MANIFEST · SECURITY REVIEW

Manifest
Manifest Hook.

A security review of Manifest Hook.

CLIENT	PROJECT	AUDITORS	AUDIT WINDOW	FINAL REPORT
Manifest	Manifest Hook	Ali Shehab, Osman Ozdemir, Ferran Celades, Vlad Zotov, Michael Lett	August 18 – October 16, 2026	November 26, 2026

Published at guardian.security/portfolio/manifest/hook-2026-11-26

§ 00

Contents.

66 pages: overview, confidence ranking, scope, 40 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. 07

Main Review · 34 findings P. 10

Remediation Review · 6 findings P. 56

§ 05 **Disclaimer** P. 65

§ 06 **About Guardian** P. 66

§ 01

Overview.

Guardian reviewed Manifest Hook across 2 review rounds. The final report includes 7 Medium, 8 Low, 25 Informational findings.

AUDIT FIRM
Guardian

CLIENT
Manifest

PROJECT
Manifest Hook

PREPARED BY
Ali Shehab, Osman Ozdemir, Ferran Celades, Vlad Zotov, Michael Lett

AUDIT WINDOW
August 18 – October 16, 2026

FINAL REPORT
November 26, 2026

REPOSITORY
https://github.com/manifest-finance/ush

METHODOLOGY
Smart contract · Manual · Main Review · Remediation Review

MAIN REVIEW COMMIT
c13c2eae09b2

REMEDIATION REVIEW COMMIT
472479730dc3

1.1 Confidence Ranking

4 / 5

High Confidence

Guardian assigns a Confidence Ranking of 4 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
Critical	**0**	0	0	0	0
High	**0**	0	0	0	0
Medium	**7**	0	3	0	4
Low	**8**	0	3	1	4
Informational	**25**	0	16	0	9
All severities	**40**	**0**	**22**	**1**	**17**

§ 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 ASSIGNED	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	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 Manifest Hook in `https://github.com/manifest-finance/ush`. The scope includes 3,917 source lines across 13 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.

FILE	NSLOC	TOTAL
MAIN REVIEW		
contracts/hooks/interfaces/IManifestHook.sol	97	353
contracts/hooks/lib/ArbExec.sol	167	272
contracts/hooks/lib/DynamicFeeLib.sol	107	195
contracts/hooks/lib/HookMath.sol	59	130
contracts/hooks/lib/PriceMath.sol	21	40
contracts/hooks/lib/SwapEstimation.sol	142	224
contracts/hooks/ManifestHook.sol	849	1,555
contracts/module/base/SafeModuleBase.sol	149	422
contracts/module/interfaces/IPositionFeeModule.sol	18	84
contracts/module/PositionFeeModule.sol	128	300
REMEDIATION REVIEW		
contracts/hooks/interfaces/IManifestHook.sol	114	426
contracts/hooks/lib/ArbExec.sol	181	312
contracts/hooks/lib/BackingCheck.sol	14	33
contracts/hooks/lib/DynamicFeeLib.sol	107	196
contracts/hooks/lib/HookMath.sol	105	210
contracts/hooks/lib/PriceMath.sol	21	40
contracts/hooks/lib/SwapEstimation.sol	348	524
contracts/hooks/ManifestHook.sol	920	1,704
contracts/module/base/SafeModuleBase.sol	165	470
contracts/module/interfaces/IPositionFeeModule.sol	18	84
contracts/module/interfaces/IRewardDistributionModule.sol	6	20
contracts/module/PositionFeeModule.sol	128	311
contracts/module/RewardDistributionModule.sol	53	91

§ 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					
M-01	Flash Accounting Bypasses Arb Caps	Logical Error	MEDIUM	RESOLVED	p. 10
M-02	Terminal Fee Overcharges Entire Sale And Incentivizes Splitting	Math	MEDIUM	ACKNOWLEDGED	p. 11
M-03	Target Updates Enable MEV	MEV	MEDIUM	ACKNOWLEDGED	p. 13
M-04	Universal Router Is Incompatible For Partial Fill	Compatibility	MEDIUM	ACKNOWLEDGED	p. 14
M-05	Target updates can leave `isReverseFlex` stale, letting reverse flex satisfy the below-target add gate	Logical Error	MEDIUM	RESOLVED	p. 16
M-06	Remote withdrawal bridging bypasses execution-time compliance checks	Out of Scope	MEDIUM	RESOLVED	p. 18
M-07	Executor-supplied value irreversibly cancels local composed deposits	Out of Scope	MEDIUM	RESOLVED	p. 19
L-01	`getPendingAuthManager` reports a matured unlock time for a request that does not exist	Unexpected Behavior	LOW	RESOLVED	p. 20
L-02	`usdeUsdTolerance` is decoupled from the ±0.5% `usdeUsdPeg` band, leaving a window where swaps continue against an uncorrectable target price	Logical Error	LOW	RESOLVED	p. 22
L-03	Full-route Profitability Check May Skip Profitable Partial Arbs	Logical Error	LOW	PARTIALLY RESOLVED	p. 24
L-04	Duplicate Queued Additions Can Be Performed Without A Fresh Delay	Logical Error	LOW	RESOLVED	p. 25
L-05	`setDynamicFeeParams` retroactively re-decays stored pressure when `pressureHalfLife` changes	Logical Error	LOW	RESOLVED	p. 26
L-06	quoteFees under-quotes the sell fee for an exact-output sell	Math	LOW	ACKNOWLEDGED	p. 28
I-01	Flex Wallet Can Trigger Repeated USH Minting	Trust Assumptions	INFORMATIONAL	ACKNOWLEDGED	p. 30
I-02	Misleading `onlyPauser` modifier name and `OnlyPauser` error	Best Practices	INFORMATIONAL	RESOLVED	p. 31

Findings continued.

ID	TITLE	CATEGORY	SEVERITY	STATUS	PAGE
I-03	`setAuthManager` silently succeeds without effect when the pending address is re-submitted before maturity	Unexpected Behavior	INFORMATIONAL	RESOLVED	p. 32
I-04	Redundant `onlyPoolManager` modifier on internal hook overrides	Superfluous Code	INFORMATIONAL	RESOLVED	p. 34
I-05	No explicit cap on the dynamic sell fee; callers can only bound it indirectly via output slippage	Informational	INFORMATIONAL	ACKNOWLEDGED	p. 35
I-06	Informational Note Regarding Oracle Latency	Trust Assumption	INFORMATIONAL	ACKNOWLEDGED	p. 37
I-07	Exact-output sells pay a slightly lower dynamic sell surcharge than exact-input	Math	INFORMATIONAL	ACKNOWLEDGED	p. 38
I-08	Misleading comment: "Price exactly at target" branch also runs while price is above target	Documentation	INFORMATIONAL	RESOLVED	p. 40
I-09	Queued Actions Carry Over To A New Owner	Unexpected Behavior	INFORMATIONAL	ACKNOWLEDGED	p. 41
I-10	No Minimum Timelock Delay Check	Configuration	INFORMATIONAL	ACKNOWLEDGED	p. 42
I-11	Timelock execution window is inclusive on both ends, so it spans `TIMELOCK_EXPIRY + 1` second rather than exactly `TIMELOCK_EXPIRY`	Logical Error	INFORMATIONAL	ACKNOWLEDGED	p. 43
I-12	Protocol-minted forward flex is counted in the sell-surcharge denominator, diluting the sell deterrent while the pool rests below target	Business logic errors	INFORMATIONAL	ACKNOWLEDGED	p. 44
I-13	Treasury rotation non-atomically activates replacement capacity for a searcher bracket	state-transition / economic MEV	INFORMATIONAL	ACKNOWLEDGED	p. 46
I-14	Cross-chain deposits settle after global whitelist revocation	Out Of Scope	INFORMATIONAL	RESOLVED	p. 47
I-15	Treasury-clipped downward arbitrage can realize dust losses even when the full re-peg is profitable	Rounding	INFORMATIONAL	ACKNOWLEDGED	p. 48
I-16	Tolerance reductions expose pending repricing surplus to searchers	MEV / economic	INFORMATIONAL	ACKNOWLEDGED	p. 49
I-17	Consider using LiquidityMath.addDelta In SwapEstimation	Math	INFORMATIONAL	RESOLVED	p. 50
I-18	Document AuthManager Address Timelock Behavior	Upgradeability	INFORMATIONAL	ACKNOWLEDGED	p. 51
I-19	NAV updates leave public target and flex views stale until a successful refresh	stale state	INFORMATIONAL	ACKNOWLEDGED	p. 52
I-20	Oracle-driven catch-up mints unbacked rewards from protocol-owned liquidity	accounting / unbacked issuance	INFORMATIONAL	ACKNOWLEDGED	p. 53
I-21	Informational Note for Users Regarding Trust Assumptions	Trust Assumptions	INFORMATIONAL	ACKNOWLEDGED	p. 54

Findings continued.

ID	TITLE	CATEGORY	SEVERITY	STATUS	PAGE
REMEDIATION REVIEW					
L-01	Arb gas budget ignores EIP-150 call retention, so at the minimum `externalGasReserve` swaps from smart-wallet call stacks exceed the EIP-7825 gas limit cap	Denial of Service	LOW	ACKNOWLEDGED	p. 56
L-02	Later swaps in one transaction still run unbudgeted flex walks, so batched swaps exceed the EIP-7825 gas cap at the default `externalGasReserve`	Denial of Service	LOW	ACKNOWLEDGED	p. 58
I-01	The arb's backward scan stops at a checkpoint that fails only on rounding, so a passing route near break-even is skipped or under-sized	Logical Error	INFORMATIONAL	ACKNOWLEDGED	p. 60
I-02	Stale `SafeModuleBase` comment after `RewardDistributionModule` addition	Informational	INFORMATIONAL	RESOLVED	p. 62
I-03	Setup documentation incorrectly requires `REWARDER_ROLE` for reserve-only distributions	Documentation	INFORMATIONAL	RESOLVED	p. 63
I-04	Consider validating that `STAKED_USH.asset()` matches `USH`	Validation	INFORMATIONAL	RESOLVED	p. 64

MEDIUM **M-01** RESOLVED

Flash Accounting Bypasses Arb Caps

CATEGORY	Logical Error
LOCATION	`contracts/hooks/lib/ArbExec.sol:93-111, 163-180; contracts/hooks/ManifestHook.sol:965-970, 1047-1049`
REVIEW	Main Review

DESCRIPTION

The hook caps arbitrage using the PoolManager's physical USH or USDe balance. However, these balances can be temporarily reduced earlier within the same V4 unlock. The trusted Universal Router permits an inner `TAKE` action before the canonical `SWAP`, transferring tokens immediately while leaving a transient debt that is settled only when the unlock finishes.

