Diamond PAU

Prepared by: Octane Security **Date:** June 2, 2026 **Target:** Diamond PAU

1. Executive Summary

Octane Security conducted an Adversarial Research Engagement on **Diamond PAU**, the diamond-architecture rebuild of the Spark ALM Controller — a system that custodies DeFi treasury funds in an `ALMProxy` and routes relayer-initiated operations through a `Controller` that delegatecalls protocol-specific facets. The review covered the full controller surface — the `Beacon`→`Controller` dispatch and integration-sync path, every in-scope facet integration, both custody proxies (`ALMProxy`, `ALMProxyFreezable`), the rate-limit and access-control primitives, and the two UUPS-upgradeable side contracts (`OTCBuffer`, `WEETHModule`). Scope and the commit range are detailed in §2.1.

The engagement identified **36 findings**, grouped by theme — recurring patterns across the facet codebase rather than isolated bugs. **No critical- or high-severity findings were identified.** The four highest-impact issues are Medium-severity: a self-inflicted slippage-check revert on Uniswap V3 liquidity unwinds, an approval granted before a non-static quote call that lets a malicious or upgraded counterparty pull the just-granted allowance, an OTC readiness gate that admits concurrent in-flight swaps, and an async-redeem rate-limit that exchange-rate drift can outrun. Worst-case loss in each remains bounded by per-transaction rate limits and the freezer kill-switch — but the latter three each erode a rate-limit or invariant guarantee the documented trust model itself depends on. The controller's defense-in-depth — per-facet validation, double-keyed rate limits, and the freezer kill-switch — is consistently applied; most findings are missing-guard or accounting-precision gaps rather than breaks of a core invariant.

Severity breakdown:

Severity	Count
Critical	0
High	0
Medium	4
Low	24
Informational	8
Total	**36**

Remediation status (detail in §5): **10 Fixed** and **26 Acknowledged**.

Top findings:

`LayerZeroFacet.transfer` grants the destination OFT a full ERC-20 allowance from `ALMProxy` and then calls `quoteOFT` and `quoteSend` on that OFT via `IALMProxy.doCall` — a regular `CALL`, not a `STATICCALL`. A malicious or post-upgrade OFT can call `transferFrom` back on the asset from inside either quote call, draining the just-granted approval before `send()` ever executes. The Controller's `nonReentrant` guard does not block this, because the malicious call targets the ERC-20 directly rather than re-entering the Controller. The drain is bounded by the per-tx approved amount, but the loss is realized as an outright transfer to the attacker rather than as bridge slippage.

`ERC7540Facet.requestRedeem` charges the redeem rate-limit against a request-time conversion estimate, while `claimRedeem` pulls the vault's claimable amount with no claim-time reconciliation. On async vaults that settle at a drifting exchange rate, any claimed amount above the request-time estimate escapes the redeem rate-limit entirely — eroding the per-period exit cap the threat model relies on to bound a compromised relayer. There is no direct theft (assets still flow to the proxy), but unlike the self-inflicted revert in the Uniswap V3 finding below, this weakens a primary defense rather than a peripheral one.

`OTCFacet`'s `isSwapReady` test admits time-based recharge, so the documented "one outstanding swap per exchange" invariant breaks whenever the prior swap's claim has not yet settled but enough wall-clock time has passed. A compromised relayer can cycle `send()` calls as readiness regenerates, accumulating concurrent off-custody exposure well beyond the single-swap amount the design intends to cap.

`UniswapV3Facet.removeLiquidity` deterministically reverts on positions with material accrued fees, with no attacker required. It constrains `decreaseLiquidity` against a principal-only minimum but then post-checks the combined principal-plus-fees balance delta against the same slippage bound, so the two become simultaneously unsatisfiable once accrued fees grow past a small fraction of principal. There is no separate `collect()` path to drain fees first, and the operational workaround — lowering `maxSlippage` — directly undermines slippage protection.

