donation function

```
``plain text
if (!_rewardsSucceeded) {
if (!_canReceiveExternalReward(valid)) {
s_recoverableRewards += _rewardPortion.toUint120();
s_unpaid.rewards = 0;
} else {
s_unpaid.rewards = _rewardPortion.toUint120();
}
}
``
```

**— RESOLUTION**

Status: Acknowledged.

MEDIUM **M-21**

ACKNOWLEDGED

# Coinbase auth bypasses reward snapshot ordering

CATEGORY Logical Error  
LOCATION `processCoinbaseByAuth()`  
REVIEW Main Review 2

## — DESCRIPTION

The issue is that ShMonad lets a validator auth process the Coinbase reward path outside the normal validator crank order, but the reward is later split using ShMonad stored pooled / isolated snapshots. If those snapshots no longer match who actually had productive stake when the reward was injected, the money is credited to the wrong group.

ShMonad has two internal owner groups, pooled shMON holders: everyone shares one pool and isolated vShare holders: users tied to a specific validator.

Monad native staking precompile does not know this internal split. It mostly sees ShMonad as one delegator. So ShMonad must keep its own photo of the stake split and later use that photo to divide rewards fairly, which the snapshot

Normally `_crankValidator()` is carefully ordered

1. claim old staking rewards
2. verify/sync stake
3. apply stake changes
4. process the funded coinbase contract
5. roll the validator epoch and refresh snapshots

That order is visible in StakeTracker.sol,

But `processCoinbaseByAuth()` skips that full sequence. It checks that the caller is the validator auth or owner, then directly calls `_settleCoinbaseContract()`

Inside Coinbase processing, priority fees are split into commission, donation, and priorityStakers.

The priorityStakers part is sent into Monad via `externalReward()`. After that, ShMonad does not remember, this reward came from Coinbase at this exact stake split.

Later, `_claimRewards()` only sees some MON reward arrived. Then `_handleEarnedStakingYield()` splits it using `_splitStakingYieldByProductiveSnapshot()`

That splitter uses `snapshotProductiveIsolatedStake`. Example

Say a 100 MON Coinbase reward is injected when the real productive stake is 90% pooled, 10% isolated.

**M-21** Coinbase auth bypasses reward snapshot ordering

CONTINUED

Fair result: pooled gets 90 MON, isolated gets 10 MON

But if ShMonad later splits the claimed reward using a stale snapshot that says 10% pooled, 90% isolated

Then the result becomes pooled gets 10 MON, isolated gets 90 MON. Nothing disappears, but 80 MON moved to the wrong owners

The reverse can also happen: isolated holders can lose rewards to pooled holders

The impact is internal value theft / misallocation, pooled shMON holders can be diluted or overpaid,

Isolated vShare holders can be underpaid or overpaid

Rewards can look like legitimate staking yield even though the ownership split is wrong and accounting can still balance globally, making the bug less obvious

**— RECOMMENDATION**

we need to not process Coinbase rewards directly from processCoinbaseByAuth(). consider making it queue a request, then settle inside \_crankValidator() only, or snapshot the exact reward entitlement / stake basis at externalReward() time and use that same basis when ShMonad later claims and accounts the rewards

**— RESOLUTION**

Status: Acknowledged.

MEDIUM **M-22**

ACKNOWLEDGED

# Tiny validator slices stall redemptions

CATEGORY Logical Error  
LOCATION `_applyStakeDelta()`  
REVIEW Main Review 2

## — DESCRIPTION

ShMonad records users need MON back in one big withdrawal queue, but validators are processed one by one. The global queue says this amount is bigger than dust, try to withdraw it, then the per-validator logic splits that amount so thinly that every validator says my part is dust, skip it. So nobody withdraws anything, and the same leftover comes back next epoch

The exact bad loop is

1. A user uses traditional unstaking, `requestUnstake()` burns their shMON and records a fixed MON debt to them
2. The global crank only rolls unstake dust forward when the non-turnover queue is  $\leq \text{DUST\_THRESHOLD}$ , If it is  $\text{DUST\_THRESHOLD} + 1$ , it stays in the current round
3. Then `calculateValidatorEpochStakeDelta()` splits that global amount across validators pro rata
4. Each validator tiny slice goes through `_applyStakeDelta()`, If the slice is  $< \text{DUST\_THRESHOLD}$ , ShMonad does not call `undelagate`, it requeues the amount

Example: total leftover is 1 gwei + 1 wei. with two equal validators, each gets about 0.5 gwei. each piece is below the 1 gwei dust threshold, so both validators skip. the pieces are put back into the queue. next epoch, the same thing happens again

This is wrong because the code checks dust at the wrong level. It proves only the total queue is bigger than dust. It does not prove at least one validator will receive a withdrawable amount

So the system can believe there is executable withdrawal work while every actual validator action is rejected by ShMonad's own dust rule

A tiny missing tail can block a much larger traditional unstake, `completeUnstake()` pays the full matured amount or reverts; there is no partial claim. So if a user has a huge matured redemption and the only missing backing is this stuck gwei sized tail,

The whole redemption can remain unclaimable until unrelated new liquidity, future yield, atomic liquidity clears it

**M-22** Tiny validator slices stall redemptions

CONTINUED

**— RECOMMENDATION**

Withdrawal and staking dust must be handled differently

**— RESOLUTION**

Status: Acknowledged.

MEDIUM **M-23**

ACKNOWLEDGED

# Final isolated sweep rewards late minter

CATEGORY Logical Error  
LOCATION `_sweepOrphanedIsolatedIdleAssets()`  
REVIEW Main Review 2

## — DESCRIPTION

The issue happens when the last isolated user for a validator exits, Alice has isolated stake. When she asks to exit, her isolated vShares are burned and ShMonad records a fixed IOU: pay Alice 1,000 MON later, that fixed amount is stored as the isolated unstake request, later rewards do not increase it. The docs explicitly describe this fixed principal behavior, and the code pays `request.amountMon` in `_completeUnstakeIsolated()`.

During Monad withdrawal delay, the native withdrawal can return more than the principal, for example 1,050 MON instead of 1,000 MON, ShMonad accepts this because

`_finalizeCompletedIsolatedWithdrawal()` only requires `amountReceived >= expectedAmount`.

It puts the extra 50 MON into isolated accounting as surplus / reward.

The problem is the next step. If Alice was the last isolated holder, there are now

- no isolated vShares,
- no isolated users left,
- Alice fixed request still only says 1,000 MON,
- the extra 50 MON is sitting as leftover isolated idle value.

When anyone completes Alice's request, `_completeUnstakeIsolated()` pays Alice exactly 1,000 MON. Then `_sweepOrphanedIsolatedIdleAssets()` sees no isolated owner exists anymore and moves the leftover 50 MON into the pooled shMON bucket.

That sweep is the dangerous moment.

An attacker can do this:

1. Deposit into pooled shMON right before the sweep.
2. Call `completeUnstakeIsolatedFor()` for Alice. because it's permissionless.
3. The sweep moves the leftover isolated reward into pooled equity.
4. The attacker's newly minted pooled shares now own part of that old reward.

This is wrong because the reward came from an old isolated withdrawal, but it is assigned to whoever owns pooled shMON at the sweep moment.

The attacker did not help earn that reward; they just joined right before the accounting moved it.

**M-23** Final isolated sweep rewards late minter

CONTINUED

**— RECOMMENDATION**

The withdrawal reward must be assigned to a cohort that was fixed before it became observable and before an attacker can mint participating shares, consider adding an isolated withdrawal reward index.

**— RESOLUTION**

Status: Acknowledged.

MEDIUM **M-24**

RESOLVED

# Forced exit retries duplicate unstake demand

CATEGORY Logical Error  
LOCATION `_crankValidator()`  
REVIEW Main Review 2

## — DESCRIPTION

ShMonad has two different to do lists for unstaking

1. Global queue: `queueForUnstake`, meaning the protocol needs this much MON unstaked overall
2. Validator local state: a validator target stake, meaning this validator should still have this much stake unless we successfully move it

For normal user withdrawals, adding back to `queueForUnstake` after a failed unstake is correct, the user really asked to leave, so if the validator unstake fails, the protocol should try that same global demand again later

The bug happens on forced validator exits. In `_crankValidator`, if a validator is inactive or the new target would fall below `MIN_VALIDATOR_DEPOSIT`, the code replaces the normal queue derived amount with the validator full unstakable balance

If that forced undelegation fails, `_initiateStakeAllocationDecrease` restores the validator target and also requeues the full amount into global `queueForUnstake`

That is wrong because restoring the target already makes the same validator try the forced exit again later. Adding the same amount to the global queue creates a second, fake "someone needs to unstake" demand

Example:

1. Validator V is inactive and has 1,000 MON.
2. No user requested any unstake.
3. ShMonad tries to force V out.
4. The native undelegate call fails.
5. Correct result: V still needs to be retried
6. Actual result: V still needs to be retried, and global `queueForUnstake` also becomes 1,000 MON

Next round, that fake 1,000 MON global queue can be spread across other validators. so unrelated healthy validators may be unstaked even though the only real need was retry V's forced exit

The impact is capital availability and liveness damage, healthy validators can be unnecessarily undelegated, stake can enter the native withdrawal delay, staking yield can drop,

**M-24** Forced exit retries duplicate unstake demand

CONTINUED

deposits can be netted against fake unstake demand, and repeated failures can grow the fake queue

**— RECOMMENDATION**

TBD

**— RESOLUTION**

Status: Resolved.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/835>

MEDIUM **M-25**

RESOLVED

# Global rolls erase unvested revenue protection

CATEGORY Logical Error  
LOCATION `_updateRevenueSmootherAfterEpochChange()`  
REVIEW Main Review 2

## — DESCRIPTION

ShMonad tries to hide fresh rewards from exit pricing for 50,000 blocks, but a later global crank can accidentally forget that old hidden reward early.