When USH is below target, a trader can flash-take the PoolManager's USH, causing the before-swap arbitrage cap to become zero. The trader then buys USH at the uncorrected price. Since the physical balance remains depleted through `afterSwap`, any required residual arbitrage is also suppressed. The trader then return the remaining debt and retain the purchased USH. Flash-taking USDe enables the symmetric attack above target.

The attack requires an existing deviation, but no capital for the temporarily removed inventory. Consequently, any ordinary trader can capture arbitrage that should have accrued to the protocol, bypass the intended protocol-led peg correction, and leave any remaining deviation uncorrected by manipulating poolManager balance and capping protocol arbitrage.

RECOMMENDATION

Require the router's USH and USDe currency deltas to be zero when `_beforeSwap` begins. Alternatively, require `poolManager.getNonzeroDeltaCount() == 0`, which also detects outstanding deltas held by other addresses. However, these checks may cause multihop swaps to fail due to router deltas.

If unrestricted multihop routing is required, use a constrained trusted router that guarantees the PoolManager's physical USH and USDe balances have not been reduced before the canonical swap. In that configuration, the standard Universal Router cannot remain an unrestricted trusted sender for this pool.

RESOLUTION

Status: Resolved.

MEDIUM M-02

ACKNOWLEDGED

Terminal Fee Overcharges Entire Sale And Incentivizes Splitting

CATEGORY	Math
LOCATION	`contracts/hooks/lib/DynamicFeeLib.sol:101-103; contracts/hooks/ManifestHook.sol:1579-1615`
REVIEW	Main Review

DESCRIPTION

The hook adds the entire sell amount to the existing pressure, calculates the fee at the resulting terminal pressure, and applies that single rate to the whole transaction. Consequently, every USH is charged as though it were sold at the highest pressure reached by the transaction, even though earlier units were sold at the lower-pressure portions of the fee curve.

Although the documentation states that "a sell prices its own volume," using a volume-weighted average over the traversed pressure interval would preserve that property without charging every unit at the terminal rate or rewarding sale splitting.

This systematically overcharges sells at the highest point of the fee curve for the sell compared to averaging the fee based on sale volume. The difference is small for ordinary sales but becomes significant for large sales. When starting from low pressure and selling toward the fee cap, the terminal rate can result in almost twice the total fee of the corresponding volume-weighted average.

Users are therefore incentivized to split large sells into smaller transactions, allowing each transaction to be charged at a progressively increasing rate rather than charging the entire volume at the final rate.

Pressure accumulation remains correct because `_afterSwap` records the realized volume. Splitting does not eliminate pressure or prevent the hard breaker from eventually latching, but it avoids part of the overcharge caused by applying the terminal rate to the entire sale.

Another effect of splitting concerns the hard breaker: although it eventually latches, each sub-threshold chunk receives treasury-funded downward arbitrage until the final chunk triggers it. By contrast, an equivalent one-shot sale triggers the breaker immediately and receives no post-swap treasury support.

M-02 Terminal Fee Overcharges Entire Sale And Incentivizes Splitting CONTINUED

RECOMMENDATION

Charge each sell the volume-weighted average fee over the pressure interval traversed by that sell, rather than applying the terminal-pressure fee to the entire amount. The calculation should also account for the fee cap: if the interval crosses `maxSellFee`, calculate the average across both the linear and capped portions.

RESOLUTION

Status: Acknowledged.

MEDIUM M-03

ACKNOWLEDGED

Target Updates Enable MEV

CATEGORY	MEV
LOCATION	`contracts/hooks/ManifestHook.sol:625, :941, :1182; contracts/oracle/USHPriceOracle.sol:124`
REVIEW	Main Review

DESCRIPTION

Public NAV or usdeUsdPeg updates reveal a predictable change in targetPrice. An authorized trader can buy USH before a target increase and sell afterward, or reverse the strategy for a target decrease. During the back-run swap, beforeSwap rebalances the pool using the new target, allowing the trader to unwind at the updated price and extract value from the treasury or existing holders.

RECOMMENDATION

Use protected/private transaction delivery for NAV, `usdeUsdPeg`, and other target-changing updates. Where operationally feasible, privately pause swaps before disclosing the update, apply the update while paused, then reopen through a private atomic bundle containing an authorized minimal swap so `beforeSwap` performs catch-up before public order flow. Another thing to consider is limiting/capping target changes to decrease likelihood of an attack by decreasing possible profit.

RESOLUTION

Status: Acknowledged.

MEDIUM **M-04** ACKNOWLEDGED

Universal Router Is Incompatible For Partial Fill

CATEGORY	Compatibility
LOCATION	`contracts/hooks/ManifestHook.sol:1593`
REVIEW	Main Review

DESCRIPTION

For exact-output USH sales, the hook calls `estimateInputForOutput` to determine the USH input used for dynamic-fee calculation (`contracts/hooks/ManifestHook.sol:1593`). The estimator is capped at 256 iterations and stops when either the requested output is filled or the swap's `sqrtPriceLimitX96` is reached. The comments and unit test explicitly rely on a close protective limit allowing V4 to complete a partial fill before the estimator exhausts its iteration budget.

The importance of providing the real price limit is described by the dev comment in `SwapEstimation` :

> " `sqrtPriceLimitX96` must be the caller's real limit. V4's swap loop is uncapped and this one is not, so a walk that ignored the limit would source past where the real swap stops and could exhaust MAX_TICK_ITERATIONS, reverting a swap the pool would have partially filled."

However, the intended Universal Router does not expose `sqrtPriceLimitX96` for swaps. Its V4Router always passes the extreme minimum or maximum price ([Reference](https://github.com/Uniswap/v4-periphery/blob/dce236d4e2057422d0791d9a973a58765eb46f65/src/V4Router.sol#L228)). The user's `amountInMaximum` is checked only after the swap and cannot terminate the estimator.

Consequently, the caller-provided real `sqrtPriceLimitX96` assumed by the estimator's design is unavailable through the production router. As a result, the exact case mentioned in dev comment could occur via Universal Router. A price-limited exact-output sale that would stop within 256 iterations and partially fill may instead continue estimation until `OutputWalkCapExceeded` through Universal Router and revert.

RECOMMENDATION

M-04 Universal Router Is Incompatible For Partial Fill CONTINUED

To fully support the intended price-limited partial exact-output sales, a custom router is required. Use a custom trusted router that exposes `sqrtPriceLimitX96` and passes the same limit to `PoolManager.swap()`.

For the Universal Router support, do not rely on caller-selected limits as an estimator escape. Instead, consider conservatively handling estimator fees, or clearly document Universal Router limitations for the exact-out USH sales.

RESOLUTION

Status: Acknowledged.

MEDIUM **M-05** RESOLVED

Target updates can leave `isReverseFlex` stale, letting reverse flex satisfy the below-target add gate

CATEGORY	Logical Error
LOCATION	- `contracts/hooks/ManifestHook.sol:1557-1570` — `_flexState`, below-target branch - `contracts/hooks/ManifestHook.sol:1559-1562` — credits `balanceOf(flexWallet)` as forward backing with no `isReverseFlex` check - `contracts/hooks/ManifestHook.sol:803-818` — `_beforeAddLiquidity` hard-reverts `InsufficientFlexForLiquidity` on `!ok` - `contracts/hooks/ManifestHook.sol:826-839` — `_afterAddLiquidity` hard-reverts `InsufficientFlexForLiquidity` on `!ok` - `contracts/hooks/ManifestHook.sol:219` — `isReverseFle
REVIEW	Main Review

DESCRIPTION

The flex add-gate is the protocol's only hard brake on enlarging a below-target book. When `currentPrice < targetPrice`, a liquidity add whose range overlaps the price-to-target path increases `calculateFlexBurnedToTarget(key)`, so `_beforeAddLiquidity` and `_afterAddLiquidity` revert `InsufficientFlexForLiquidity` unless `flexWallet` already holds enough live **forward** flex to cover the enlarged requirement.

The `flexWallet` balance is direction-sensitive. With `isReverseFlex == false`, the balance is **forward flex** standing in for a below-target shortfall; with `isReverseFlex == true`, it is **reverse flex** minted while price is above target and intended to be sold back during upward arbitrage. `_flexState`'s below-target branch ignores that distinction and counts any USH in `flexWallet` as forward backing:

```
if (currentPrice < targetPrice) {
    balance = IERC20(address(ushToken)).balanceOf(flexWallet);
    needed  = calculateFlexBurnedToTarget(key);
    ok = balance >= needed;
}
```

This stale-direction state is reachable because the liquidity path refreshes `targetPrice` from live NAV/peg data, but does not reconcile the direction of the existing flex buffer. Direction is updated on swap paths instead: `_beforeSwap` may consume reverse flex during catch-up upward arbitrage, and `_afterSwap` burns/mints flex and rewrites the flag from the resulting price. A simple sequence is enough:

1. A swap leaves the pool above target with reverse flex outstanding and `isReverseFlex = true` (for example because upward arbitrage is disabled, unprofitable, or inventory-bounded).
2. Before the next swap, NAV rises or the configured USDe peg falls, so the next liquidity callback refreshes `targetPrice` above the stationary pool price.

M-05 Target updates can leave isReverseFlex stale, letting reverse flex satisfy the below-target add gate CONTINUED

3. An LP add lands first. `_flexState` now takes the below-target branch and counts the stale **reverse** flex balance as if it were forward backing.

This does not create an uncapped minting primitive: the admitted add is bounded by the stale reverse balance `R`, and later swap reconciliation burns/remints flex against the then-live requirement. The issue is that the protocol's only hard below-target add gate can be satisfied by the opposite-signed buffer, admitting adds the live forward-flex policy was meant to reject. That can leave the pool materially deeper below target than intended until a later swap reconciles the state, increasing the size of any subsequent flex adjustment or treasury-funded re-peg.

— RECOMMENDATION

Do not let `_flexState` treat reverse flex as forward backing. At minimum, the below-target branch should zero the effective balance whenever the stored direction is reverse:

```
if (currentPrice < targetPrice) {
    balance = isReverseFlex ? 0 : IERC20(address(ushToken)).balanceOf(flexWallet);
    needed  = calculateFlexBurnedToTarget(key);
    ok = balance >= needed;
}
```

More robustly, reconcile flex direction whenever `targetPrice` is refreshed on the liquidity path, or compute the gate from a direction-aware effective balance that cannot be satisfied by stale state. The fix should also review the opposite target-crossing case so `_flexState` does not waive or misapply the gate solely because the stored direction lags the live `currentPrice` / `targetPrice` regime.

— RESOLUTION

Status: Resolved.