Findings by theme:

- **AMM accounting and slippage (6)** — `UniswapV3Facet.removeLiquidity` fee-inclusive slippage post-check and withdraw rate-limit, non-atomic tick bounds, `UniswapV3Utils.consult` divide-by-zero, and V4 hookless-pool / `maxSlippage`-gate gaps
- **Async vault & withdrawal lifecycle (6)** — request-time pre-charge with no claim-time reconciliation (ERC-7540, Maple), missing claim-key preflight, the wrong `mint` overload, and the WEETH withdrawal-NFT / stray-token binding gaps
- **Cross-chain destination integrity (6)** — LayerZero approval-before-quote drain, missing peer binding, `minAmountLD` floor and refund-revert; CCTP fee-ratio and last-chunk revert
- **OTC swap state-machine integrity (4)** — time-based readiness admits concurrent in-flight swaps, buffer rotation strands proceeds, Controller replacement resets pending state, `setBuffer` lacks a proxy-compat check

- **Upgrade & deployment-order hazards (2)** — unprotected UUPS `initialize()` on `WEETHModule`/`OTCBuffer`, and the `IEnumerableIntegrations` struct-layout coupling with no schema-version handshake
- **Rate-limit key dependency gaps (2)** — MapleFacet missing cancel-key precondition (§4.18), and the "Bespoke rate limits" refactor doc drift (§4.26, commit `0f3a1ef`)
- **Single-facet accounting / slippage gaps (9)** — Farm reward decrement/exit-coupling and the Farm withdraw budget charged on the relayer-supplied amount rather than the measured balance delta, Curve zero-rate slippage bypass, Superstate and ERC-4626 missing exit-rate floors, the deposit-side `maxExchangeRate` blind spot to Maple impairment, Basin missing exchange-rate guard, the ERC-7540 async deposit/claim missing min-out/exchange-rate floor, Pendle `minTokenOut` scaling
- **Strict decode of mutable-external returns (1)** — brittle `abi.decode` of Lido/EtherFi/Ethena returns in the `requestWithdraw`/cooldown paths DoSes on any upstream ABI change (the LayerZero quote-decode in the cross-chain theme shares the pattern)

---

2. Scope and Methodology

2.1 Target Scope

The engagement covered the core PAU contracts under `src/` at the time of analysis (across `dc54647` / `v1.12.0-rc.0`, `fc2fe7f` / `v1.12.0-rc.1`, `ebcc56d` / `v1.12-rc.2`, `c741e32` / `v1.12.0-rc.3`, and `a84fe96` / `v1.12.0-rc.4`). Release candidate `v1.12.0-rc.4` (`a84fe96`) was subsequently promoted unchanged to the final release tag **`v1.12.0`**.

Core contracts in scope:

Contract	Role
`ALMProxy`	Holds all funds; only `CONTROLLER` role may `doCall` / `doCallWithValue` / `doDelegateCall`
`ALMProxyFreezable`	Lower-privilege proxy variant; `RELAYER` calls directly, `FREEZER` can revoke
`Controller`	Dispatch-routing fallback; admin-synced selector → (facet, delegateSelector) map
`ControllerSharedStorage`	ERC-7201 shared slot used by Controller and every Facet
`Beacon`	Canonical integration registry; admin wires selectors, protects hardcoded ones
`AccessControls`	Roles: `DEFAULT_ADMIN_ROLE`, `RELAYER_ROLE`, `FREEZER_ROLE`
`RateLimits`	keccak-keyed linear-recharge limits; only `CONTROLLER` may trigger decrement/increment
`PAUFactory`	Atomic deploy of ALMProxy + RateLimits + AccessControls + Controller
`Facets/*`	Per-protocol logic delegatecalled by Controller; each owns ERC-7201 storage
`OTCBuffer`	UUPS-upgradeable separate deployment holding OTC settlement balances
`WEETHModule`	UUPS-upgradeable, holds the EtherFi withdrawal NFT

The engagement covered every facet integration under `src/facets/`. Facets that do not appear in Section 4 were reviewed and yielded no findings.

2.2 Methodology

The engagement combined manual review with multiple targeted automated scans of the entire codebase. Each scan was seeded with custom issue types specific to PAU's architecture, so the automated passes hunted for the patterns most relevant to this system in addition to standard smart contract flaws. Candidate findings from every pass were manually triaged against the trust model documented in `docs/THREAT_MODEL.md` and `docs/SECURITY.md`: each was manually validated against the latest source and assigned a severity calibrated to that model. Findings that treat max-slippage as a tolerance, require a trusted relayer, or require admin misconfiguration the trust model places out of scope were filtered out, along with informational-only observations without operational consequence. Prior audit work was ingested as scope context.