When the pool earns validator rewards, the rewards should mostly belong to the people who were already in the pool.

To stop someone from depositing right before rewards are claimed and leaving right after, ShMonad uses a "recent revenue smoother"

Fresh rewards are temporarily subtracted from exit quotes, so new depositors cannot instantly steal old holders' rewards, that logic is in FLERC4626.sol

It keeps current revenue fully hidden, then slowly releases the previous reward over the 50,000 blocks

What is wrong, is that the smoother only stores one old reward amount

On every global roll, ShMonad overwrites that one stored amount

It does not ask: how much of the previous reward is still locked ?

It simply replaces the old locked reward with the newest epoch's reward

So if the next epoch reward is small or zero, the old locked reward disappears from the protection math

That does not remove MON from the contract, It changes the share price used for exits, the reward becomes claimable too early

`crank()` is permissionless and runs `_crankGlobal()` before validator cranking.

A global roll is allowed when all validators are done, Monad is not in the delay period, the raw Monad epoch advanced

There is no rule saying: wait 50,000 blocks since the last smoother update.

If one global roll happens late, the next one can happen much sooner than 50,000 blocks later.

Example

1. Old users have shares and have earned validator rewards.
2. Attacker deposits right before those rewards are claimed.
3. The rewards are claimed and recorded as recent revenue.

**M-25** Global rolls erase unvested revenue protection

CONTINUED

4. First global roll starts smoothing those rewards.
5. Second global roll happens too soon and overwrites the smoother.
6. The old reward is now treated as normal pool equity
7. Attacker calls `requestUnstake()`

`requestUnstake()` uses `previewUnstake()` to compute the MON amount, burns the attacker's shares, and stores that amount as `redemptionsPayable`.

The attacker has now turned the incorrect quote into a fixed redemption claim.

Sp existing holders lose rewards they earned before the attacker joined, the attacker captures part of the prematurely unlocked reward pool.

**— RECOMMENDATION**

remember multiple reward cohorts, or carry the still unvested portion of the previous cohort into the new smoother instead of overwriting it.

**— RESOLUTION**

Status: Resolved.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/836>

MEDIUM **M-26**

ACKNOWLEDGED

# Recycled rewards collapse isolated shares

CATEGORY Logical Error  
LOCATION `_creditIsolatedRewards()`  
REVIEW Main Review 2

## — DESCRIPTION

This is a accounting vuln, but depends on low or zero `boostCommissionRate` and attacker ownership of pooled shMON

Isolated stakers get validator specific receipt shares called vShares, those shares decide how much MON they can later claim from that validator. the bug lets an attacker make a validator look like this

```
``plain text
100 MON of real isolated stake
only 1 real vShare outstanding
``
```

That is wrong because 1 tiny share is now backed by a huge amount of MON

The bad flow is

1. The attacker is the only isolated staker for a validator.
2. They send fake / self funded validator rewards.
3. The code credits those rewards immediately to isolated equity and `queuedRestake` i
4. They immediately burn some isolated shares
5. The request path can move that same queued reward into idle liquidity and pay them back quickly
6. Their real stake remains, but their share count is smaller.
7. Repeating this compresses the real share supply down toward 1

The victim then deposits into that validator, the deposit uses this formula

```
``plain text
new vShares = deposit * (total vShares + 1) / (isolated equity + 1)
``
```

That is implemented in `StakeTracker.sol`. The +1 is a virtual share safety trick. It is usually harmless when there are many real shares, but it becomes dangerous when the attacker has compressed real shares to 1

Example

attacker setup: 100 MON backed by 1 vShare

**M-26** Recycled rewards collapse isolated shares

CONTINUED

victim deposits: 100 MON

victim receives: only 1 vShare

The deposit does not revert, because the code only checks did this mint zero shares?

The victim got 1 share, so the check passes, even though the price is terrible

The final damage happens when the validator is deactivated and all isolated stake becomes idle, direct isolated claims use the same virtual-share math, with 2 real shares and 200 MON total, each real share gets about:

$200 \text{ MON} / 3 = 66.67 \text{ MON}$

we divide by 3, because the formula counts: attacker real share, victim real share, 1 virtual share

So after attacker and victim claim, about `\ 66.67 MON\` is left behind for the fake virtual share, then

`_sweepOrphanedIsolatedIdleAssets()` moves that leftover into pooled shMON revenue

That is the core bug: money that should belong to isolated stakers can become pooled shMON value

This breaks the promise of isolated staking. Isolated principal should stay inside the isolated validator owner group. It should not leak into pooled shMON holders because of a rounding / virtual share artifact

Important, this is most dangerous when boost commission is zero or low. The documented upgrade plan says boost commission is 25%, which makes the example economically unattractive, but the contract allows the owner to set it lower. so this is not fully blocked by code, and can happens if it become zero later

**— RECOMMENDATION**

The protocol needs protections at both deposit time and generation finalization

**— RESOLUTION**

Status: Acknowledged.

MEDIUM **M-27**

ACKNOWLEDGED

# Pending isolated stake distorts allocation

CATEGORY Logical Error  
LOCATION `allocationSignalStake()`  
REVIEW Main Review 2

## — DESCRIPTION

The protocol decides which validators should receive more pooled MON by giving each validator a performance score

A validator score roughly asks: Did this validator earn more than we expected for the amount of stake it had?

If yes, it gets more future pooled stake. If no, pooled stake can move away from it.

The bug is that, there is also isolated stake: a user can put their own MON behind one specific validator, separate from pooled users

When the isolated-reward allocation toggle is enabled, the code does this

`allocationSignalStake()` adds isolated snapshot stake into the validator's scoring stake:

`StakeAllocationLib.sol`

`_setIsolatedSnapshots()` sets that isolated snapshot to `activeStake + pendingStake`

But staking yield rewards are split using only productive isolated stake, not pending stake

So an attacker can deposit isolated MON into a victim validator. That MON can be pending or recently added and may not have produced the historical yield being scored

But the allocator still counts it as if the victim validator had a much bigger stake basis

That is wrong, because the protocol compares old revenue against a new / temporary stake number

That makes the victim look bad

Validator A had huge stake, so it should have earned huge yield

But A did not really have that productive stake during the measured revenue period. The attacker just added temporary isolated principal. So A's score can be pushed down to zero even though A performed normally

Once A score is suppressed

new pooled stake can be routed away from A, and attacker validator can receive most or all of the new pooled queue

A can be treated as overallocated,

Incentive turnover can unstake pooled MON from A,

The attacker can later withdraw the isolated position, while the poisoned allocation result still affects that round

**M-27** Pending isolated stake distorts allocation

CONTINUED

This is an allocation manipulation bug, temporary isolated capital can move real pooled validator allocation, causing churn and giving the attacker's validator more future stake, commission, MEV, and influence

**— RECOMMENDATION**

Consider implementing the behavior stated in StakeTracker design comment: keep the stake-position side pooled only

If isolated rewards must be normalized by isolated stake, the protocol needs cohort consistent historical accounting.

**— RESOLUTION**

Status: Acknowledged.

MEDIUM **M-28**

RESOLVED

## withdrawal ownership erased during cleanup

CATEGORY Logical Error  
LOCATION `_completeDeactivatingValidator()`  
REVIEW Main Review 2

### — DESCRIPTION

a Monad withdrawal ID is a claim ticket, ShMonad asks Monad to undelegate stake, Monad creates a withdrawal ticket, and later ShMonad must come back with the same validator ID plus the same ticket ID to collect the MON

The bug is that ShMonad can create the ticket, then delete its own copy of the ticket number in the same validator crank

The dangerous order is

1. The validator is already deactivating
2. A final crank still finds some pooled stake that can be withdrawn
3. `_initiateStakeAllocationDecrease()` consumes a pooled withdrawal ID and calls `Monad undelegate()`
4. ShMonad records `hasWithdrawal = true` and increases `pendingUnstaking`
5. Then `_rollValidatorEpochForwards()` sees the deactivation timeout passed and calls `_completeDeactivatingValidator()`
6. The base removal deletes all validator epoch and pending records

This is wrong because Monad still has the real withdrawal ticket, but ShMonad deleted the local record that says ticket ID X belongs to this pending pooled withdrawal

The v1.2 safety check only waits for isolated-stake work, not pooled `hasWithdrawal`, pooled `pendingUnstaking`, target stake, or live native withdrawal IDs

After that, ShMonad can still detect that the ID is occupied because `_consumeWithdrawalId()` checks the live precompile withdrawal amount, but detection is not recovery. It only skips the ID

It does not know the owner / accounting record anymore, and the validator was removed from the crank list, so normal settlement will not call `_completeWithdrawal()` for that ticket

Impact is the MON in that native withdrawal can be stuck in Monad's staking precompile

ShMonad may still count the money as pending/staked even though it cannot collect it

Users can get wrong redemption / share pricing because accounting includes money that is not actually reachable