MEDIUM M-06 RESOLVED

Remote withdrawal bridging bypasses execution-time compliance checks

CATEGORY	Out of Scope
LOCATION	`packages/ush/contracts/helpers/hub/WithdrawHelper.sol:bridgeToRemoteChain():L411-L480`
REVIEW	Main Review

DESCRIPTION

Withdrawal initiation authenticates the user, and local completion rechecks the current AuthManager status, but bridgeToRemoteChain only validates the stored user and output data. It never rechecks whether that recipient has since been banned, sanctioned, or de-whitelisted before passing the recipient directly to CCTP or LayerZero. The later bookkeeping-only finalization also has no compliance check.

Impact: A remote withdrawal queued while compliant can still deliver USDC or USDe to its user after that user becomes restricted. Normal operator processing is sufficient; the recipient need not bypass OPERATOR_ROLE. This creates a material sanctions/ban-policy bypass for pending remote redemptions, although it does not let the user exceed the stored output cap.

RECOMMENDATION

Call _checkUserAuth(user, $.authManager) in bridgeToRemoteChain before starting either bridge, and define a quarantine or compliant recovery flow for requests whose recipients become restricted. A check in markBridgeWithdrawComplete may keep state transitions consistent, but cannot substitute for the pre-bridge check because value has already been sent.

RESOLUTION

Status: Resolved.

MEDIUM **M-07** RESOLVED

Executor-supplied value irreversibly cancels local composed deposits

CATEGORY	Out of Scope
LOCATION	packages/ush/contracts/sush/ovault/VaultComposerSync.sol:depositAndSend():L370-L402 packages/ush/lib/devtools/packages/ovault-evm/contracts/VaultComposerSync.sol:lzCompose():L98-L127 packages/ush/lib/devtools/packages/ovault-evm/contracts/VaultComposerSync.sol:handleCompose():L139-L158 packages/ush/lib/devtools/packages/ovault-evm/contracts/VaultComposerSync.sol:_send():L319-L330 packages/ush/lib/devtools/packages/ovault-evm/contracts/VaultComposerSync.sol:_refund():L339-L347 packages/ush/lib/LayerZero-v2/packages/layerzero-v2/evm/protocol/contracts/MessagingComposer.sol:lzCompose():L39-L59
REVIEW	Main Review

DESCRIPTION

EndpointV2.lzCompose is publicly executable and forwards caller-selected msg.value. For a valid remote-USH compose requesting local sUSH delivery, an unrelated caller can attach the return-bridge fee as nonzero value. The vault deposit then reaches the local _send branch and reverts NoMsgValueExpected; the outer catch unwinds the deposit, spends the attached value to refund the original USH cross-chain, and returns successfully, causing EndpointV2 to consume the compose entry permanently.

Impact: Any third party can front-run execution of a pending remote-to-hub local-delivery compose and cancel the requested stake for the cost of its refund message. The intended recipient receives no sUSH, the authenticated compose cannot be retried, and the user must wait for the return bridge and initiate a new deposit, allowing targeted or repeated censorship and lost staking time without any protocol role.

RECOMMENDATION

Validate before the catch boundary that msg.value is zero for local delivery. At minimum, bubble NoMsgValueExpected and other executor-controlled value/fee failures so EndpointV2 rolls back its delivered marker and permits retry, rather than converting them into terminal refunds.

RESOLUTION

Status: Resolved.

LOW **L-01** RESOLVED

`getPendingAuthManager` reports a matured unlock time for a request that does not exist

CATEGORY	Unexpected Behavior
LOCATION	`contracts/hooks/ManifestHook.sol:417-422`
REVIEW	Main Review

DESCRIPTION

`getPendingAuthManager` derives the returned `unlockTime` directly from `_setAuthManagerReqTimestamp` without checking whether a request is actually outstanding:

```
function getPendingAuthManager() external view returns (address pendingManager, uint256
  unlockTime) {
    pendingManager = _pendingAuthManager;
    unchecked { unlockTime = _setAuthManagerReqTimestamp + TIMELOCK_PERIOD; }
}   // :417-422
```

When no AuthManager rotation is pending, `_pendingAuthManager == address(0)` and `_setAuthManagerReqTimestamp == 0`. Both state variables are zeroed together in two places:

- the successful-apply branch of `setAuthManager` (`ManifestHook.sol:393-394`), and
- `cancelAuthManagerUpdate` (`ManifestHook.sol:411-412`).

In that state the view still computes `unlockTime = 0 + TIMELOCK_PERIOD = 7 days`. The function therefore returns a `unlockTime` that reads as "already matured / ready to execute" for a rotation that does not exist.

RECOMMENDATION

Return an unambiguous sentinel when nothing is pending:

```
function getPendingAuthManager() external view returns (address pendingManager, uint256
  unlockTime) {
    pendingManager = _pendingAuthManager;
    if (_pendingAuthManager != address(0)) {
        unchecked { unlockTime = _setAuthManagerReqTimestamp + TIMELOCK_PERIOD; }
    }
    // else unlockTime = 0
}
```

Returning `unlockTime = 0` when `_pendingAuthManager == address(0)` makes the "no request" state distinguishable from a matured one for every off-chain consumer.

L-01 `getPendingAuthManager` reports a matured unlock time for a request that does not exist CONTINUED

— RESOLUTION

Status: Resolved.

LOW **L-02** RESOLVED

`usdeUsdTolerance` is decoupled from the ±0.5% `usdeUsdPeg` band, leaving a window where swaps continue against an uncorrectable target price

CATEGORY	Logical Error
LOCATION	`contracts/hooks/ManifestHook.sol:640-644 (setUsdeUsdTolerance), :628-632 (setUsdeUsdPeg), :1536-1538 (_applyNavToTarget); contracts/hooks/lib/HookMath.sol:36 (isUsdeDepegged)`
REVIEW	Main Review

DESCRIPTION

`targetPrice`, the USDe-per-USH price the arbitrage engine defends, is derived as `targetPrice = NAV / usdeUsdPeg` (`_applyNavToTarget`, `:1538`). `usdeUsdPeg` (`p`) represents the USDe/USD price and is owner-adjustable only within `[0.995e18, 1.005e18]`, a **±0.5% band** (`setUsdeUsdPeg`, `:629`). Per the dev comment at `:89-91`, `p` is intended to track real USDe drift ("lets governance nudge p for minor USDe/USDC drift").

Separately, the USDe/USD depeg circuit breaker halts swaps only when the live oracle price deviates from `$1` by more than `usdeUsdTolerance` (`isUsdeDepegged`, `HookMath.sol:36`): `deviation = |oraclePrice - 1e18|; return deviation > tolerance`. `usdeUsdTolerance` defaults to `0.01e18` (1%) and is owner-settable up to `MAX_USDE_USD_TOLERANCE = 0.05e18` (5%) with **no lower coupling to the peg band** (`setUsdeUsdTolerance`, `:641`). The breaker compares the oracle to a hardcoded `1e18`, never to `p`.

Because the breaker's halt band (± `tolerance`, ≥1%) is **wider than the peg knob's correction range (±0.5%)**, there is a drift window, USDe more than 0.5% but less than `tolerance` away from `$1`, in which:

1. the depeg breaker does **not** trip, so swaps keep executing, and
2. `usdeUsdPeg` **cannot** be set to the true USDe price (it saturates at `0.995` / `1.005`),

so `targetPrice = NAV / p` is computed against a `p` pinned at the band edge while USDe actually trades further out. The pool then defends a structurally wrong target, and no breaker catches it because the entire window is, by construction, inside `tolerance`. The USD-terms mispricing is approximately `(true USDe drift − 0.5%)`:

L-02 usdeUsdTolerance is decoupled from the ±0.5% usdeUsdPeg band, leaving a window where swaps continue against an uncorrectable target price

CONTINUED

- **USDe above `$1`** → `p` pinned low → `targetPrice` too high → USH defended above its USD backing → downward arbitrage overpays treasury USDe to buy USH.
- **USDe below `$1`** → `p` pinned high → `targetPrice` too low → USH defended below its USD backing → external arbitrageurs buy USH from the pool under fair value and drain the discount.

At the default 1% tolerance the window is 0.5%–1% drift (≈0.5% mispricing) and is **live with no configuration change**; at the 5% maximum it widens to roughly **4.5%** mispricing. The condition is a market state (USDe sitting in the gap), not attacker-triggered, but an arbitrageur can opportunistically harvest it whenever it occurs, and no circuit breaker or event flags that the pool is defending an uncorrectable target.

RECOMMENDATION

Derive `targetPrice` from the live `usdeUsdOracle` price already read each swap, instead of the ±0.5%-capped `usdeUsdPeg`, so the target tracks USDe up to the halt point and no gap exists.

RESOLUTION

Status: Resolved.

LOW L-03

PARTIALLY RESOLVED

Full-route Profitability Check May Skip Profitable Partial Arbs

CATEGORY	Logical Error
LOCATION	`contracts/hooks/ManifestHook.sol:1182; contracts/hooks/lib/HookMath.sol:73; contracts/hooks/lib/ArbExec.sol:92`
REVIEW	Main Review

DESCRIPTION

ManifestHook evaluates arbitrage profitability for the complete route to `targetPrice` or `arbTarget` before applying the treasury and shared-inventory caps. If the complete route is classified as unprofitable, execution is skipped and the caps are never considered.

With a nonzero V4 protocol fee, the initial portion of an arbitrage might be profitable while the later portion near the target makes the complete route unprofitable. An arb trade after the caps applied might have executed only on that profitable prefix, but these trades are never executed due to before cap full route profitability check.

Consequently, the hook can miss backing-safe peg-defense and reserve-accretion opportunities, leaving a larger price discrepancy for flex or later transactions.

RECOMMENDATION

Consider applying the treasury and shared-inventory caps before the final profitability decision and evaluate the real capped amount instead of the complete route. If this behavior is accepted, document that the full-route profitability check may skip otherwise profitable capped execution.

RESOLUTION

Status: Partially resolved.

Guardian: The original issue is substantially addressed, but the remediation retains the dust-level search inconsistency documented in remediations I-01 and introduces gas-limit related findings described in remediations L-01 and L-02

LOW L-04

RESOLVED

Duplicate Queued Additions Can Be Performed Without A Fresh Delay

CATEGORY	Logical Error
LOCATION	`contracts/module/PositionFeeModule.sol:155,174; contracts/module/base/SafeModuleBase.sol:274`
REVIEW	Main Review

DESCRIPTION

`queueAddTokenId` checks only whether the ID is currently active. It does not prevent multiple pending additions for the same inactive ID.

Example sequence:

1. Queue operations A and B for token ID X.
2. After the delay, execute A; X becomes collectable.
3. Executing B now reverts because X is already collectable.
4. Operation B stays live.
5. Token X is removed.
6. Execute B during its existing execution window.
7. X is re-added without a newly announced delay.