---

3. Findings Summary

The following findings were identified in commit `fc2fe7f` (tagged `v1.12.0-rc.1`).

#	Title	Severity
1	Fee-inclusive slippage post-check in `UniswapV3Facet.removeLiquidity` deterministically reverts on positions with accrued fees	Medium
2	Approval granted before non-static `quoteOFT`/`quoteSend` in `LayerZeroFacet.transfer` allows OFT-side drain of just-approved allowance	Medium
3	Time-based recharge in `OTCFacet.isSwapReady` allows multiple concurrent in-flight swaps, breaking the one-outstanding-swap-per-exchange invariant	Medium
4	Request-time pre-charge without claim-time reconciliation in `ERC7540Facet` bypasses redeem rate-limit on exchange-rate drift	Medium
5	No fee-to-amount ratio bound in `CCTPFacet.transferWithFee` and per-chunk `maxFee` reuse across burn-limit splits	Low

#	Title	Severity
6	Unprotected UUPS `initialize()` on `WEETHModule` and `OTCBuffer` if proxy is not atomically initialized at deployment	Low
7	`LayerZeroFacet.transfer` sets `minAmountLD` to OFT-quoted value with no operator-configured ratio floor; strict `abi.decode` of quote returns can DoS the bridging path	Low
8	Hard-required ETH refund in `LayerZeroFacet.transfer` reverts entire bridge for contract relayers with non-payable fallback	Low
9	Fee-inclusive withdraw rate-limit accounting in `UniswapV3Facet.removeLiquidity` charges fees against principal-only budget	Low
10	Non-atomic `setLiquidityLowerTickBound` / `setLiquidityUpperTickBound` updates in `UniswapV3Facet` open transient out-of-policy mint window	Low
11	Unconditional `÷ secondsPerLiquidityCumulativesDelta` in `UniswapV3Utils.consult` reverts when zero in-range liquidity in TWAP window; result is discarded by callers	Low
12	Missing `poolKey.hooks == address(0)` enforcement in `UniswapV4Facet` allows hook-driven balance-delta manipulation that undercharges deposit rate limits	Low
13	Missing `maxSlippage != 0` gate on `UniswapV4Facet` liquidity operations conflicts with `CODE_NOTES.md`'s disable-by-zeroing-slippage convention	Low
14	`OTCFacet` does not snapshot the active buffer at `send()`; admin rotation between send and claim orphans settlement proceeds at the old buffer	Low
15	OTC pending-swap state lives in Controller's ERC-7201 slot — Controller replacement zeroes pending state	Low
16	`OTCFacet.setBuffer` does not verify the new buffer's stored `proxy` matches the Controller's ALMProxy	Low
17	Preview-based rate-limit charging in `MapleFacet.requestRedemption` (`convertToAssets(shares)`) without claim-time reconciliation	Low
18	`MapleFacet.requestRedemption` does not require the cancel key to exist; relayer can enqueue an irreversible request	Low
19	`ERC7540Facet.requestDeposit`/`requestRedeem` do not require the corresponding claim keys to exist	Low
20	`ERC7540Facet.claimDeposit` calls 2-arg `mint(shares, receiver)`, bricking claims on vaults that require 3-arg `mint(shares, receiver, controller)`	Low
21	`WEETHModule.claimWithdrawal` re-derives the EtherFi `WithdrawRequestNFT` at claim time from a live pointer chain; rotation of the NFT contract bricks claims for in-flight requests	Low
22	`FarmFacet.claimReward` gate-checks but does not decrement; `FarmFacet.withdraw` calls `_claimReward` unconditionally	Low
23	`SuperstateFacet.subscribe` has no min-out and no governance-configured exchange-rate floor	Low
24	`ERC4626Facet.redeem` has no governance exit-rate floor	Low
25	Missing remote-peer (`peers[dstEid]`) binding in `LayerZeroFacet.transfer` allows misroute on external OFT peer rotation	Informational
26	Documentation drift from the "Bespoke rate limits" refactor	Informational
27	`FarmFacet.withdraw` charges the relayer-supplied amount against the withdraw budget rather than the measured staking-token balance delta	Informational
28	Unchecked zero `stored_rates` in `CurveFacet._validateSwapMinAmountOut`	Informational
29	Last burn-limit chunk in `CCTPFacet.transferWithFee` reverts when remainder ≤ `maxFee`	Informational
30	No admin sweep on `WEETHModule` — WETH accidentally routed to the module is stranded until UUPS upgrade	Informational
31	`IEnumerableIntegrations` struct-layout coupling between `Beacon`, `Controller`, and facets has no schema-version handshake	Informational

The following additional findings were identified in later release candidates (`c741e32` / `v1.12.0-rc.3` and `a84fe96` / `v1.12.0-rc.4`).