**M-28** withdrawal ownership erased during cleanup

CONTINUED

**— RECOMMENDATION**

Validator removal must be conditioned on both local and native pooled withdrawal quiescence

**— RESOLUTION**

Status: Resolved.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/820>

MEDIUM **M-29**

RESOLVED

# Zero snapshot misroutes pooled validator stake

|          |                                                |
|----------|------------------------------------------------|
| CATEGORY | Logical Error                                  |
| LOCATION | <code>_handleStakeBalanceVerification()</code> |
| REVIEW   | Main Review 2                                  |

## — DESCRIPTION

The bug is that old pooled stake can be forgotten during validator removal / readd, then later discovered and accidentally assigned to the first isolated depositor

1. A validator is removed / radded while old pooled MON is still pending / active in the Monad staking precompile
2. ShMonad's local validator record starts fresh with target stake 0
3. If the old stake is still pending, `_handleStakeBalanceVerification()` only emits an event and returns without freezing or quarantining it
4. The validator can roll forward and become ready for isolated deposits
5. Attacker deposits the minimum isolated stake, so `pendingStake = 1 MON`
6. Later, the old pooled stake becomes active. ShMonad sees extra active stake
7. To split that extra stake, it uses `_splitIsolatedAndPooledBySnapshot()`. but when snapshots are (0,0), the function falls back to live isolated stake
8. Since the only live local stake is the attacker isolated deposit, ShMonad says, this extra old stake belongs to isolated
9. It then increases `isolatedState.activeStake` and `s_isolatedTotalAssets`

The old MON existed before the attacker deposited. It came from the pooled side, so it must not become attacker-owned isolated equity. A reward-splitting fallback is being used for real principal, and that lets the newest isolated user claim old pooled money.

So the attacker's small isolated deposit can become backed by old pooled stake + attacker deposit. Then `requestUnstakeIsolated()` prices their vShares against that enlarged value and queues an isolated withdrawal

Because the precompile has one aggregate ShMonad delegation, the isolated withdrawal can pull the real MON that actually belonged to pooled holders

So the damage is direct theft from pooled shMON holders into the attacker's isolated redemption

Possible double-counted pooled NAV if the old stake was already in

**M-29** Zero snapshot misroutes pooled validator stake

CONTINUED

s\_globalCapital.stakedAmount, because the "discovered" stake is added again  
Later pooled redeemers may withdraw against phantom assets, making insolvency worse  
So once a reused validator has leftover pooled native stake, a public caller can use the zero snapshot  
isolated fallback to turn that pooled principal into isolated backing

**— RECOMMENDATION**

unexpected validator principal must not be distributed using reward splitting snapshots

**— RESOLUTION**

Status: Resolved.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/820>

MEDIUM **M-30**

RESOLVED

# Reward Claims Mask Validator Exit

|          |                                                                                                                                                                                                                                                                                                                                               |
|----------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| CATEGORY | Rewards                                                                                                                                                                                                                                                                                                                                       |
| LOCATION | <a href="https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/9f116de99a16a35f1889e08880e0b50141d28701/src/shmonad/StakeTracker.sol#L1132">https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/9f116de99a16a35f1889e08880e0b50141d28701/src/shmonad/StakeTracker.sol#L1132</a> |
| REVIEW   | Main Review 2                                                                                                                                                                                                                                                                                                                                 |

## — DESCRIPTION

A validator can leave Monad's active set while ShMonad continues treating it as active, leaving pooled stake allocated to a validator that no longer earns rewards.

Monad removes a validator when its auth address undelegates below the minimum self-stake requirement, even if third-party delegations remain. Existing delegators can still call `claimRewards()` after removal, and the call succeeds even when no rewards are payable.

During each validator crank, `_advanceActiveSetFlags()` shifts the previous membership flag and resets `inActiveSet_Current` to true. ShMonad does not verify active-set membership unconditionally.

`_markValidatorNotInActiveSet()` is only reached when another staking operation, such as claiming rewards, settling external rewards, delegating, or undelegating, fails.

If the validator auth exits while ShMonad's delegation remains, `claimRewards()` continues succeeding. When no other native operation is required, every crank leaves `inActiveSet_Current` equal to true. The validator therefore never records the two inactive observations required to trigger `_beginDeactivatingValidator()`.

ShMonad can consequently retain a material pooled allocation on a validator that cannot produce blocks or staking yield. The stale validator also remains included in allocation accounting, preventing the affected stake from being automatically withdrawn and reassigned to productive validators.

Owner intervention can deactivate the validator and recover the principal. Until then, users lose the yield that the retained allocation would otherwise earn.

## — RECOMMENDATION

Check native active-set membership during every validator roll, independently of whether other precompile calls succeed. Advance the two-strike flags at most once per Monad epoch using the phase-appropriate snapshot or consensus view. Treat staking-operation failures only as additional evidence of validator removal.

## — RESOLUTION

Status: Resolved.

**M-30** Reward Claims Mask Validator Exit

CONTINUED

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/840>

LOW **L-01**

ACKNOWLEDGED

# Frozen vault allows stale priced atomic exits

CATEGORY Logical Error  
LOCATION `_handleStakeBalanceVerification()`  
REVIEW Main Review 2

## DESCRIPTION

This issue could cause unfair exits during an emergency

The vault's share price depends on how much MON the system thinks it has. Some MON is liquid in the contract, and some is staked with validators

If ShMonad sees that a validator has less stake than expected, the code treats that as an emergency. In `_handleStakeBalanceVerification`, it detects

```
``plain text
expected validator stake > actual validator stake
``
```

Then it sets the protocol to frozen and closed, but it does not reduce `s_globalCapital.stakedAmount` to remove the missing MON. In `StakeTracker.sol`, line 2589, it emits `UnexpectedPrincipalStakeDeficit`, sets `frozen = true` and `closed = true`, then returns.

The problem is that instant exits still work. `withdraw()` and `redeem()` have `nonReentrant` and `cbPooledShareRate`, but not `notWhenFrozen` or `notWhenClosed`

The docs also confirm atomic withdraw/redeem has "no gate" for frozen or closed state.

If 100 MON is missing, everyone's shMON should become cheaper, or all exits should stop until the loss is handled. Instead, the vault still prices shares as if the missing 100 MON exists.

Fast users can redeem using the instant-liquidity pool and receive real MON at the old, too-high price, assume

```
``plain text
Vault says it has: 1,000 MON
Actually has: 900 MON
shMON supply: 1,000 shMON
Alice owns: 100 shMON
``
```

**L-01** Frozen vault allows stale priced atomic exits**CONTINUED**

Alice's fair value after the loss is 90 MON. But because accounting still says 1,000 MON exists, Alice can redeem close to 100 MON from atomic liquidity. She escapes her 10 MON share of the loss. The remaining users now absorb more of the missing stake

this needs a real principal deficit. The project currently assumes Monad staking does not slash principal, and the docs say ShMonad does not support slashing. So this is not normal to happen unless that deficit state actually happens. and the emergency branch is reached with a real missing stake event, the code path is live

**— RECOMMENDATION**

Add notWhenClosed to all atomic-exit paths: withdraw(), redeem(), and agentWithdrawFromCommitted()

**— RESOLUTION**

Status: Acknowledged.

LOW **L-02**

RESOLVED

# Revoked validator auth still settles rewards

|          |                   |
|----------|-------------------|
| CATEGORY | Validation        |
| LOCATION | onlyAuthAddress() |
| REVIEW   | Main Review 2     |

## — DESCRIPTION

After a validator changes its auth key, the old key is still treated as the boss by the Coinbase contract until someone manually updates the Coinbase cache. During that gap, the old key can change reward settings and route money to itself

There are two places that know "who controls this validator"

1. Monad staking precompile: the real/current validator auth address.
2. ShMonad Coinbase contract: a cached copy of that auth address.

The Coinbase contract trusts its cached copy, not the live staking precompile value  
onlyAuthAddress checks only

```
``plain text
msg.sender == s_config.authAddress
``
```

So if `s_config.authAddress` still equals `oldAuth`, then `oldAuth` still has power  
`updateAuthAddress()` can refresh the cache, but it is manual, not automatic. The code explicitly reads the new auth from the staking precompile, then stores it, but only when someone calls this function

Until then, old auth can still call the protected setters:

```
updatePriorityCommissionRate
updateMEVCommissionRate
updateShMonadDonationRate
updateCommissionRecipient
```

Those are all protected by the stale cached auth check, not by the live precompile auth:  
Validator key rotation is supposed to revoke the old key. If the old key was compromised, rotating to `newAuth` should stop `oldAuth` from controlling reward routing

Here, rotation does not fully revoke `oldAuth`. It only changes the staking precompile. The Coinbase system keeps trusting `oldAuth` until a separate sync transaction happens.  
`updateAuthAddress()` helps only if the new auth calls it before the old auth abuses the stale cache

**L-02** Revoked validator auth still settles rewards

CONTINUED

**— RECOMMENDATION**

Authorize only against the live staking precompile, replace onlyAuthAddress with (uint64 id,) = \_immutableIdentity(); (address liveAuth,) = \_getValidator(id); require(msg.sender == liveAuth, OnlyAuthAddress());  
and in ValidatorRegistry.processCoinbaseByAuth(uint64) read liveAuth from STAKING.getValidator(validatorId) instead of ICoinbase(coinbase).AUTH\_ADDRESS();

**— RESOLUTION**

Status: Resolved.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/824>

LOW **L-03**

ACKNOWLEDGED

# Public isolated claim can strand principal

|          |                                            |
|----------|--------------------------------------------|
| CATEGORY | Logical Error                              |
| LOCATION | <code>claimDeactivatedIsolatedFor()</code> |
| REVIEW   | Main Review 2                              |

## — DESCRIPTION

Isolated vShares are like a receipt saying “this account can later withdraw MON from validator X.” The safe owner path lets the receipt owner choose where the MON goes. The permissionless path lets any stranger cash someone else’s receipt, but forces the MON to be sent to the receipt owner address itself. That is fine for normal wallets. It is dangerous for contracts.