Only the current owner can use this sequence, so it is not an outsider authorization bypass. However, it does weaken the stated "removal is immediate, widening must wait" response model.

RECOMMENDATION

Do not allow duplicate actions being queued as it can allow time delay bypass for the second action.

RESOLUTION

Status: Resolved.

LOW L-05

RESOLVED

`setDynamicFeeParams` retroactively re-decays stored pressure when `pressureHalfLife` changes

CATEGORY	Logical Error
LOCATION	`contracts/hooks/ManifestHook.sol:1507-1543, :1545-1548, :1136-1137; contracts/hooks/lib/DynamicFeeLib.sol:198-217; contracts/hooks/ManifestHook.sol:1593-1595`
REVIEW	Main Review

DESCRIPTION

`setDynamicFeeParams` overwrites `pressureHalfLife` directly, but it does not first settle `netSellPressure` or restamp `lastPressureUpdateTime`. The decay function does not know that the half-life changed: `_applyDecay` always computes decay over the full `block.timestamp - lastUpdate` interval using the half-life passed into that call.

What happens now is path-dependent accounting. If governance raises the half-life, pressure that had already economically decayed under the old half-life is partially resurrected. If governance lowers the half-life, pressure that should still stand is erased. The effect is immediate because `quoteFees` reads the same stale `netSellPressure` and `lastPressureUpdateTime` through `previewFee`, so the quoted sell fee can jump even when no new trade has occurred.

What should happen instead is a checkpoint at the old rate: first decay the stored pressure from the old timestamp to the retune timestamp using the old `pressureHalfLife`, then set `lastPressureUpdateTime = block.timestamp`, and only then install the new half-life for future elapsed time.

The impact is fee mis-accounting around governance retunes. A routine parameter change can temporarily overcharge or undercharge the next sells relative to the intended curve, and if standing pressure is large enough the inflated post-retune fee can help push the next realized sell to `maxSellFee`, which latches `hardBreakerTripped`. This remains a low-severity configuration issue because the setter is `onlyOwner`, the effect is reversible by the same role, and I did not find an untrusted path to trigger it.

RECOMMENDATION

Checkpoint pressure before changing the decay parameter. Inside `setDynamicFeeParams`, decay `netSellPressure` using the current `pressureHalfLife`, write the settled value back, set `lastPressureUpdateTime = block.timestamp`, and then store the new `_pressureHalfLife`.

L-05 `setDynamicFeeParams` retroactively re-decays stored pressure when `pressureHalfLife` changes — CONTINUED

RESOLUTION

Status: Resolved.

LOW L-06 ACKNOWLEDGED

quoteFees under-quotes the sell fee for an exact-output sell

CATEGORY	Math
LOCATION	`contracts/hooks/ManifestHook.sol:1452 (quoteFees → DynamicFeeLib.previewFee, DynamicFeeLib.sol:170-186)`
REVIEW	Main Review

DESCRIPTION

`quoteFees` is the view integrators call to preview the sell fee. It returns the plain surcharge, `_feeFor(pressure + amount)`, with no gross-up and no protocol fee.

The fee actually charged by `computeSellFee` matches that preview for an **exact-input** sell, but not for an **exact-output** sell. On the exact-output path (`DynamicFeeLib.sol:82-98`) `computeSellFee` first grosses the amount up (by `1 / (1 - combined)`, where `combined` also includes the V4 protocol fee) and then prices the fee off that larger amount. `previewFee` (`DynamicFeeLib.sol:170-186`) has neither the gross-up nor a `protocolFeePips` input, so it cannot reproduce that number.

The result is that for an exact-output sell the previewed fee is always **less than or equal to** the fee charged, and strictly less whenever the gross-up moves the fee. The gap is small at the current `maxSellFee` (5%) and grows as `maxSellFee` is raised. It only affects off-chain consumers of `quoteFees`; no in-scope contract reads it, and nothing on-chain is mispriced.

The direction of the error is what makes it worth flagging: the quote comes back **lower** than the fee the swap actually charges. An under-*charge* would only benefit the trader, but an under-*quote* means an integrator or user who sizes a trade off `quoteFees` is then charged **more** than the preview showed, the surprise lands on the user, not in their favor. This is a view-layer correctness issue (the quote is really an exact-input approximation), not an on-chain mispricing.

RECOMMENDATION

At minimum, document that the quote is an approximation and can under-state the fee actually charged. Preferably, make the preview reproduce the charged fee:

- Add optional `protocolFeePips` and `grossUp` parameters to `previewFee` / `quoteFees` so an integrator can preview an exact-output sell exactly, or
- Document in the `quoteFees` natspec that the returned value is the exact-input sell fee,

L-06 quoteFees under-quotes the sell fee for an exact-output sell CONTINUED

that it is not exact for an exact-output sell, and that an exact-output sell is charged slightly more (the gross-up plus protocol-fee term) — so integrators do not size trades against a quote that is lower than the real charge. The gap grows as `maxSellFee` increases.

— RESOLUTION

Status: Acknowledged.

INFORMATIONAL I-01 ACKNOWLEDGED

Flex Wallet Can Trigger Repeated USH Minting

CATEGORY	Trust Assumptions
LOCATION	`contracts/hooks/ManifestHook.sol:1059-1118`
REVIEW	Main Review

DESCRIPTION

The hook calculates flex using the USH balance currently held by `flexWallet`. If this balance is below the calculated requirement, `_afterSwap` mints the deficit.

The flex wallet can therefore transfer its USH elsewhere and, while a nonzero flex requirement persists or is recreated, have a subsequent swap mint replacement tokens. The transferred USH remains outstanding and is not tracked or recovered. Although each refill is bounded by the current flex requirement, repeating this process permits uncapped aggregate USH issuance.

This is not permissionless because it requires control of the flex wallet and qualifying swaps. Nevertheless, the Safe quorum and any module capable of transferring USH are effectively part of the protocol's minting trust boundary despite the wallet not holding `MINTER_ROLE`.

RECOMMENDATION

Clearly document it as part of the minting trust boundary, ensure funds are not transferred from flex wallet, and monitor unexpected USH outflows. Alternatively, consider using a dedicated non-withdrawable escrow as the flex wallet, since flex USH only needs to be minted and burned by the hook.

RESOLUTION

Status: Acknowledged.

INFORMATIONAL I-02

RESOLVED

Misleading `onlyPauser` modifier name and `OnlyPauser` error

CATEGORY Best Practices
LOCATION `contracts/hooks/ManifestHook.sol:278-281`
REVIEW Main Review

DESCRIPTION

The `onlyPauser` modifier permits **both** the `owner` and the `pauser` to pass:

```
modifier onlyPauser() {
    if (msg.sender != owner && msg.sender != pauser) revert OnlyPauser();
    _;
}
```

The modifier name (`onlyPauser`) and the revert error (`OnlyPauser()`) both imply that access is exclusive to the `pauser` , when in fact the `owner` is equally authorized.

RECOMMENDATION

Rename the modifier and error to reflect that both roles are permitted, e.g. `onlyPauserOrOwner` and a corresponding `NotAuthorizedToPause()` / `OnlyPauserOrOwner()` error.

RESOLUTION

Status: Resolved.

INFORMATIONAL **I-03** RESOLVED

`setAuthManager` silently succeeds without effect when the pending address is re-submitted before maturity

CATEGORY	Unexpected Behavior
LOCATION	`contracts/hooks/ManifestHook.sol:386-404`
REVIEW	Main Review

DESCRIPTION

`setAuthManager` (`ManifestHook.sol:376-405`) is a three-branch `if / else if / else if` chain, and one reachable input combination matches **none** of the branches, so the call falls through to nothing, no state change, no event, no revert:

```
function setAuthManager(address _authManager) external onlyOwner {
    if (_authManager == address(0)) revert InvalidAuthManager();

    uint256 timelockReady;
    uint256 timelockExpiry;
    unchecked {
        timelockReady = _setAuthManagerReqTimestamp + TIMELOCK_PERIOD;
        timelockExpiry = timelockReady + TIMELOCK_EXPIRY;
    }

    if (
        block.timestamp >= timelockReady && block.timestamp <= timelockExpiry
            && _authManager == _pendingAuthManager && _pendingAuthManager != address(0)
    ) {
        // branch 1: timelock passed and not expired -> execute update
        ...
    } else if (
        block.timestamp > timelockExpiry && _pendingAuthManager != address(0)
            && _authManager == _pendingAuthManager
    ) {
        revert TimelockExpired();                    // branch 2
    } else if (_authManager != _pendingAuthManager) {
        _pendingAuthManager = _authManager;          // branch 3: stage a new request
        _setAuthManagerReqTimestamp = block.timestamp;
        emit AuthManagerUpdateRequested(_authManager, block.timestamp);
    }
    // no else -> unhandled combination falls through silently
}
```

The unhandled combination is precisely: **`_authManager == _pendingAuthManager` (non-zero) while `block.timestamp < timelockReady`**, i.e. the owner re-submits the address that is already pending, before the 7-day timelock has matured.

I-03 setAuthManager silently succeeds without effect when the pending address is re-submitted before maturity

CONTINUED

- Branch 1 requires `block.timestamp >= timelockReady`: not yet true.
- Branch 2 requires `block.timestamp > timelockExpiry`: not yet true.
- Branch 3 requires `_authManager != _pendingAuthManager`: false, since they are equal.

The transaction therefore mines successfully, changes no state, emits no event and reverts nothing, leaving the old `AuthManager` in place. To an operator or a governance script that treats a mined (non-reverting) transaction as execution, this is **indistinguishable from a successful rotation**.

RECOMMENDATION

Close the unhandled branch with an explicit terminal case, e.g. `else revert TimelockNotReady();` (or emit a distinct `NoOp` / `StillPending` event) so a re-submission before maturity cannot be mistaken for execution.

RESOLUTION

Status: Resolved.

INFORMATIONAL **I-04** RESOLVED

Redundant `onlyPoolManager` modifier on internal hook overrides

CATEGORY	Superfluous Code
LOCATION	`contracts/hooks/ManifestHook.sol:768 (_beforeInitialize), :806 (_beforeAddLiquidity), :878 (_beforeRemoveLiquidity), :1159 (_beforeDonate)`
REVIEW	Main Review

DESCRIPTION

Four internal hook overrides apply `onlyPoolManager` a second time, but the check is already enforced one level up and cannot fail when the internal function runs.