#	Title	Severity
32	`BasinFacet` deposit/withdraw lack a governance-configured exchange-rate guard, leaving conversion bounds entirely to the compromisable allocator	Low
33	`PendleFacet.redeem` computes router `minTokenOut` with a hardcoded `1e18` scale, deterministically reverting redemption on non-18-decimal yield-token markets	Low
34	`ERC4626Facet.deposit`'s `maxExchangeRate` ceiling does not model Maple's impairment asymmetry, so deposits into an impaired pool socialize pre-existing losses onto the proxy	Low
35	`ERC7540Facet` async deposit/claim has no min-out or governance-configured exchange-rate floor, accepting unfavorable async settlements	Low

#	Title	Severity
36	Strict `abi.decode` of mutable-external return data in `WSTETHFacet`/`WEETHFacet` `requestWithdraw` and `EthenaFacet` cooldowns DoSes the path on any upstream return-shape change	Informational

4. Detailed Findings

> Remediation status for every finding is consolidated in **§5 (Remediation Status)**.

4.1: Fee-inclusive slippage post-check in `UniswapV3Facet.removeLiquidity` deterministically reverts on positions with accrued fees

Severity: Medium

What happens

`UniswapV3Facet.removeLiquidity` calls `INonfungiblePositionManager.decreaseLiquidity` — which Uniswap enforces against principal-only `amount0Min` / `amount1Min` constraints — then calls `collect(type(uint128).max)` to sweep both principal *and* all accrued fees out of the position. After the sweep the facet measures the post-call balance delta of the proxy and asserts `min ≥ (principal + fees) × maxSlippage / 1e18`. The first check (passed to Uniswap) compares against principal only; the post-check compares against principal-plus-fees. The two are simultaneously satisfiable only when `fees / principal ≤ (1 − S) / S`, where `S` is `maxSlippage / 1e18` — roughly 2% of principal at `S = 0.98e18`, 1% at `S = 0.99e18`.

For any in-range position that has accumulated material fees, the call deterministically reverts. There is no separate `collect()` entry point on the facet to drain fees first. Operator workarounds (lowering `maxSlippage` to admit the larger ratio) directly weaken the slippage protection that is supposed to defend the position from MEV.

Root cause

- `UniswapV3Facet.removeLiquidity` (~`src/facets/uniswap-v3/UniswapV3Facet.sol:417-463`) — function body
- `_callDecreaseLiquidity` (~`UniswapV3Facet.sol:538`) — passes `amount0Min`/`amount1Min` derived from principal only to `decreaseLiquidity`
- `_checkSlippage` (~`UniswapV3Facet.sol:938`) — post-call check uses the full balance delta after `collect(type(uint128).max)`
- No standalone `collect`-only entry point — fees cannot be drained before the slippage check

Impact boundary

What is possible:

- Liveness DoS on `removeLiquidity` for any position whose accrued fees exceed `principal × (1 − S) / S`
- Triggered by normal pool usage — no attacker required, no external manipulation needed
- Forces operators to either skip removeLiquidity entirely or to relax `maxSlippage` (weakening MEV protection on the principal portion)

What is not possible:

- Fund loss — the call reverts atomically; positions remain intact and re-callable
- Bypass of slippage on principal — the Uniswap-side `amount0Min` check still applies on subsequent attempts

Reproduction recommendation

Open a position, advance time and route a few swaps through it to accrue fees, then attempt `removeLiquidity` with operationally reasonable `maxSlippage` (≥ 0.98e18). The call reverts; lowering `maxSlippage` to 0.95e18 admits the unwind but leaves the principal exposed to a 5% MEV haircut.