The difference is here:

Owner path: `claimDeactivatedIsolated(receiver)` lets the owner choose receiver.

Public “for” path: `claimDeactivatedIsolatedFor(account)` forces `receiver = account`. `depositIsolated` can mint isolated vShares to any receiver, including a smart contract. There is no EOA-only check and no must-be-payable check. The shares are stored directly in `s_isolatedVShareBalances[receiver][validatorId]`, so the contract can be the real holder.

The bad flow is:

1. A contract holds isolated vShares.
2. The validator is deactivated and all isolated stake becomes idle.
3. The contract owner wants to call the safe claim and send MON to a treasury.
4. An attacker calls `claimDeactivatedIsolatedFor` first.
5. ShMonad burns the contract’s vShares.
6. ShMonad force-sends MON to the contract itself.
7. If the contract has no native-MON sweep function, the MON is stuck there.
8. The contract cannot retry the safe claim because the vShares are already gone.

The burn happens before settlement in `_claimDeactivatedIsolated`,

The forced payout happens through `_sendIsolatedSettlement`, where the permissionless path uses `forceSafeTransferETH`.

This permissionless function exists for liveness. The codebase wants abandoned isolated positions to be clearable so a deactivated validator can eventually be reused.

**L-03** Public isolated claim can strand principal

CONTINUED

The issue is that it treats every contract holder as if it is abandoned, with no approval, no signature, no operator permission, and no chance for the real holder to choose the final receiver

the attacker can turn a recoverable claim into stuck funds. For EOAs or contracts with a sweep function, impact is mostly unwanted timing and forced custody. For contracts that cannot move raw native MON, this can freeze up to the full isolated position

**— RECOMMENDATION**

Replace `claimDeactivatedIsolatedFor` forced payout with a pull payment, credit `s_deferredIsolatedClaims\[account\] += amountMon` while retaining that amount in `s_isolatedTotalAssets / s_isolatedLiquidAssets`, Then add a caller only withdrawal function that zeroes the credit, decrements those totals, and `safeTransferETH` to the owner chosen receiver

**— RESOLUTION**

Status: Acknowledged.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/823>

LOW **L-04**

RESOLVED

# Raw signatures bypass ERC1271 permit checks

|          |               |
|----------|---------------|
| CATEGORY | Validation    |
| LOCATION | permit()      |
| REVIEW   | Main Review 2 |

## — DESCRIPTION

permit() is a signed approval. It lets someone set an allowance without the token owner sending a transaction. After allowance is set, the spender can call transferFrom() or redeem() and move the owner's tokens

The issue is in the order of checks. In both shMON and hMON, the code first checks: "does this normal ECDSA signature recover to owner?" If yes, it immediately approves. It only asks the smart-wallet policy through ERC-1271 if ECDSA fails.

That is wrong for an account that has code, such as an EIP-7702 delegated EOA or a smart-account-style owner. For those accounts, the account contract's isValidSignature() policy should decide whether a signature is valid.

That policy may require multisig, guardian approval, spending limits, or may reject an old key. This code lets a raw old / private key signature skip that policy completely

An attacker who gets a valid permit signature from the underlying EOA key can create allowance even if the live smart account would reject it. Then

For shMON, they can steal uncommitted shMON with transferFrom(), or use redeem() because \_withdraw() spends allowance and burns the victim's share

## — RECOMMENDATION

Replace the ECDSA-first branch in both FLERC20.permit() and HolisticMON.permit() with  
require(SignatureChecker.isValidSignatureNow(owner, hash, abi.encodePacked(r, s, v)),  
ERC2612InvalidSigner(address(0), owner));

So EOAs use ECDSA while every code-bearing owner must pass its ERC-1271 policy.

## — RESOLUTION

Status: Resolved.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/822>

LOW L-05

RESOLVED

# Zero snapshots credit attacker isolated stake

CATEGORY Logical Error  
 LOCATION `_splitIsolatedAndPooledBySnapshot()`  
 REVIEW Main Review 2

## DESCRIPTION

ShMonad has two groups, `\pooled\` users: normal shMON holders, `\isolated\` users: users who stake on one specific validator and get validator-specific `\vShares\`.

When a validator reward arrives, ShMonad must decide: "how much belongs to pooled users, and how much belongs to isolated users?" It is supposed to use an old snapshot for that, so a new depositor cannot jump in at the last second and claim old rewards

The bug is that a real snapshot can be:

```
``plain text
pooled snapshot = 0
isolated snapshot = 0
`[
```

That can simply mean: "at snapshot time, nobody from ShMonad had stake here."

But the code treats `\(0,0)\` as "snapshot missing" and falls back to live state

So if an attacker deposits isolated stake after the snapshot, their new `\pendingStake\` is used to split an older reward, assume

1. Validator had old reward waiting.
2. At the real snapshot, isolated stake was zero.
3. Attacker deposits the minimum isolated amount.
4. Their deposit creates `vShares` and increases `\pendingStake\`: `StakeTracker.sol`.
5. Reward is processed through `\sendValidatorRewards\`, `\boostYield\`, or coinbase donation settlement.
6. The splitter now says: "pooled stake is zero, isolated stake is attacker's new deposit."
7. Result: the whole reward is credited to attacker-owned isolated equity

Once credited, `\_creditIsolatedRewards()\` explicitly makes the value current isolated equity owned by current `vShare` holders

The attacker did not own isolated stake when the reward was earned or when the snapshot was taken. The protocol is supposed to use the snapshot to prevent this exact last-second

**L-05** Zero snapshots credit attacker isolated stake

CONTINUED

capture. Instead, because zero means both "valid empty snapshot" and "missing snapshot," the code gives old rewards to a new depositor.

A small isolated deposit can capture a much larger reward. If pooled fallback is zero, the attacker can get essentially 100% of the validator-attributed reward portion. Then `\requestUnstakeIsolated()\` prices their vShares against the enlarged isolated equity and locks a MON claim

This specific bug is for non-staking validator-attributed rewards using `_splitIsolatedAndPooledBySnapshot()`: `sendValidatorRewards`, `native boostYield`, and `coinbase donation settlement`. Normal staking yield settlement uses a different productive snapshot splitter

**— RECOMMENDATION**

TBD

**— RESOLUTION**

Status: Resolved.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/833>

LOW **L-06**

RESOLVED

# Utilized atomic pool prevents target shutdown

CATEGORY Logical Error  
 LOCATION `_checkSetNewAtomicLiquidityTarget()`  
 REVIEW Main Review 2

## DESCRIPTION

The atomic pool is like a cash drawer used for instant shMON -\> MON exits

``plain text

`allocatedAmount` = how big the cash drawer is supposed to be.

`distributedAmount` = how much of that drawer was already used.

available liquidity is roughly `allocatedAmount - distributedAmount`

`

The owner can request a lower target with `setPoolTargetLiquidityPercentage()`; that only stores a pending target first\|

The wrong part is in `_checkSetNewAtomicLiquidityTarget()`. If the owner asks to reduce the drawer below the amount already used, the code refuses and replaces the owner's requested lower amount with `\ oldUtilizedAmount\`

Then it writes that temporary value back as the active target, assume

- Atomic target is 100 MON.
- Users have already instantly withdrawn 100 MON.
- So the pool is fully used: 100 allocated, 100 distributed, 0 available.
- Owner says: "set target to 0."
- Crank tries to apply 0, but sees 0 is below the used amount 100.
- So the code changes the active target back to 100.
- Result: target-zero did not actually disable the atomic pool

The next code already scales utilization proportionally when changing the allocation. So the contract could reduce both accounting numbers together without creating fake withdrawable MON.

The bug is that it treats "already used liquidity" as a reason to keep the target alive.

A user can trigger this because normal `\ withdraw()\` uses the atomic pool

Withdraw accounting increases `\ distributedAmount\`, So a holder can consume available atomic liquidity, making the pool fully utilized before the owner's reduction is cranked

The impact is that governance cannot reliably shut down or reduce atomic liquidity. New deposits and returned validator liquidity can keep refilling the atomic pool because

**L-06** Utilized atomic pool prevents target shutdown

CONTINUED

`\_subtractNetToAtomicLiquidity()\` uses the active target percentage:

An attacker can then instantly withdraw that newly reopened liquidity again, keeping the pool pinned

**— RECOMMENDATION**

we need to not floor the new allocation at `oldUtilizedAmount`. Apply the requested allocation and let

`_settleGlobalNetMONAgainstAtomicUnstaking()` proportionally scale utilization

At 100% utilization, reducing both `allocatedAmount` and `distributedAmount` by the same amount generates zero net cash flow, so this safely retires the accounting target without creating liquidity.