`ManifestHook` extends `BaseHook`, whose **external** entrypoints are the functions the `PoolManager` actually calls, and they already carry `onlyPoolManager`:

```
// lib/v4-periphery/src/utils/BaseHook.sol
function beforeInitialize(address sender, PoolKey calldata key, uint160 sqrtPriceX96)
    external
    onlyPoolManager
    returns (bytes4)
{
    return _beforeInitialize(sender, key, sqrtPriceX96); // dispatches to the override
}
```

The internal overrides are reachable only through those external entrypoints, so by the time `_beforeInitialize` / `_beforeAddLiquidity` / `_beforeRemoveLiquidity` / `_beforeDonate` execute, `msg.sender == address(poolManager)` has already been established and the second check can never revert. `_beforeSwap` (`:922`) and `_afterSwap` (`:992`) omit the modifier, so the codebase is also inconsistent on this point.

RECOMMENDATION

Remove `onlyPoolManager` from the four internal overrides — the `BaseHook` external entrypoints already enforce it — and align with `_beforeSwap` / `_afterSwap`, which correctly omit it.

RESOLUTION

Status: Resolved.

INFORMATIONAL I-05 ACKNOWLEDGED

No explicit cap on the dynamic sell fee; callers can only bound it indirectly via output slippage

CATEGORY	Informational
LOCATION	`contracts/hooks/ManifestHook.sol:980-987 (dynamicFee returned as the pool LP fee), :922 (unused hookData argument)`
REVIEW	Main Review

DESCRIPTION

The sell fee is dynamic: `_beforeSwap` computes it from whatever `netSellPressure` (and the surcharge denominator) happen to be at execution time, and returns it as the pool's LP fee (`OVERRIDE_FEE_FLAG | dynamicFee`). A caller has no direct way to say "revert if the fee exceeds X":

- `SwapParams` carries only `zeroForOne`, `amountSpecified`, and `sqrtPriceLimitX96`. The price limit does not help, because V4 takes the LP fee off the input *before* the curve step, so a higher fee produces *less* price movement, not more, `sqrtPriceLimitX96` is never the binding constraint on a high-fee swap.
- The `hookData` argument to `_beforeSwap` is unnamed and never read, so it cannot carry a caller-supplied fee cap.

Because the surcharge tracks live sell pressure (which can rise between the time a caller quotes and the time the swap executes, e.g. another sell banks pressure first), the fee a swap actually pays is not fixed at submission and cannot be bounded by any pool-level parameter.

Callers are not left economically unprotected, however. For an exact-input sell the fee is taken from the input before the curve, so a higher fee yields strictly less output; the Universal Router's `amountOutMinimum` (and, for exact-output, `amountInMaximum`) therefore reverts the swap whenever a fee spike pushes the result past the caller's slippage tolerance. Output slippage thus bounds the cost indirectly. The residual effect is limited to (a) callers who route directly through the `PoolManager` with no output-amount check receive no protection, and (b) a fee spike (organic or induced by a prior sell) can turn a caller's swap into a slippage revert (wasted gas) rather than a fill.

I-05 No explicit cap on the dynamic sell fee; callers can only bound it indirectly via output slippage CONTINUED

RECOMMENDATION

Document that the sell fee is dynamic and uncapped on-chain, and that the only cost bound available to integrators is router-level output slippage.

RESOLUTION

Status: Acknowledged.

INFORMATIONAL I-06

ACKNOWLEDGED

Informational Note Regarding Oracle Latency

CATEGORY	Trust Assumption
LOCATION	`contracts/hooks/lib/HookMath.sol:42`
REVIEW	Main Review

DESCRIPTION

The hook detects a USDe depeg only from the oracle's latest report and there is an inherent trust on the oracle. Because the feed updates approximately daily, a market depeg may occur before it is reported onchain. During this time gap between a depeg event and its reporting on chain, the previous round remains valid, the circuit breaker stays inactive, and swaps, arbitrage, and flex operations continue under the near-$1 assumption. This can result unbacked minting and arbitrage opportunity for traders.

RECOMMENDATION

Monitor USDe prices off-chain and pause the hook when a depeg is observed but not yet reported by the oracle.
Also consider bounding cumulative upward-arbitrage minting per oracle round/epoch to prevent unbacked minting in case of a depeg event.

RESOLUTION

Status: Acknowledged.

INFORMATIONAL I-07

ACKNOWLEDGED

Exact-output sells pay a slightly lower dynamic sell surcharge than exact-input

CATEGORY	Math
LOCATION	contracts/hooks/lib/DynamicFeeLib.sol:82-98 (computeSellFee gross-up), :96-98 (f0 evaluated at un-grossed pressure), :146-157 (_feeFor clamp); contracts/hooks/ManifestHook.sol:1389 (setDynamicFeeParams maxSellFee bound), :1593 (_computeDynamicFee passes grossUpSellTerm = true)
REVIEW	Main Review

DESCRIPTION

The sell surcharge is priced from `updatedPressure = decayed + term`, where `term` is the USH volume of the swap folded into its own fee. Exact-input and exact-output sells supply that volume differently:

- **Exact-input** sells: `term` is the fee-INCLUSIVE USH the trader specifies. Correct as-is.
- **Exact-output** sells: `term` is the fee-EXCLUSIVE geometric walk result `g`, which is grossed up to the fee-inclusive amount so both directions price the same real volume the same.

The gross-up (`DynamicFeeLib.sol:98`) computes `term = g · MAX_LP_FEE / (MAX_LP_FEE − combined)`, with `combined` derived from `f0 = _feeFor(decayed + g)`, the fee at the **un-grossed** pressure (`:96`). The fee actually charged is `_feeFor(decayed + term)` at the higher grossed pressure. This is a **single pass of a circular relation** (`term` depends on the fee, the fee depends on `term`) and does not iterate, so `combined` is slightly understated, `term` lands just below the true fee-inclusive input, and the exact-output surcharge comes out marginally **below** an exact-input sell of the same volume. The direction is one-sided: exact-output is always `≤` exact-input, never above. The dev comment at the gross-up documents the one-pass design intent.

The size of the gap is governed by the gross-up factor `1 / (1 − fee)`, which is flat at low fees and steep near the cap:

- At the shipped **`maxSellFee = 5%`** the divergence is about **0.014% of the trade**, negligible.
- It stays small through moderate caps and only becomes material at high caps: roughly **12% of the trade at a 40% cap**, rising toward the fee itself as `maxSellFee` approaches the `≈99.59%` ceiling.

So at present values this is immaterial. It would only matter if governance raised `maxSellFee` to a large value, for instance to stiffen the dump deterrent during a depeg, in

I-07 Exact-output sells pay a slightly lower dynamic sell surcharge than exact-input CONTINUED

which case an exact-output dumper would pay materially less surcharge than an exact-input dumper for the same sell. The awareness point is that the gap is not a fixed dust amount; it scales with `maxSellFee`.

RECOMMENDATION

No change is required at the current cap. Track this as a known limitation and revisit it before ever configuring a high `maxSellFee`:

- Document that exact-output sells are charged slightly less surcharge than exact-input, and that the gap is `~0.014%` of the trade at a 5% cap but grows with `maxSellFee`.
- If a high `maxSellFee` is ever adopted, iterate the gross-up's fixed point to convergence rather than in a single pass, so the exact-output term equals the fee-inclusive input regardless of the cap.

RESOLUTION

Status: Acknowledged.

INFORMATIONAL I-08

RESOLVED

Misleading comment: "Price exactly at target" branch also runs while price is above target

CATEGORY	Documentation
LOCATION	`contracts/hooks/ManifestHook.sol:1122` — comment `// Price exactly at target, no reverse flex needed`
REVIEW	Main Review

DESCRIPTION

In the `_afterSwap` reverse-flex branch (entered when `currentPrice > targetPrice`), the `else` taken when `reverseFlexNeeded == 0` is commented "Price exactly at target." It isn't only reachable at target: `calculateFlexMintedToTarget` can return `0` for a price a hair **above** target, because it compares raw sqrt prices (`sqrtStartX96` vs `sqrtTargetX96`) and floor-rounds the surplus, while the branch guard compared prices in the USH frame. In that dust band the two disagree, so the "exactly at target" branch runs while price is strictly above it.

The branch's behavior (burn stale flex, set `isReverseFlex = false`) is correct, with no gap to fund there is nothing to mint, so this is a comment-accuracy nit only.

RECOMMENDATION

Reword the comment to the real condition, e.g. `// reverseFlexNeeded == 0: no gap to fund (at target, or within a dust band just above it) — clear flex and drop the flag.`

RESOLUTION

Status: Resolved.

INFORMATIONAL I-09 ACKNOWLEDGED

Queued Actions Carry Over To A New Owner

CATEGORY	Unexpected Behavior
LOCATION	`contracts/module/base/SafeModuleBase.sol:274,407; documentation/03-access-control/timelock-mechanisms.md:245`
REVIEW	Main Review

DESCRIPTION

Pending token additions and bot rotations are not cleared when ownership changes. Operation IDs include the queueing owner, but consumption checks only that the caller is the current owner. Consequently, a successor owner can execute an inherited operation queued by the previous owner.

This contradicts the `timelock-mechanisms.md` statement that only the original initiator can execute an operation.

An old owner cannot execute after losing ownership, this is mainly a governance-handover and stale-capability risk.

RECOMMENDATION

No change is needed if this is acceptable. Otherwise, clear queued actions after ownership change or check the originator and executor are the same.

RESOLUTION

Status: Acknowledged.

INFORMATIONAL I-10

ACKNOWLEDGED

No Minimum Timelock Delay Check

CATEGORY	Configuration
LOCATION	`scripts/DeployPositionFeeModule.s.sol:48,136; contracts/module/base/SafeModuleBase.sol:32,183,230`
REVIEW	Main Review

DESCRIPTION

The `DEFAULT_TIMELOCK_DELAY` defaults to six hours, but accepts any nonzero delay up to seven days. There isn't any minimum delay check and can be set to 1 second. As a result, all actions can be performed in the next block with basically no timelock.

RECOMMENDATION

No change required if this is intended. Otherwise, consider adding a minimum delay parameter.

RESOLUTION

Status: Acknowledged.

INFORMATIONAL **I-11** ACKNOWLEDGED

Timelock execution window is inclusive on both ends, so it spans `TIMELOCK_EXPIRY + 1` second rather than exactly `TIMELOCK_EXPIRY`

CATEGORY	Logical Error
LOCATION	- `contracts/module/base/SafeModuleBase.sol:284` — `if (block.timestamp < op.executeAfter) revert TimelockNotReady(operationId);` - `contracts/module/base/SafeModuleBase.sol:285` — `if (block.timestamp > op.executeAfter + TIMELOCK_EXPIRY) revert TimelockExpired(operationId);`
REVIEW	Main Review

DESCRIPTION

`_consumeReserved` bounds the executable window with a strict `<` on the lower edge and a strict `>` on the upper edge, so both boundary timestamps are themselves executable:

- lower edge `<` → executable when `now >= executeAfter` (start instant included), and
- upper edge `>` → executable when `now <= executeAfter + TIMELOCK_EXPIRY` (end instant included).