4.2: Approval granted before non-static `quoteOFT`/`quoteSend` in `LayerZeroFacet.transfer` allows OFT-side drain of just-approved allowance

Severity: Medium

What happens

When `approvalRequired(asset, oft) == true`, `LayerZeroFacet.transfer` first calls `IALMProxy.doCall(asset, abi.encodeCall(IERC20.approve, (oft, amount)))` to grant the OFT a full allowance from the proxy, then invokes `oft.quoteOFT(...)` and `oft.quoteSend(...)` via `IALMProxy.doCall` — a regular `CALL`, not a `STATICCALL`. From inside either quote call, the OFT contract can synchronously invoke `asset.transferFrom(ALMProxy, attacker, amount)` and exhaust the approval before `oft.send()` ever executes. The Controller's `nonReentrant` guard does not block this because the malicious call targets the ERC-20 directly, not the Controller.

The drain is bounded by the per-tx approved amount (the rate-limit decrement). A stale-allowance follow-on exists in the partial-fill case: if the OFT's `send` debits less than the approved `amount` (e.g. fee-adjusted), the residual allowance persists until the next `transfer()` overrides it via `ApproveLib`'s zero-then-set. A reverted `send` does **not** leave stale allowance — the entire tx reverts atomically, including the `approve`.

Root cause

- `LayerZeroFacet.transfer` approval site (~`src/facets/layer-zero/LayerZeroFacet.sol:180`) — full-amount `approve` to OFT before any quote call
- `LayerZeroFacet.transfer` quote call sites (~`LayerZeroFacet.sol:196, 206`) — both use `IALMProxy.doCall` (state-mutating `CALL`)
- The OFT is a whitelisted-but-mutable counterparty; LayerZero v2 explicitly supports OFT contract upgrades
- `IALMProxy` exposes no `doStaticCall` variant (`src/interfaces/IALMProxy.sol`, `src/ALMProxy.sol` declare only `doCall`/`doCallWithValue`/`doDelegateCall`) — the only escape today is a direct `staticcall` outside the proxy, as `UniswapV3Facet._getPosition` does at `src/facets/uniswap-v3/UniswapV3Facet.sol:964`. The unsafe approve-then-non-static-call pattern is therefore structural to facets that route view-shaped calls through the proxy, not a local selection error.

Impact boundary

What is possible:

- A malicious or post-upgrade OFT drains `amount` of `asset` from the proxy in a single `transfer()` invocation
- The drained amount is exactly the just-approved value; the rate-limit decrement still applies, so the per-period bound holds *for that key*
- Partial-fill stale-allowance follow-on: if `send` consumes less than `amount`, the residual `allowance(proxy, oft)` survives until the next `transfer()` resets it via `ApproveLib`
- Loss is bounded by the configured rate limit per period, but the loss is realized as an outright transfer to attacker rather than as bridge slippage

What is not possible:

- Re-entry into the Controller — `nonReentrant` blocks that path
- Drain beyond the approved amount in a single call

Attack economics:

- Capital required: 0 (attacker controls a whitelisted OFT or its upgrade)
- Transactions: 1 (victim's `transfer()` call)
- Timing: Any time after governance whitelists the malicious or compromised OFT
- Net profit/loss to attacker: Approved amount per call (≈ rate-limit budget per period)
- Who can trigger: The OFT contract itself; in practice, anyone who controls the OFT or who can persuade governance to whitelist a malicious one

Reproduction recommendation

Deploy a mock OFT whose `quoteOFT` calls `IERC20(asset).transferFrom(proxy, attacker, IERC20(asset).allowance(proxy, address(this)))` synchronously, whitelist it, invoke `LayerZeroFacet.transfer`, observe drain.

4.3: Time-based recharge in `OTCFacet.isSwapReady` allows multiple concurrent in-flight swaps, breaking the one-outstanding-swap-per-exchange invariant

Severity: Medium

What happens

`OTCFacet.getIsSwapReady(exchange)` returns true when `normalizedClaimed + (block.timestamp − sentTimestamp) × normalizedRate ≥ normalizedSent × maxSlippage / 1e18` (`src/facets/otc/OTCFacet.sol:257-260`). The time-based recharge term means readiness can return true even when `normalizedClaimed` is unchanged — before any claim has actually settled the prior outstanding swap. `send()` checks `getIsSwapReady` and proceeds, overwriting the prior state in place (`src/facets/otc/OTCFacet.sol:160-162`), allowing a fresh `send()` to issue while a prior swap is still pending settlement.