**— RESOLUTION**

Status: Resolved.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/843>

LOW L-07

ACKNOWLEDGED

## Small slope collapses atomic withdrawal fees

CATEGORY Math  
LOCATION `_linearFeeAtGross()`  
REVIEW Main Review 2

### DESCRIPTION

ShMonad has two ways to unstake: wait through traditional unstaking, or use instant unstaking from the AtomicUnstakePool. Instant unstaking is supposed to charge a fee because it uses scarce liquid MON immediately

The docs describe this as shMON  $\rightarrow$  MON right away, with a dynamic fee based on pool utilization

The bug is in the math used to calculate that fee when the owner sets a very tiny nonzero slope, like  $mRay = 1$ , with a real base fee like  $cRay = 1\%$

if the base fee is 1%, withdrawing 1000 MON instantly should cost about 10 MON. but because `mRay` is nonzero, the code does not use the simple constant-rate branch. It uses the quadratic solver in FeeLib, and that solver subtracts two almost-equal huge numbers here

That subtraction loses precision. The solver then checks only two tiny nearby candidates and, if both are wrong, returns the lower fee result

This is wrong because the configured fee curve still has a 1% base fee. A tiny slope should behave almost like a constant 1% fee, not like almost free

The bad path is

`setUnstakeFeeCurve()` accepts  $mRay = 1$  because it only checks slope  $\leq RAY$ , intercept  $\leq RAY$ , and slope + intercept  $\leq RAY$

`maxWithdraw()` uses the capped fee calculation to decide how much can be withdrawn

`withdraw()` then trusts that quote, burns shares, and sends MO

`_accountForWithdraw()` records exactly the returned netAmount and fee, so it does not later fix the missing fee

If governance / owner sets this bad curve, any normal user with enough shMON can instantly withdraw the pool's available MON while paying almost no fee. In the example, they receive `1000 MON` and pay only 1 gwei fee instead of about 10 MON

That skipped fee would normally stay in the system as value for the pool / protocol / remaining holders and as a price signal that the instant pool is being depleted.

So the attacker is not creating MON from nowhere, but they are bypassing the intended scarcity fee and can drain the instant liquidity much more cheaply than the fee curve says they should.

**L-07** Small slope collapses atomic withdrawal fees

CONTINUED

Note, the default slope is not this tiny, so the exact example is not active under default settings. The bug becomes exploitable after an owner / governance fee-curve update into the accepted but unsafe parameter range

**— RECOMMENDATION**

we need to avoid subtracting nearly equal roots. A rationalized expression can be used as the initial estimate

```
`plain text
// Instead of:
// net0 = (sqrt(D) - A) * L / m;
// Algebraically equivalent continuous form without catastrophic cancellation:
net0 = Math.fullMulDiv(gross, 2 * RAY, A + sqrtD);
`
```

That estimate must then be corrected against the actual integer fee function. The solver should not merely inspect  $d_0$  and  $d_0 + 1$  and return the lower fee when both fail. It should perform a bounded monotonic search or otherwise prove

```
`plain text
net(gross - 1) < targetNet <= net(gross)
`
```

**— RESOLUTION**

Status: Acknowledged.

LOW **L-08**

RESOLVED

# Queue planner preserves unrelated unstake demand

|          |                         |
|----------|-------------------------|
| CATEGORY | Logical Error           |
| LOCATION | planExcessQueueOffset() |
| REVIEW   | Main Review 2           |

## DESCRIPTION

ShMonad has two big to do lists for validator stake

queueToStake: MON should be put into validators.

queueForUnstake: MON should be pulled out of validators

If both lists contain the same unrelated amount, they should usually cancel. Example: "stake 1,000 MON" and "unstake 1,000 MON" means the system should do nothing, not move validators around

The bug is that when ShMonad is short on redemption reserves by even a tiny amount,

QueueLib.planExcessQueueOffset() becomes too conservative,

If reserves are still short after the reserve fill, it only allows remaining netting against

turnoverQueueForUnstake. If turnover is zero, it refuses to cancel the big stake/unstake pair

That is wrong because non-turnover unstake is not only user redemption demand. The code itself treats non-turnover unstake as user / liability / pool-replenishment withdrawal demand

So the planner is protecting too much. It should protect only the tiny uncovered redemption amount, not the whole unrelated atomic-pool queue, example

1. boostYield() records real revenue and adds it to queueToStake

2. Atomic withdraw() burns shares, sends MON out, and calls \_accountForWithdraw()

3. If the atomic pool needs backfill, \_accountForWithdraw() adds to queueForUnstake, but it does not reduce the boost-created queueToStake

4. A tiny traditional unstake adds the dust redemption liability and another tiny queueForUnstake

Then global crank runs. \_offsetExcessQueueCapacityWithNet() calls the bad planner and writes the reduced queues back, after that, \_clampQueuesToCapacityOrRoll() rolls the stake side forward if there is no liquid MON,