The window is therefore the closed interval `[executeAfter, executeAfter + TIMELOCK_EXPIRY]`. Counted in whole seconds, an operation is executable for `TIMELOCK_EXPIRY + 1` seconds (both endpoints inclusive), i.e. `7 days + 1 second`, not the `7 days` the constant name implies. The `TIMELOCK_DELAY` readiness edge is correct as-is (`<` makes the op executable the instant the delay is satisfied); the extra second lives only on the expiry edge.

RECOMMENDATION

If an exactly-`TIMELOCK_EXPIRY` window is intended, make the upper edge exclusive by changing the expiry check to `>=`:

```
if (block.timestamp >= op.executeAfter + TIMELOCK_EXPIRY) revert
  TimelockExpired(operationId);
```

RESOLUTION

Status: Acknowledged.

INFORMATIONAL I-12

ACKNOWLEDGED

Protocol-minted forward flex is counted in the sell-surcharge denominator, diluting the sell deterrent while the pool rests below target

CATEGORY Business logic errors

LOCATION - `contracts/hooks/lib/DynamicFeeLib.sol:229-243` — `_surchargeDenominator` returns `totalSupply - poolUshBaseline` - `contracts/hooks/lib/DynamicFeeLib.sol:156-168` — `_feeFor`, where the surcharge is `surchargeMultiplier · pressure / denominator` - `contracts/hooks/lib/DynamicFeeLib.sol:119-142` — `reconcilePressure` derives the hard-breaker `feeLevel` from the same denominator - `contracts/hooks/ManifestHook.sol:1164-1194` — `_afterSwap` forward-flex mint `ushToken.mint(flexWallet, toMint, 3)` - `contracts/hooks/ManifestHook.sol:719-753` — `calculateFlexBurnedToTarget` (the net gap, sized to `targetPrice`) - `contracts/hooks/ManifestHook.sol:1319-1344` — downward-arb trigger deadband (`arbTrigger = arbTarget - swapPriceTolerance`)

REVIEW Main Review

DESCRIPTION

This is a documentation/observation note on intended behavior that has a side effect worth being aware of, not an exploitable defect.

The sell surcharge prices accumulated sell pressure against a denominator documented as "the float that can actually sell", issuance outside the pool, computed as `totalSupply - poolUshBaseline` (`_surchargeDenominator`) and sitting under pressure in the fee curve (`_feeFor`), so `surcharge ∝ pressure / denominator`: a larger denominator yields a smaller fee. When the pool sits below `targetPrice`, `_afterSwap` mints forward flex, protocol USH, to `flexWallet` to stand in for the under-backing gap (`ManifestHook.sol:1184`). That mint raises `totalSupply`; `poolUshBaseline` is never touched by flex, so the minted flex flows into the denominator and lowers the surcharge. Forward flex is protocol-owned bridging inventory that is only ever burned (never dumped on the pool), yet it is counted in the float that prices retail dumps. The same denominator feeds `reconcilePressure`'s hard-breaker `feeLevel`, so the breaker latches slightly later as well.

I-12 Protocol-minted forward flex is counted in the sell-surcharge denominator, diluting the sell deterrent while the pool rests below target — CONTINUED

The observation is only that including it in the surcharge denominator softens the deterrent, and two states make it noticeable. In normal operation, when the owner sets `targetArbPrice < targetPrice` (its documented treasury-conservation purpose) or `swapPriceTolerance > 0`, the arb fires only below `arbTrigger = arbTarget - swapPriceTolerance` and sizes to `arbTarget`, so the pool has a designed resting band `[arbTrigger, targetPrice)` where the arb is silent but `_afterSwap` still mints flex to the full `targetPrice`. Because the arb targets `arbTarget` while the flex measures to `targetPrice`, the flex for the `[arbTarget, targetPrice)` slice covers a gap the protocol deliberately declines to close and has no mechanism to push price up through, so it stays continuously outstanding and the surcharge carries a standing discount even with no depeg. In a hard-breaker deep depeg the effect is larger: the downward arb is disabled entirely, `flexNeeded` grows with the depeg, and the surcharge shrinks toward `baseSellFee`.

The effect is bounded, which is why it is informational. `calculateFlexBurnedToTarget` returns the net gap `ushOut - usdeInValue`, second-order in the price deviation, so the erosion is minor at realistic depegs and only becomes sizeable where flex approaches the free float. It is a self-weakening of the deterrent, not a fee-evasion or griefing primitive: the surcharge is quoted before a swap's own flex mint, so a seller's own realized volume adds its full amount to the pressure numerator while adding at most the net gap to the denominator, and splitting or pushing the price down first never lowers the seller's own total fee.

RECOMMENDATION

No change is required if the softened deterrent in the below-target band is acceptable. If the team wants the surcharge to price only the genuine retail float, the direction is to exclude hook-minted flex from the denominator via a dead-reckoned counter rather than a `balanceOf(flexWallet)` read (which a stray transfer could move). Note this is not a free change: the counter would have to be maintained at every flex mint/burn site, forward, reverse, the reverse-flex clears, and the `ArbExec` consume, with clamped decrements so donations cannot desync it, so it adds accounting surface and should be weighed against the small, bounded benefit.

RESOLUTION

Status: Acknowledged.

INFORMATIONAL I-13

ACKNOWLEDGED

Treasury rotation non-atomically activates replacement capacity for a searcher bracket

CATEGORY	state-transition / economic MEV
LOCATION	packages/ush/contracts/hooks/ManifestHook.sol:setTreasury():L1320-L1325 packages/ush/contracts/hooks/ManifestHook.sol:_runArbitrage():L1198-L1222 packages/ush/contracts/hooks/lib/ArbExec.sol:executeDownwardArbitrage():L158-L182
REVIEW	Main Review

DESCRIPTION

setTreasury() immediately activates the replacement reserve without settling the current price/flex gap or preventing public swaps across the transition. With downward arbitrage continuously enabled, ordinary sells can consume the retiring reserve only down to minTreasuryBalance and leave USH below target. A searcher can buy the residual discount before a public rotation and sell afterward; beforeSwap uses the replacement capacity to restore price before the searcher's fill.

Impact: A reserve rotation can create a predictable MEV bracket that diverts price-to-NAV recovery surplus from protocol peg defense/rewards to a priority trader. The strengthened PoC uses no treasury drain, allowance failure, breaker, or arb-disable toggle: normal arbitrage stops at the configured old reserve floor, an unlimited-approved replacement is installed, over 250,000 USDe funds catch-up, and a 100,000-USDe position realizes over 10,000 USDe fee-net profit.

RECOMMENDATION

Document that activating replacement treasury capacity makes an existing recovery spread contestable. If protocol-first recovery is desired, operationally pause swaps, rotate the treasury, reconcile the pool, and then unpause

RESOLUTION

Status: Acknowledged.

INFORMATIONAL I-14 RESOLVED

Cross-chain deposits settle after global whitelist revocation

CATEGORY	Out Of Scope
LOCATION	packages/ush/contracts/helpers/hub/DepositHelper.sol:deposit():L146-L190 packages/ush/contracts/helpers/hub/DepositHelper.sol:_lzReceive():L221-L267 packages/ush/contracts/helpers/hub/DepositHelper.sol:completeDeposit():L282-L315 packages/ush/contracts/ush/USHTokenBase.sol:_checkAuthSingle():L305-L317
REVIEW	Main Review

DESCRIPTION

DepositHelper snapshots whitelist authorization at source initiation, while destination registration and completeDeposit never consult current authorization. Even if the backend performs INV-CC-15's required check before submitting completion, an independent revocation can be ordered before execution and the stale completion still succeeds. This is a realistic permitted path because production whitelists completeDeposit by selector for the Automation Safe through SafeExecutorModule, whose authorized bot controls calldata. The updated PoC verifies authorization successfully, revokes both source and destination before settlement, proves new deposits fail, and then settles both queued claims. USHToken's transfer checks only bans and sanctions and therefore cannot supply the missing deposit-operation whitelist check.

Impact: Whitelist revocation cannot atomically stop queued USH settlement. Pending routine automation can release every ratio-compliant claim after global revocation, up to prefunded helper inventory, and a lagging source can create further claims before synchronization. The released USH remains transferable. Payouts stay backed and capped, so the impact is an authorization/compliance bypass rather than unbacked theft.

RECOMMENDATION

Check the recorded user against the destination AuthManager inside completeDeposit immediately before changing state and transferring USH. Keep off-chain verification as defense in depth.

RESOLUTION

Status: Resolved.

INFORMATIONAL I-15

ACKNOWLEDGED

Treasury-clipped downward arbitrage can realize dust losses even when the full re-peg is profitable

CATEGORY	Rounding
LOCATION	contracts/hooks/ManifestHook.sol:1343-1356 — _runArbitrage checks profitability for the full downward arbitrage, then clips the spend to the treasury's currently spendable USDe before execution contracts/hooks/lib/HookMath.sol:131-159 — calcUsdeForArbitrage returns a profitability verdict for the full buy-to-target path contracts/hooks/lib/ArbExec.sol:206-235 — executeDownwardArbitrage executes the clipped exact-input swap and books the received USH after swap/output rounding
REVIEW	Main Review

DESCRIPTION

`calcUsdeForArbitrage` returns `profitable = true` when the full downward arbitrage path to `arbTarget` yields enough USH, valued at `targetPrice`, to cover the full `usdeToSpend`. `_runArbitrage` then reuses that verdict after clipping the spend to spendable, assuming the same conclusion still holds for the smaller partial trade.

That implication is not guaranteed. Once the spend is clipped into a very small exact-input amount, fee and curve rounding can push the executed swap into a dust regime where the hook spends a small amount of treasury USDe but receives zero or negligible realizable value back at `targetPrice`. The full re-peg may be profitable while the clipped sub-trade is not.

Current evidence suggests the practical impact is limited to dust loss rather than a scalable extraction path: the effect appears only at extremely small clipped spends, and the observed loss so far is microscopic.

RECOMMENDATION

Re-check profitability after applying treasury or inventory clipping, rather than relying on the full-path profitability verdict for a smaller executed trade.

Alternatively, skip partial downward arbitrage below a minimum spend or minimum-output threshold that guarantees non-zero realizable value at targetPrice.

RESOLUTION

Status: Acknowledged.