but leaves the unstake side executable if validators have stake available

So the protocol ends up doing this

```plain text

L-08 Queue planner preserves unrelated unstake demand

CONTINUED

Should happen:

stake 1000 and unstake 1000 cancel

only 1 wei redemption remains

Actually happens:

stake 1000 is rolled to next epoch

unstake 1000 runs now

validators undelegate 1000

...

This can force a large amount of validator stake to exit for no real reason. That means lost staking yield, atomic liquidity drying up, users waiting on the withdrawal delay, validators possibly falling below useful stake, and the protocol repeatedly doing expensive unstake / restake churn

— RECOMMENDATION

The planner must preserve only the amount of non-turnover queue actually required for uncovered redemptions, rather than protecting the entire non-turnover queue.

— RESOLUTION

Status: Resolved.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/849>

LOW **L-09**

ACKNOWLEDGED

hMON Pending Stake Delays Pooled Exits

| | |
|----------|---|
| CATEGORY | Logical Error |
| LOCATION | https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/9f116de99a16a35f1889e08880e0b50141d28701/src/shmonad/libraries/AccountingLib.sol#L286 |
| REVIEW | Main Review 2 |

— DESCRIPTION

depositToZeroYieldTranche() creates a matching totalZeroYieldPayable liability and routes the deposited MON through the shared staking queue. When this backing is delegated, it is recorded in the aggregate s_globalPending.pendingStaking bucket without identifying whether it belongs to pooled users or hMON holders.

Pooled equity already subtracts totalZeroYieldPayable. Therefore, hMON backing is excluded from pooled equity even when it remains included in the contract's staked capital. However, maximumNewGlobalRedemptionAmount() then subtracts all aggregate pendingStaking from that already-adjusted pooled equity.

For pooled active stake P and hMON-backed pending stake H , pooled equity correctly equals $(P + H) - H = P$, but the no-delay redemption capacity becomes $P - H$. The hMON-backed pending delegation is consequently excluded twice.

An hMON depositor can create this state by depositing into the zero-yield tranche before an allocation crank. Once the backing is delegated, a sufficiently large pending amount can make an unrelated pooled unstake request exceed the understated redemption capacity. The request is then assigned the pending-principal deadline, two epochs later than the ordinary deadline.

By the ordinary completion epoch, reserves can already fully cover the pooled redemption. However, completeUnstake() continues reverting solely because the request was assigned the additional two-epoch delay. The hMON depositor retains a fully backed hMON claim throughout this process.

The same mixed state can arise organically from hMON deposits and zero-yield commission flows. The impact is a permissionless but capital-intensive delay of pooled withdrawals rather than loss of funds.

— RECOMMENDATION

Track pooled and zero-yield staking queues and pending delegation separately. Subtract only pooled-backed pending stake when calculating pooled redemption capacity. Do not infer provenance using the current zero-yield liability because older hMON backing may already be active while unrelated pooled capital is pending.

L-09 hMON Pending Stake Delays Pooled Exits

CONTINUED

— **RESOLUTION**

Status: Acknowledged.

LOW **L-10**

RESOLVED

Closed Vault Advertises Deposits

| | |
|----------|---|
| CATEGORY | Unexpected Behavior |
| LOCATION | https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/9f116de99a16a35f1889e08880e0b50141d28701/src/shmonad/FLERC4626.sol#L465 |
| REVIEW | Main Review 2 |

— DESCRIPTION

deposit() and mint() are disabled whenever the current global epoch is closed through the notWhenClosed modifier. However, maxDeposit() and maxMint() continue returning type(uint128).max regardless of the closed status.

A closure can be triggered manually by governance or automatically following a principal or unstake deficit. The closed state is carried across global epoch rolls until governance explicitly reopens the protocol.

Throughout this state, the vault advertises enormous deposit and mint capacity even though every corresponding transaction deterministically reverts with NotWhenClosed.

ERC-4626 requires maxDeposit() and maxMint() to return no more than the amount their corresponding operation would currently accept. It specifically requires them to return zero when deposits or mints are disabled, including temporarily.

Compatible frontends can use these functions to determine whether ShMonad accepts funds and to size transactions or batched operations. During an emergency closure, they receive a false capacity signal and may construct transactions that inevitably revert.

This can break the advertised limit-discovery interface precisely during the protocol's emergency state.

— RECOMMENDATION

Return zero from maxDeposit() and maxMint() whenever globalEpochPtr_N(0).closed is true. Otherwise, return the applicable deposit or mint capacity.

— RESOLUTION

Status: Resolved.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/846>

LOW **L-11**

RESOLVED

Deactivated Validator Duplicates Stake Queue

| | |
|----------|---|
| CATEGORY | Logical Error |
| LOCATION | https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/9f116de99a16a35f1889e08880e0b50141d28701/src/shmonad/StakeTracker.sol#L707 |
| REVIEW | Main Review 2 |

— DESCRIPTION

`_recomputeCurrentAllocationTotals()` excludes deactivated validators from `totalAdjustedWeightedRevenue`, but the later validator pass still calls `_computeStakeDelta()` for every linked validator.

A deactivated validator can therefore calculate a new staking allocation from its historical revenue using an active-only denominator. Active validators already partition and consume the entire prior `queueToStake`.

When the deactivated validator reaches the forced-exit branch, its separately calculated `stakeAllocationIncrease` is requeued as carryover.

For a staking queue Q , active-validator weight A , and deactivated-validator historical weight D , active validators consume Q while the deactivated validator requeues an additional $Q \times D / A$. The resulting `queueToStake` is not backed by remaining pooled liquid MON.

The phantom queue can persist across allocation rounds or net against future unstake demand, suppressing legitimate atomic-liquidity replenishment or validator rebalancing.

— RECOMMENDATION

Do not calculate or requeue `stakeAllocationIncrease` for inactive validators excluded from the allocation denominator. Process only their forced withdrawal; the returned principal is already queued when the withdrawal completes. Preserve the existing carryover logic for active validators falling below the minimum stake threshold.

— RESOLUTION

Status: Resolved.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/844>

LOW **L-12**

ACKNOWLEDGED

Toggle Reprices Historical Yield

| | |
|----------|---|
| CATEGORY | Unexpected Behavior |
| LOCATION | https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/9f116de99a16a35f1889e08880e0b50141d28701/src/shmonad/ValidatorRegistry.sol#L225 |
| REVIEW | Main Review 2 |

— DESCRIPTION

updateIncludelIsolatedRewardsInStakeAllocation() immediately changes the stake basis used to score existing allocation-revenue history.

Revenue recorded while the setting is enabled may include isolated yield. If the setting is later disabled, that historical yield is evaluated using pooled-only stake, potentially overstating validator performance.

Conversely, enabling the setting can evaluate pooled-only historical yield against pooled-plus-isolated stake and suppress the validator's score.

The idle-round check prevents changes during an active allocation pass but does not prevent the two historical windows from being interpreted under a different mode. The distortion persists until the old buckets expire.

— RECOMMENDATION

Store pooled and isolated revenue separately and apply one consistent mode when scoring. Alternatively, ignore historical allocation signals until two clean windows have accumulated after a toggle change.

— RESOLUTION

Status: Acknowledged.

LOW **L-13**

ACKNOWLEDGED

Pending Exit Inflates Yield Signal

| | |
|----------|---|
| CATEGORY | Rewards |
| LOCATION | https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/9f116de99a16a35f1889e08880e0b50141d28701/src/shmonad/StakeTracker.sol#L540 |
| REVIEW | Main Review 2 |

— DESCRIPTION

When isolated reward allocation is enabled, isolated stake moved into pendingUnstake is excluded from snapshotIsolatedStake.

Monad withdrawal requests continue earning rewards until deactivation and return principal plus yield. ShMonad records this surplus as validator stakingYield, but the deadband evaluates it using only the remaining pooled and isolated snapshot stake.

A large isolated exit can therefore make ordinary stake-proportional yield appear to be validator outperformance. This gives the validator additional weight when routing later pooled deposits. A partial exit can retain the minimum isolated position while excluding most of the reward-producing capital from the deadband basis.

The same mismatch affects pooled exits even when isolated reward allocation is disabled: targetStakeAmount decreases immediately when pooled unstaking begins, while the outgoing principal remains reward-bearing, allowing its historical yield to be evaluated against the reduced post-exit target.

— RECOMMENDATION

Track the stake basis associated with each staking-yield observation window and use that historical basis when applying the deadband. Isolated principal that earns rewards through undelegation must remain in the applicable basis for as long as those rewards remain in the allocation signal's smoothing windows. Do not rely solely on live pendingUnstake, since it is cleared before withdrawal surplus is recorded. Alternatively, exclude isolated withdrawal surplus from the allocation signal.

— RESOLUTION

Status: Acknowledged.

LOW **L-14**

RESOLVED

Early Activation Stales hMON History

| | |
|----------|---|
| CATEGORY | Events |
| LOCATION | https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/9f116de99a16a35f1889e08880e0b50141d28701/src/shmonad/HolisticMON.sol#L73 |
| REVIEW | Main Review 2 |

— DESCRIPTION

HolisticMON is deployed and initialized before the timelocked ShMonad v1.2 upgrade, making `activate()` publicly callable while the live ShMonad proxy still runs v1.1.

Any account can activate a legacy holder during this pre-upgrade window. `_activateIfNeeded()` immediately marks the holder as activated and emits a one-time mint-style Transfer using the holder's current zero-yield balance.

The holder's balance can then change before the upgrade through existing zero-yield deposits, conversions, or ordinary owner-commission accrual. Because v1.1 is not bound to the new hMON token, those changes emit no corresponding hMON Transfer events.

After the upgrade, the final holder manifest includes the holder and its correct live balance. However, the corresponding `activate()` call silently returns because the activation flag was already set. The pre-upgrade balance delta is therefore not surfaced through hMON events.

For example, a holder activated with a zero balance can receive a legacy deposit before the upgrade. Its canonical hMON balance is correct after the upgrade, but no mint-style Transfer records that balance.

Similarly, activating a funded holder before a legacy conversion can leave the previously emitted activation amount overstating its later balance.

Batch activation allows multiple known holders to be affected during the mandatory three-day upgrade delay. Wallets, explorers, or indexers relying exclusively on hMON Transfer events may therefore display stale balances until they explicitly resynchronize from `balanceOf()`. Canonical ShMonad balances, supply, and protocol accounting remain unaffected.

— RECOMMENDATION

Disable activation until ShMonad has installed the exact hMON token, preferably through an activation flag enabled only by the bound ShMonad during its initializer. Alternatively, checkpoint each holder's previously surfaced balance and emit the missing mint or burn delta instead of relying on a terminal boolean.

L-14 Early Activation States hMON HistoryCONTINUED

— RESOLUTION

Status: Resolved.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/847>

LOW **L-15**

ACKNOWLEDGED

Fail stop state lacks recovery transition

CATEGORY Logical Error
LOCATION `_handleStakeBalanceVerification()`
REVIEW Main Review 2

— DESCRIPTION

This issue happens when ShMonad internal book says `validator V should have 100 MON` but the real Monad staking precompile says `validator V only has 99 MON`. That missing 1 MON is a principal deficit. Freezing the system when this happens is reasonable, but ShMonad freezes without fixing the accounting. In `StakeTracker.sol`, the code sees the missing stake, emits `UnexpectedPrincipalStakeDeficit`, then sets `frozen = true` and `closed = true`, but does not reduce the validator expected stake or global staked amount. Then it clears `wasCranked`, meaning the same validator must be tried again later, the validator loop then stops before moving the cursor forward, so the system bookmark remains stuck on the same broken validator. The owner can call `setFrozenStatus(false)`, but that only changes the flag, it does not fix the missing MON. So the next `crank()` checks the same validator, sees the same deficit, and freezes again. Also, the normal admin actions like `deactivateValidator()` and `addValidator()` require the validator round to be idle, but the round is not idle, because the cursor is stuck on the bad validator.

— RECOMMENDATION

Consider implementing an explicit `FrozenDeficit` record and a dedicated `reconcileFrozenValidator()` recovery path that fully reconciles validator losses before advancing state or allowing unfreeze.

— RESOLUTION

Status: Acknowledged.

LOW **L-16**

ACKNOWLEDGED

Late rewards buy closed allocation rounds

CATEGORY Validation
 LOCATION `_handleValidatorRewards()`
 REVIEW Main Review 2

— DESCRIPTION

The protocol decides how much pooled stake each validator should get. that decision is partly based on validator revenue signal, meaning validators that brought in more useful revenue can receive more of the next queued stake

The bug is a timing problem between two clocks

- Monad's real chain epoch has already moved to the next epoch.

- ShMonad's internal epoch has not moved yet, because nobody has called `\crank()\`

In that small gap, `sendValidatorRewards()` is still open. It does not check "has Monad already entered a new epoch and is ShMonad waiting to roll?" In FLERC4626.sol

So the attacker can do

```
``plain text
```

```
sendValidatorRewards()
```

```
crank()
```

```
`
```

The reward is written into ShMonad current epoch bucket, `N(0)`, in `_handleValidatorRewards()`,

then `crank()` runs `_crankGlobal()`,

which first uses that same bucket for allocation math, and only afterward increments

```
\internalEpoch\
```

That increment is the key: the same storage bucket that was `N(0)` becomes `N(-1)` after the roll,

then validator cranking treats it as an old reward and can settle it immediately

This is wrong because after an epoch is already over, nobody should be able to add a new performance score into that finished round. It is like letting someone place a bid after an auction has closed,

The impact is mainly two things

1. A validator can wait until the queued stake is visible, then buy enough reward signal to receive an unfairly large share of that queued stake. The allocation formula really does split `\queueToStake\`` by weighted revenue share

2. A reward that was supposed to wait for the next settlement window can become payable in the same raw Monad epoch, so the wrong current validator / delegator cohort can benefit

There is a helper view that can detect this pending roll state, `isGlobalCrankAvailable()`, but

`sendValidatorRewards()` does not use it

L-16 Late rewards buy closed allocation rounds

CONTINUED

— RECOMMENDATION

Consider rejecting reward submissions whenever Monad has entered a stable new epoch but ShMonad has not yet rolled

— RESOLUTION

Status: Acknowledged.

LOW **L-17**

PARTIALLY RESOLVED

ShMonad strikes ignore native epoch identity

| | |
|----------|---------------------------------------|
| CATEGORY | Logical Error |
| LOCATION | <code>_advanceActiveSetFlags()</code> |
| REVIEW | Main Review 2 |

— DESCRIPTION

In `crank()`, It updates global epoch accounting, then walks through validators one by one. because there can be many validators, it keeps a cursor: if one transaction cannot finish all validators, the next transaction resumes from the next validator

The intended safety rule is, do not deactivate a validator after one bad observation. A validator should be considered inactive only after two misses

But the code does not prove those two misses came from two different Monad epochs. It only knows that the validator was cranked twice, example

```
```plain text
```

Validator starts healthy:

Last = true

Current = true

Same Monad epoch has one temporary bad state

First crank sees bad state:

Last = true

Current = false

Cursor catches up, global crank resets the validator loop.

Second crank happens in the same Monad epoch and sees the same bad state:

Last = false

Current = false

Code says: two misses, deactivate validator

```
```
```

If the system wanted "two separate chances across time," it accidentally accepts "same bad moment counted twice."

So a validator that only had one temporary problem can be treated as permanently bad. ShMonad then starts unwinding stake from it, If there is one real temporary false observation, the cursor timing can turn that one miss into two strikes

L-17 ShMonad strikes ignore native epoch identity

CONTINUED**— RECOMMENDATION**

If this is not intended, active set strikes must be keyed to the epoch represented by the staking precompile observation, not to the number of times `_crankValidator()` has executed

— RESOLUTION

Status: Partially resolved.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/840>

LOW **L-18**

RESOLVED

Isolated withdrawals can strand ownerless MON

CATEGORY Logical Error
LOCATION `_finalizeCompletedIsolatedWithdrawal()`
REVIEW Main Review 2

— DESCRIPTION

The isolated side tracks buckets like `pendingUnstake`, `idleAssets`, and `redemptionsPayable`. When the last isolated user exits, their vShares are burned and ShMonad records: we owe this user MON, that is `redemptionsPayable`. But the actual validator principal may still be waiting inside Monad native withdrawal system as `pendingUnstake`. If isolated rewards or replacement liquidity arrive before that native withdrawal finishes, ShMonad can pay the exiting user from that replacement liquidity via `completeUnstakeIsolatedFor()`. At that moment no isolated shares remain. no redemption request remains, but the old principal is still pending withdrawal. The sweep function exists exactly to clean ownerless isolated idle money. Its own comment says that if isolated owners are gone, the idle MON must rejoin pooled equity, otherwise value is stranded. The problem is that when the original native withdrawal finally completes, `_finalizeCompletedIsolatedWithdrawal()` adds the returned principal into `idleAssets`, but does not call the sweep. At that point the isolated bucket has money, but nobody owns it, no vShares, no redemption request, no pending withdrawal, only `idleAssets`. That should be swept into pooled shMON accounting, but it stays marked as isolated. The normal cleanup paths do not save it. `completeUnstakeIsolated` cannot run because the request is gone. `claimDeactivatedIsolated` cannot run because there are no shares. `depositIsolated\`` can sweep, but only for active validators. validator removal ignores `idleAssets` as pending crank work. readding the validator is blocked because `idleAssets` still counts as residual isolated state. Impact is that the MON is still physically inside the contract, but accounting says it belongs to isolated users who no longer exist.

L-18 Isolated withdrawals can strand ownerless MON

CONTINUED

That means, the MON cannot be claimed by any user, pooled shMON holders do not get it either pooled totalAssets() stays too low because isolated assets are subtracted from pooled equity, the validator ID can become impossible to reuse

If the pending isolated withdrawal was 1,000 MON, the stranded amount can be 1,000 MON

— RECOMMENDATION

Consider running orphan reconciliation inside the finalizer, after principal and surplus accounting

— RESOLUTION

Status: Resolved.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/850>

LOW **L-19**

ACKNOWLEDGED

Cursor lag prematurely finalizes pending deposit

CATEGORY Logical Error
LOCATION `_settlePastEpochEdges()`
REVIEW Main Review 2

— DESCRIPTION

ShMonad can mark a validator deposit as finished staking before Monad staking precompile has actually activated it

Think of ShMonad as keeping its own notebook, this MON is still waiting to become staked, this MON is already productive validator stake, this MON can be safely counted for unstaking/reward math.

The bug is that ShMonad can move MON from waiting to already staked too early.

`crank()` is resumable. If there are many validators or low gas, ShMonad may process only part of the validator list and save the next validator in `s_nextValidatorToCrank`.

Later, if Monad has already moved to a new epoch, ShMonad can:

1. finish the old validator pass,
2. run `_crankGlobal()`,
3. reset `s_nextValidatorToCrank` to the first validator,
4. immediately process validators again in the same Monad epoch.

The bad part is here: `_settlePastEpochEdges()` clears old deposits based on ShMonad's epoch slots and a rough did we jump epochs? check, not by asking whether Monad has actually activated the deposit.

Then `_handleCompleteIncreasedAllocation()` just clears `hasDeposit` and subtracts from `s_globalPending.pendingStaking`.

Monad deposits do not become productive immediately. The staking precompile schedules them for a future activation epoch. The mock shows this clearly: normal deposits activate at current epoch + 1, boundary-period deposits at current epoch + 2.

So ShMonad is saying: this deposit is active now

while Monad can still be saying: this deposit is pending.

The direct impact is wrong accounting during the pending window:

global pending staking becomes too low, views can show more productive and unstakable stake than really exists;

`requestUnstake()` can quote timing using an overstated redemption capacity;

later stake and unstake planning can be based on stake that Monad still treats as pending

L-19 Cursor lag prematurely finalizes pending deposit

CONTINUED**— RECOMMENDATION**

we need to not infer deposit completion from `s_nextValidatorToCrank`, ShMonad internal epoch advancement, or one global epoch marker.

Each pooled delegation should store its native activation epoch, using the precompile's `deltaEpoch` and `nextDeltaEpoch` information.

— RESOLUTION

Status: Acknowledged.

MEDIUM **M-01**

RESOLVED

Shared Epoch Prematurely Matures Deposits

CATEGORY Rewards

LOCATION <https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/0c8641c61a36236d3fdbb073b66a6df2864d75/src/shmonad/StakeTracker.sol#L1384-L1395>

REVIEW Remediation Review 3

DESCRIPTION

The remediation for the previous M-02: Late Cranks Mis-Split Rewards introduced productive-deposit accounting intended to prevent newly activating pooled stake from sharing rewards prematurely. However, pooled deposit maturity is still inferred from a global native-epoch marker rather than the epoch in which each validator's delegation was actually initiated.

The remediation therefore does not cover validators processed late in a gas-bounded crank round.

crank() persists its validator cursor when it cannot complete the round. A validator reached after a native epoch transition can consequently initiate its pooled delegation in a newer epoch than the round's shared marker. If the unfinished round crosses one native epoch boundary, the next global roll can produce `monadEpochDistance > 1`.

`_settlePastEpochEdges()` interprets this distance as evidence that the validator's $N(-1)$ deposit has completed. It clears `hasDeposit` and adds the deposit to the productive pooled snapshot, even though Monad's staking precompile still reports it as pending.

A validator reward can expose the resulting accounting error in the same crank. If `sendValidatorRewards()` is called after the late delegation but before the next crank, the reward and deposit are rolled into $N(-1)$.

`_settlePastEpochEdges()` first prematurely includes the deposit in productive pooled stake.

`_settleValidatorRewardsPayable()` subsequently sends the payment through Monad's externalReward syscall and immediately calls `_settleEarnedStakingYield()`.

Monad distributes the reward using live consensus stake, under which the new pooled delegation remains pending and earns nothing. ShMonad then divides its returned reward using a productive-stake denominator that already includes that pending deposit.

For example, with 100 MON of productive pooled stake, 100 MON of productive isolated stake, and a 100 MON pending pooled deposit, a 30 MON reward returned to ShMonad should be divided 15 MON to each cohort. The contaminated snapshot instead attributes 20 MON to pooled holders and only 10 MON to isolated holders.

M-01 Shared Epoch Prematurely Matures Deposits

CONTINUED

Only one validator round must remain unfinished across an epoch boundary. Once that state exists, reward submission and crank completion are public operations. The result is an irreversible transfer of validator rewards from isolated owners to pooled holders.

— RECOMMENDATION

Record the native initiation epoch for every pooled delegation and require its activation plus the intended productive reward period to complete before including it in productive pooled stake. Do not infer per-validator maturity from a global marker established before that validator was processed.

— RESOLUTION

Status: Resolved.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/859>

MEDIUM **M-02**

RESOLVED

Validator exit forgets queued withdrawal portion

| | |
|----------|-------------------------------------|
| CATEGORY | Logical Error |
| LOCATION | calculateValidatorEpochStakeDelta() |
| REVIEW | Remediation Review 3 |

— DESCRIPTION

A validator can receive both a stake instruction and a withdrawal instruction in the same round. The code combines them into one net result and forgets the original withdrawal amount. For example: stake 0.9 MON, withdraw 0.8 MON, so the result becomes "stake 0.1 MON." If the validator must then be forcefully removed, the protocol replaces that plan with a full validator exit. However, it saves only the net 0.1 stake result and ignores the gross 0.8 withdrawal. If the forced exit returns only 0.1 MON, the remaining 0.7 MON should be added back to queueForUnstake. The code does not do this because the saved net result says "stake," not "withdraw." Therefore, the remaining 0.7 MON of real withdrawal demand silently disappears from the queue. The protocol still records that it owes users through redemptionsPayable, but stops requesting enough MON from validators. Users may therefore be unable to complete matured withdrawals until unrelated deposits or other inflows cover the missing funds. If the forced exit completely fails, the entire hidden withdrawal amount can be lost from the queue.

— RECOMMENDATION

The forced exit path must preserve the gross values before netting