INFORMATIONAL I-16

ACKNOWLEDGED

Tolerance reductions expose pending repricing surplus to searchers

CATEGORY	MEV / economic
LOCATION	packages/ush/contracts/hooks/ManifestHook.sol:setSwapPriceTolerance():L597-L603 packages/ush/contracts/hooks/ManifestHook.sol:_beforeSwap():L965-L970 packages/ush/contracts/hooks/ManifestHook.sol:_runArbitrage():L1198-L1221
REVIEW	Main Review

DESCRIPTION

Lowering swapPriceTolerance immediately makes a pre-existing below-target price eligible for downward arbitrage without reconciling the pool. A searcher can buy USH inside the outgoing deadband before the owner transaction, then sell after it. The sale's beforeSwap catch-up raises the pool before the searcher's fill, enabling a fee-net round-trip profit. The treasury arbitrage remains economically sound; the ordering diverts part of the surplus the protocol would otherwise capture. The PoC shows 43.36 USDe attacker profit and 58.74 USH protocol rewards, versus an unprofitable attacker round trip and 154.18 USH rewards when the reduction lands first.

Impact: A searcher can capture part of the arbitrage surplus the protocol would otherwise realize when narrowing an active tolerance band. This is foregone protocol revenue, not an undercollateralized treasury trade or principal loss. Exposure is bounded by the removed portion of the maximum 1% band and active liquidity, and arises only when the owner reduces a previously increased tolerance while a qualifying residual exists.

RECOMMENDATION

When lowering tolerance, require the current price to be within the new band, or stage the change through a paused/private atomic flow that reconciles the residual before swaps resume. Explicitly document the contestability if it is accepted policy.

RESOLUTION

Status: Acknowledged.

INFORMATIONAL I-17

RESOLVED

Consider using LiquidityMath.addDelta In SwapEstimation

CATEGORY	Math
LOCATION	`contracts/hooks/lib/SwapEstimation.sol:115,210`
REVIEW	Main Review

DESCRIPTION

`SwapEstimation` updates liquidity using:

```
liquidity = uint128(int128(liquidity) + liquidityNet);
```

This performs checked `int128` arithmetic, while V4 supports liquidity across the full `uint128` range and uses `LiquidityMath.addDelta`.

RECOMMENDATION

Use V4's canonical `LiquidityMath.addDelta(liquidity, liquidityNet)` in both estimators.

RESOLUTION

Status: Resolved.

INFORMATIONAL I-18 ACKNOWLEDGED

Document AuthManager Address Timelock Behavior

CATEGORY	Upgradeability
LOCATION	`contracts/hooks/ManifestHook.sol:setAuthManager`
REVIEW	Main Review

DESCRIPTION

`ManifestHook` timelocks updates to its `AuthManager` address, providing advance notice before the configured authorization contract is replaced. However, the configured `AuthManager` is a UUPS proxy whose implementation can be upgraded immediately by the `UPGRADER_ROLE` holder. An implementation upgrade can therefore change the hook's authorization behavior without changing the configured address or observing the hook's address-update timelock.

RECOMMENDATION

Document that the hook's timelock applies only to `AuthManager` address rotation and does not delay implementation upgrades or the resulting changes to authorization behavior.

RESOLUTION

Status: Acknowledged.

INFORMATIONAL I-19

ACKNOWLEDGED

NAV updates leave public target and flex views stale until a successful refresh

CATEGORY	stale state
LOCATION	packages/ush/contracts/oracle/USHPriceOracle.sol:updatePrice():L124-L154 packages/ush/contracts/hooks/ManifestHook.sol:calculateFlexBurnedToTarget():L675-L699 packages/ush/contracts/hooks/ManifestHook.sol:_refreshTargetPrice():L1519-L1522
REVIEW	Main Review

DESCRIPTION

USHPriceOracle.updatePrice() does not synchronize ManifestHook.targetPrice, while the public flex calculators walk toward that cached value. After a NAV update, those views can continue reporting the prior target and buffer requirement. A liquidity addition refreshes NAV in its callback and may then reject the add; if it reverts, the refresh rolls back and the public views remain stale.

Impact: Manifest-account automation that treats the cache-based flex view as an executable live-NAV quote can submit avoidably reverting liquidity additions until it obtains the live requirement or another successful operation refreshes the cache. Value-moving paths refresh before use, so the issue does not bypass the buffer invariant, affect existing liquidity, or cause asset loss.

RECOMMENDATION

Expose a live target/flex preview derived from ushPriceOracle.getCurrentPrice() and usdeUsdPeg, or add a permissionless target synchronization function.

RESOLUTION

Status: Acknowledged.

INFORMATIONAL I-20

ACKNOWLEDGED

Oracle-driven catch-up mints unbacked rewards from protocol-owned liquidity

CATEGORY	accounting / unbacked issuance
LOCATION	`packages/ush/contracts/hooks/ManifestHook.sol:_beforeSwap():L941-L970` `packages/ush/contracts/hooks/ManifestHook.sol:_runArbitrage():L1182-L1191` `packages/ush/contracts/hooks/lib/ArbExec.sol:executeUpwardArbitrage():L114-L149`
REVIEW	Main Review

DESCRIPTION

An in-band downward NAV update lowers `targetPrice` while the pool price is unchanged, leaving the pool above the new target with no trade having occurred. The next swap runs upward catch-up arbitrage, which mints USH, sells it into the pool for USDe, and mints the above-par surplus as reward USH to `rewardsWallet`. `executeUpwardArbitrage` treats `usdeReceived` as new backing, but since liquidity is gated to Manifest Accounts (`checkManifestAccount`), that USDe comes from the protocol's own LP.

The backing therefore holds only locally: the treasury receives USDe covering par plus the reward, but that USDe was already the protocol's own collateral, so consolidated protocol USDe (LP + treasury) is unchanged while a fresh reward liability is minted. The same collateral is double-counted, so at the protocol level the reward is effectively unbacked and the collateralization ratio dilutes. The downward-arbitrage path is symmetric (treasury buys USH cheap from its own LP and sends the surplus to rewards).

RECOMMENDATION

Clearly document and acknowledge this behavior: the backing invariant in `manifest-hook.md:1193` currently states unconditionally that the surplus backs the minted reward USH, but it holds only when `usdeReceived` is external inflow, not when the counterparty is protocol-owned liquidity on an oracle-only catch-up. Update that documentation to note the reused-collateral dilution as accepted, bounded behavior.

RESOLUTION

Status: Acknowledged.

INFORMATIONAL I-21

ACKNOWLEDGED

Informational Note for Users Regarding Trust Assumptions

CATEGORY Trust Assumptions
REVIEW Main Review

DESCRIPTION

The protocol is permissioned and operationally managed. Its safety, solvency, and availability depend on privileged roles, Safe signers, automation accounts, operators, oracle publishers, LPs, and external infrastructure.

Key trust assumptions include:

Governance and Safes: Privileged roles collectively can pause operations, manage roles and parameters, and upgrade several contracts. Each Safe controls only its assigned roles and assets, but its signer quorum can transact directly and is not restricted by module-level protections.

Hook owner: The owner can change fees, arbitrage settings, trusted routers, oracles, treasury and flex wallets, peg parameters, and override the circuit breaker.

Automation: Authorized bots can execute through enabled Safe modules without per-transaction signer approval. Production-configured Safes have broad USH minting and burning authority. A zero mint limit is unlimited, while a nonzero limit applies per call and does not prevent repeated calls or direct Safe transactions.

Treasury, flex wallet, and LPs: Downward arbitrage depends on the treasury retaining sufficient USDe and maintaining an allowance to the hook. Treasury actions can therefore affect backing and swap availability. The flex wallet must keep flex-issued USH segregated and use it only for its intended accounting purpose. Protocol LPs are trusted and retain control over liquidity, meaning pool depth and continuous swap availability are not guaranteed.

NAV oracle: NAV is a protocol-published valuation rather than a trustless market price. Authorized parties are trusted to calculate and publish it accurately and promptly. Privileged updates can bypass the ordinary timing and change limits, and the hook continues using the last NAV when it becomes old unless the oracle is explicitly paused.

I-21 Informational Note for Users Regarding Trust Assumptions CONTINUED

USDe peg and redemption: The USDe/USD oracle is primarily a circuit breaker and is not the live price used to calculate the target. The target instead relies on the administratively configured USDe peg, normally $1. Because the feed updates approximately daily, a real-world depeg may occur before it is reported onchain. During this interval, swaps, flex operations, and upward arbitrage may continue under the near-$1 assumption, potentially issuing USH against impaired USDe. Users therefore bear USDe solvency, custody, liquidity, redemption, and delayed-depeg-detection risk.

Deposits and redemptions: Final settlement is operator-assisted and lacks a user-defined minimum output, completion deadline, cancellation, or automatic refund. Cross-chain minAmountOut protection applies to transport, not the later operator-determined conversion.

Compliance and external dependencies: Access depends on managed whitelists, bans, and a sanctions oracle. Cross-chain settlement also relies on LayerZero, Circle CCTP, configured peers, and offchain attestations.

RECOMMENDATION

No protocol changes are required. This finding is intended solely to inform users of the protocol's privileged roles, operational dependencies, and external trust assumptions so they can evaluate the associated risks.

RESOLUTION

Status: Acknowledged.

LOW **L-01** ACKNOWLEDGED

Arb gas budget ignores EIP-150 call retention, so at the minimum `externalGasReserve` swaps from smart-wallet call stacks exceed the EIP-7825 gas limit cap

CATEGORY	Denial of Service
LOCATION	`contracts/hooks/lib/HookMath.sol:161`, `:178-184`; `contracts/hooks/ManifestHook.sol:128`, `:639-640`, `:1047`, `:1138`, `:1375`
REVIEW	Remediation Review

DESCRIPTION

`HookMath.legBudget` sizes each arb leg so that the gas the transaction has used so far, plus `externalGasReserve`, one view walk and the afterSwap tail, stays under `TX_GAS_CAP` (2^24). EIP-7825 caps the transaction gas limit, not gas used, and under EIP-150 every call frame between the sender and the hook withholds 1/64 of its gas from its callee. That withheld gas is never consumed, so it never shows up in `spent`, but it must fit inside the limit, adding roughly 1/63 of the work below each frame. Nothing in the budget covers it, and `setExternalGasReserve` accepts a reserve as low as 500,000. When a dense ladder of one-tick ranges with zero fee-growth-outside slots sits between a below-target price and target, both arb legs walk until the budget stops them, so every swap's gas used lands near `TX_GAS_CAP - externalGasReserve` whatever its size.

```
// ManifestHook.sol:128
uint256 private constant MIN_EXTERNAL_GAS_RESERVE = 500_000;

// HookMath.sol:178-184
function legBudget(uint256 spent, uint256 externalReserve, uint256 flags) external pure
  returns (uint256) {
    uint256 reserved = spent + externalReserve + VIEW_WALK_RESERVE_GAS + AFTER_TAIL_GAS;
    if (flags & FLAG_EXACT_OUTPUT != 0) reserved += VIEW_WALK_RESERVE_GAS;
    if (TX_GAS_CAP <= reserved) return 0;
    uint256 room = TX_GAS_CAP - reserved;
    return flags & FLAG_CATCH_UP != 0 ? room * CATCH_UP_SHARE_BPS / 10_000 : room;
}
```

At the 500,000 reserve on a 500-rung ladder, a 1e12 wei USDe buy uses about 15.0M gas at every call depth. From an EOA through the router it needs a 15,939,772 gas limit and fits, but behind an ERC-4337 EntryPoint, `innerHandleOp` and account frame it needs 16,798,635 (21,419 over 2^24), and behind six forwarding frames it needs 17,542,823 (765,607 over). Because the sizing never reads the caller's gas, the arb legs are identical at every depth

L-01 Arb gas budget ignores EIP-150 call retention, so at the minimum externalGasReserve swaps from smart-wallet call stacks exceed the EIP-7825 gas limit cap

CONTINUED

and any lower limit reverts instead of shrinking the arb, so the transaction cannot be included and `eth_estimateGas` returns a value no transaction can carry. Smart-wallet and aggregator users are locked out of the pool until an EOA-depth swap moves the price or the owner raises the reserve. Router-depth retention alone is 961,079 gas, nearly twice the 500,000 floor, so EOA swaps fit only because the per-segment gas constants overprice real tick crossings.

RECOMMENDATION

Add a retention term to `HookMath.legBudget` that reduces the room by 1/64 per call frame for a documented worst-case frame count (about four at router depth plus the deepest supported wallet stack), or raise `MIN_EXTERNAL_GAS_RESERVE` to at least 2,000,000 and document that the reserve must also cover this retention.

PROOF OF CONCEPT https://github.com/GuardianOrg/mirror-manifest-finance-ush-01a00a54/blob/pocs/test/poc/ArbGasBudgetIgnoresCallRetention.t.sol

RESOLUTION

Status: Acknowledged.

LOW **L-02** ACKNOWLEDGED

Later swaps in one transaction still run unbudgeted flex walks, so batched swaps exceed the EIP-7825 gas cap at the default `externalGasReserve`

CATEGORY	Denial of Service
LOCATION	`contracts/hooks/lib/HookMath.sol:178-184; contracts/hooks/ManifestHook.sol:1083, :1248, :1329; documentation/02-contracts/manifest-hook.md:1651-1653`
REVIEW	Remediation Review

DESCRIPTION

The arb gas ledger spans the whole transaction, and `HookMath.legBudget` keeps back one view walk and the afterSwap tail for the current swap only. On a dense ladder of one-tick ranges between a below-target price and target, the first swap's afterSwap leg therefore takes all the room up to `TX_GAS_CAP - externalGasReserve`. Every later swap of the pool in the same transaction gets a leg budget of zero and emits `ArbitrageSkippedForBudget`. But `_beforeSwap` still runs the advisory `_flexState` walk and `_afterSwap` still runs the `calculateFlexBurnedToTarget` reconcile walk. Each is a full 256-step walk, and neither is metered against the ledger.

```
// HookMath.sol:178-184
function legBudget(uint256 spent, uint256 externalReserve, uint256 flags) external pure
  returns (uint256) {
    uint256 reserved = spent + externalReserve + VIEW_WALK_RESERVE_GAS + AFTER_TAIL_GAS;
    if (flags & FLAG_EXACT_OUTPUT != 0) reserved += VIEW_WALK_RESERVE_GAS;
    if (TX_GAS_CAP <= reserved) return 0;
    uint256 room = TX_GAS_CAP - reserved;
    return flags & FLAG_CATCH_UP != 0 ? room * CATCH_UP_SHARE_BPS / 10_000 : room;
}

// ManifestHook.sol:1083, runs on every user swap whatever the leg budget
(bool flexOk, bool staleDirection, uint256 flexBalance, uint256 flexNeeded) =
  _flexState(key);
```

Each starved swap still costs 3.22M to 3.25M gas, about 98% of it the two flex walks, which is more than the whole 2,000,000 default reserve. On a 500-rung cold ladder at the default reserve, two 1e12 wei USDe buys in one transaction need a 17,344,756 gas limit with USH as currency0. Three buys need 20.68M with USH as currency0 (19.88M of gas used) and 19.68M with USH as currency1. On the client's own 240-rung ladder, three sells need 17,140,315 and 17,197,943. Raising the reserve only shifts the threshold: at the 8,000,000 maximum,

L-02 Later swaps in one transaction still run unbudgeted flex walks, so batched swaps exceed the EIP-7825 gas cap at the default `externalGasReserve` CONTINUED

four swaps need 17.65M to 17.95M, so no owner setting lets four swaps fit. A Universal Router call with several V4 swap actions on this pool, or a batched Safe transaction, cannot be included and `eth_estimateGas` returns more than 2^24. This contradicts the documentation, which states that a second swap "fits under the cap together with the first" and "rather than failing the transaction" reports `ArbitrageSkippedForBudget`.

RECOMMENDATION

Skip the advisory `_flexState` walk in `_beforeSwap` when the catch-up leg's budget is zero, and keep a transient count of this pool's user swaps that reverts with a named error before any walk once the count exceeds what the cap can hold. Correct the documentation to state how `externalGasReserve` must scale with the number of swaps a router batches.

PROOF OF CONCEPT https://github.com/GuardianOrg/mirror-manifest-finance-ush-01a00a54/blob/pocs/test/poc/LaterSwapsRunUnbudgetedFlexWalks.t.sol

RESOLUTION

Status: Acknowledged.

INFORMATIONAL **I-01** ACKNOWLEDGED

The arb's backward scan stops at a checkpoint that fails only on rounding, so a passing route near break-even is skipped or under-sized

CATEGORY	Logical Error
LOCATION	contracts/hooks/lib/SwapEstimation.sol:434-448, :457-503, :520-530; contracts/hooks/lib/HookMath.sol:203-214
REVIEW	Remediation Review

DESCRIPTION

When the walked route fails the backing check at its end, `walkWhileBacked` scans the checkpoints backward for the last one that passes or holds less than `MIN_PARTIAL_ROUTE`. If none qualifies, `seg` stays 0 and `_bisect` searches only the first segment. `_segmentAmounts` rounds both amounts up, so when break-even sits within about 2e-12 of the first tick boundary past the start price, the first checkpoint can hold at least 1e12, fail the check by 1 wei, and still have a positive true surplus. The marginal price after that checkpoint is still better than break-even, so later segments keep adding surplus. The scan reads the rounding failure as the point where the route turns, and never searches past it. The same miss exists on the sell side.

```
// SwapEstimation.sol:438-448
uint256 seg;
for (uint256 j = n - 1; j > 0; j--) {
    if (
        cps[j - 1].cumIn < BackingCheck.MIN_PARTIAL_ROUTE
            || _passes(buyingUsh, cps[j - 1].cumIn, cps[j - 1].cumOut, targetPrice, pFee)
    ) {
        seg = j;
        break;
    }
}
_bisect(r, cps, seg, zeroForOne, buyingUsh, targetPrice, pFee);
```

I-01 The arb's backward scan stops at a checkpoint that fails only on rounding, so a passing route near break-even is skipped or under-sized CONTINUED

On one book the catch-up leg emits `ArbitrageSkippedUnprofitable`, while a 1.33e19 wei route passes the backing check on real V4 execution by 6,742 wei. On another, the hook executes 1,364,700,737,648 wei, while a 2.6e19 wei route passes by 48,252 wei. The missed route is bounded by about 1e-12 of the pool's liquidity in input: at a liquidity of 1e27, that is about 0.001 tokens and 250 wei of surplus. The arb leaves the price further from target than the largest passing prefix allows, which is what the fallback exists to prevent.

RECOMMENDATION

Given the dust-scale amounts, acknowledge the behaviour and correct the design note in `Arb-zero-margin-fallback.md`, which states that above `MIN_PARTIAL_ROUTE` the aggregate turns from pass to fail at most once. Alternatively, pass the break-even sqrt price that `HookMath._atOrPastBreakEven` already computes into `walkWhileBacked`, and in the backward scan also accept as the anchor any checkpoint whose end price is still before break-even, so the bisection starts in the segment where the marginal price crosses it; widening the check by a rounding tolerance instead can turn a large passing route into a skip.

PROOF OF CONCEPT https://github.com/GuardianOrg/mirror-manifest-finance-ush-01a00a54/blob/pocs/test/poc/BackwardScanStopsAtRoundingFailure.t.sol

RESOLUTION

Status: Acknowledged.

INFORMATIONAL **I-02** RESOLVED

Stale `SafeModuleBase` comment after `RewardDistributionModule` addition

CATEGORY	Informational
LOCATION	`contracts/module/base/SafeModuleBase.sol:10`
REVIEW	Remediation Review

DESCRIPTION

The contract-level comment states that `PositionFeeModule` is the only current inheritor of `SafeModuleBase`: _"...PositionFeeModule is the only one that inherits it today..."_

However, `RewardDistributionModule` now inherits it as well and makes the comment stale.

RECOMMENDATION

Update the comment to include `RewardDistributionModule`.

RESOLUTION

Status: Resolved.

INFORMATIONAL I-03

RESOLVED

Setup documentation incorrectly requires REWARDER_ROLE for reserve-only distributions

CATEGORY	Documentation
LOCATION	documentation/02-contracts/reward-distribution-module.md:56-63
REVIEW	Remediation Review

DESCRIPTION

The documentation states that the module does nothing until both the module is enabled and the Safe receives `REWARDER_ROLE`. However, the role is required only when `stakerAmount` is nonzero; `distribute(0, reserveAmount)` works without it.

RECOMMENDATION

Clarify that module enablement is always required, while `REWARDER_ROLE` is required only for distributions containing a nonzero staker share.

RESOLUTION

Status: Resolved.

INFORMATIONAL I-04

RESOLVED

Consider validating that STAKED_USH.asset() matches USH

CATEGORY	Validation
LOCATION	contracts/module/RewardDistributionModule.sol:40-57,67-69
REVIEW	Remediation Review

DESCRIPTION

The constructor independently accepts `_ush` and `_stakedUSH`, performs zero address checks, but does not verify that the staking vault's underlying asset matches `_ush`. While this would be a deployment misconfiguration and would generally cause the first staker distribution to revert, consider having the check as a deployment hardening.

RECOMMENDATION

During construction, verify that `IERC4626(_stakedUSH).asset() == _ush` and revert otherwise.

RESOLUTION

Status: Resolved.

§ 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, or read more at **guardian.security**.

This report and its findings are also published at guardian.security/portfolio/manifest/hook-2026-11-26.