```
``plain text
gross stake increase
gross withdrawal decrease
gross turnover withdrawal
`
After a successful forced exit
``plain text
```

M-02 Validator exit forgets queued withdrawal portion

CONTINUED

```
fulfilledWithdrawal = min(forcedWithdrawal, grossWithdrawal)
withdrawalToRequeue = grossWithdrawal - fulfilledWithdrawal
...
```

On failure or dust skip, it should restore both the full gross stake instruction and the full gross withdrawal instruction

— RESOLUTION

Status: Resolved.

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/860>

MEDIUM **M-03**

RESOLVED

Mature withdrawals lose priority against atomics

CATEGORY Logical Error
LOCATION _handleCompleteDecreasedAllocation
REVIEW Remediation Review 3

— DESCRIPTION

A validator withdrawal can return more MON than expected; the extra amount which called the surplus. The fix correctly gives this surplus to pooled shMON holders instead of isolated holders. However, it immediately records the surplus as revenue and adds it to queueToStake. It then removes that surplus from the amount processed by the reserve-first logic. Therefore, only the expected principal increases reservedAmount, even when users are still owed MON. Example: the protocol owes 100 MON and receives 90 MON principal plus 10 MON surplus. Only 90 MON becomes reserved, while the 10 MON is treated as stakeable revenue. The traditional user completeUnstake(100) can therefore revert because it sees only 90 MON available. But atomic withdrawal accounting treats the unallocated 10 MON revenue as available liquidity. A newer atomic exiter can withdraw that 10 MON before the older traditional user receives it, the permissionless, gas-limited validator crank lets callers create this ordering between validators. The impact is withdrawal queue-jumping, failed mature withdrawals, and unnecessary user delays.

— RECOMMENDATION

Revenue ownership and physical cash routing handling separately.
we should route the entire received amount through reserves first, then stake only the leftover.

— RESOLUTION

Status: Resolved.

M-03 Mature withdrawals lose priority against atomics

CONTINUED

<https://github.com/FastLane-Labs/fastlane-contracts-internal/pull/853>

MEDIUM **M-04**

Sponsored operation can overcharge bonded stake

| | |
|----------|----------------------|
| CATEGORY | Validation |
| LOCATION | getHash() |
| REVIEW | Remediation Review 3 |

— DESCRIPTION

FastLane have experimental Paymaster that sponsors ERC-4337 gas using bonded shMON. The sponsor signature should state both the approved action and the maximum payable amount. Here, it signs the action and gas price, but not `accountGasLimits`, `preVerificationGas`, or the Paymaster gas limits. `EntryPoint` uses exactly those omitted limits to calculate `maxCost`, the money reserved before execution. A malicious user controlling the sender can obtain a normal sponsor signature, raise the unsigned limits, then re-sign the changed operation themselves. The sponsor signature remains valid, while the Paymaster now locks the much larger `maxCost` from the sponsor shMON. For example, an operation needing little gas can falsely declare a total limit of 10,000,000 gas. After execution, `EntryPoint` gives the Paymaster `actualGasCost`, meaning the real cost incurred so far. But `_postOp` ignores that value and bills `declared maximum gas × actual gas price`. Therefore, even when only a small part of the limit was needed, the sponsor is charged as though almost all of it was used, that bill deducts the sponsor's committed/bonded shares and sends the resulting MON to the Paymaster. So one authorization can severely deplete the sponsor shMON and prevent future sponsored operations.

— RECOMMENDATION

sign every field affecting `maxCost`, then charge using `actualGasCost` plus only tightly bounded `postOp` overhead.

— RESOLUTION

Status: Reported during the fix review period.

LOW **L-01**

ACKNOWLEDGED

Dust Scores Dilute Stake Allocation

| | |
|----------|---|
| CATEGORY | Logical Error |
| LOCATION | https://github.com/GuardianOrg/fastlane-contracts-internal-team1-1777384866352/blob/shmonad-v1.2/src/shmonad/libraries/StakeAllocationLib.sol#L231-L245 |
| REVIEW | Remediation Review 3 |

— DESCRIPTION

The stake-allocation denominator and validator eligibility check use inconsistent dust rules.

`_recomputeCurrentAllocationTotals()` includes every active validator's adjusted weighted revenue in `totalAdjustedWeightedRevenue`.

However, `calculateValidatorEpochStakeDelta()` allocates stake only when the validator's individual score is strictly greater than `DUST_THRESHOLD`

Consequently, scores at or below dust consume denominator share without receiving the corresponding stake. With scores of 2 gwei, 1 gwei, and 1 gwei, the denominator is 4 gwei, but only the first validator is eligible and receives 50% of `queueToStake`.

If every validator has an exact-dust score, the aggregate can pass the global queue guard while every validator receives zero.

The state is permissionlessly reachable through `sendValidatorRewards()`. Under the documented 25% boost commission, a gross 1,333,333,333 wei payment leaves exactly 1 gwei of retained revenue when fully attributed to pooled stake.

With an empty adjacent window and the default 2x multiplier, this produces an exact-dust score.

The unallocated MON remains liquid and is later rediscovered as goodwill, so principal is not lost.

Practical impact is primarily delayed staking yield and potentially unnecessary turnover

Exploitation is most relevant during bootstrap or unusually sparse-revenue periods; normal MON-scale validator revenue makes gwei-sized denominator dilution negligible.

This is a residual of the previous deadband-adjusted allocation finding: its remediation changed the global cutoff to `DUST_THRESHOLD` without aligning denominator membership with validator-level eligibility.

L-01 Dust Scores Dilute Stake Allocation

CONTINUED

— RECOMMENDATION

Define a single eligible allocation weight: adjusted weighted revenue strictly greater than DUST_THRESHOLD, otherwise zero. Use it consistently for the denominator, stake shares, ideal stake, overallocation, and turnover. Roll queueToStake when the eligible denominator is zero. Do not filter totalAllocationSignalStake, as it is required to calculate the underlying scores.

— RESOLUTION

Status: Acknowledged.

MEDIUM **M-01**

ACKNOWLEDGED

Unclaimed priority rewards underprice new deposits

CATEGORY Logical Error
LOCATION `_settleCoinbaseContract()`
REVIEW Remediation Review 4

— DESCRIPTION

There is a short period where the protocol has already earned extra rewards, but the deposit price still acts like the rewards is smaller. during that period, a new depositor can buy slices too cheaply. Later, when the reward is finally counted, that new depositor owns more of the rewards than they should

There are two reward paths, and they are treated differently

1. Normal staking rewards are claimed immediately during validator cranking
2. Direct externalReward fallback payments are also claimed immediately

But coinbase processed priority fee rewards are different. the coinbase contract forwards the priorityStakers portion into Monad staking and ShMonad only books the MON that is sent directly back to it, The forwarded staking reward is not claimed back into ShMonad until a later crank

That matters because deposit pricing uses ShMonad currently tracked equity in

If the reward is sitting inside the staking precompile and has not been claimed yet, deposits do not see it, So

- 1 - Suppose existing holders own 1,000 MON worth of value and 1,000 shares exist
- 2 - A validator has already earned 100 MON more through this coinbase path, but ShMonad has not counted it yet
- 3 - A new user deposits 100 MON
- 4 - Because the system still thinks the vault is worth 1,000 instead of 1,100, that user gets about 100 shares instead of about 90.9 shares
- 5 - Later, when the missing 100 MON reward is finally claimed and counted, that extra 9.1-ish shares turns into free value for the new user
- 6 - That value came from older holders being diluted

— RECOMMENDATION

we recommend acknowledging as a design tradeoff similar to

M-01 Unclaimed priority rewards underprice new deposits

CONTINUED

```
```typescript
```

L-01 Migration retargets pending EOA rewards into the Coinbase JIT window

```
```
```

— RESOLUTION

Status: Acknowledged.

§ 05

Disclaimer.

This report is not, nor should be considered, an "endorsement" or "disapproval" of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Guardian to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. Guardian's position is that each company and individual are responsible for their own due diligence and continuous security. Guardian's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

The assessment services provided by Guardian is subject to dependencies and under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis.

Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.

Notice that smart contracts deployed on the blockchain are not resistant from internal/external exploit. Notice that active smart contract owner privileges constitute an elevated impact to any smart contract's safety and security. Therefore, Guardian does not guarantee the explicit security of the audited smart contract, regardless of the verdict.

§ 06

About Guardian.

Guardian is a smart contract security firm. Every engagement is worked by two independent teams competing on the same scope, each with its own tooling and methods, so no single reviewer's blind spot survives.

\$45B+DIGITAL ASSETS UNDER
REVIEW**200+**

ENGAGEMENTS COMPLETED

500+CRITICAL & HIGH FINDINGS
REPORTED

Reach us at audits@guardianaudits.com.