The documented OTC design intends one outstanding swap per exchange at a time. Specifically:

- `docs/LIQUIDITY_OPERATIONS.md:121-124` — "The contract prevents sending more funds until the required balance is returned… This provides guarantees that at most `X` can be at risk per whitelisted OTC route."
- `docs/THREAT_MODEL.md:89` — "OTC Desks: Assumed to complete trades; **max loss bounded by single swap amount**."
- `docs/LIQUIDITY_OPERATIONS.md:142` — recharge is documented as a **partial-settlement unbricker** for when "an exchange returns an amount of funds that is materially below the configured `maxSlippage`", not as a replacement for actual settlement.

A compromised relayer can cycle `send()` calls as readiness regenerates from time-based recharge alone, with each call overwriting state. The documented "max loss bounded by single swap amount" invariant is broken: realized loss on exchange default scales with `N` outstanding swaps, where `N` is bounded above by the per-period `LIMIT_OTC_SWAP[exchange]` budget divided by per-send amount, not by 1.

A second-order bookkeeping defect emerges as soon as concurrent swaps exist: `claim()` at `src/facets/otc/OTCFacet.sol:189` reads the full buffer balance and credits it to the **current** state's `normalizedClaimed`, with no association to which `send()` the buffer balance settled. A settlement of an *older* swap therefore retroactively credits the *latest* `send()` — immediately satisfying its readiness gate via real claim rather than recharge — and lets `send()` fire again with no actual settlement of the latest position. The state machine becomes degenerate once it leaves the one-at-a-time regime.

Root cause

- `OTCFacet.getIsSwapReady` formula at `src/facets/otc/OTCFacet.sol:257–260` — time-based recharge term added to `normalizedClaimed`
- `OTCFacet.send` at `:156` consults only `getIsSwapReady`; no separate "no swap currently pending" gate, and `:160–162` overwrites prior state unconditionally
- `OTCFacet.claim` at `:189` credits buffer balance to the current state's `normalizedClaimed` with no association to which `send()` the balance settled — a settlement-aliasing defect that compounds once concurrent swaps exist
- Time-based recharge is documented as a partial-settlement unbricker (`docs/LIQUIDITY_OPERATIONS.md:142`) but the code uses it as a substitute for any settlement at all
- The two semantics need separation — readiness for a new swap should require both rate-limit availability *and* prior-swap completion (a separate `settled` flag set in `claim()`)

Impact boundary

What is possible:

- A compromised relayer issues multiple concurrent swaps to the same exchange, each individually within the per-tx rate-limit
- The documented `THREAT_MODEL.md:89` bound ("max loss bounded by single swap amount") is broken; realized loss on exchange default scales with `N` outstanding swaps, where `N` is bounded above by the per-period `LIMIT_OTC_SWAP[exchange]` budget divided by per-send amount
- Settlement of an older swap retroactively credits the *latest* state via the `claim()` aliasing at `:189`, satisfying the readiness gate by real claim and re-enabling another `send()` — the state machine progresses on stale settlement data

What is not possible:

- Bypass of the per-tx rate-limit decrement — each `send()` still draws from `LIMIT_OTC_SWAP[exchange]`
- Loss with a fully honest exchange — all sent funds eventually return; the bound that breaks is *max instantaneous exposure*, not solvency
- Theft beyond what the destination exchange defaults on

Attack economics:

- Capital required: 0 (compromised relayer)
- Transactions: N (one per swap, paced by the time-based recharge interval)
- Timing: After `(normalizedSent × maxSlippage / 1e18) / normalizedRate` seconds per cycle (the recharge interval)
- Net profit/loss to attacker: Conditional on exchange default — realized loss is `N × singleSwap` rather than the documented `1 × singleSwap`
- Who can trigger: Compromised relayer; FREEZER can evict but only after detection

Reproduction recommendation

With `normalizedRate > 0` configured, initiate a `send()`, advance time past the recharge interval without calling `claim`, attempt a second `send()` — it succeeds. Observe two pending swaps in storage.

4.4: Request-time pre-charge without claim-time reconciliation in `ERC7540Facet` bypasses redeem rate-limit on exchange-rate drift

Severity: Medium

What happens

`ERC7540Facet.requestRedeem` charges `LIMIT_7540_REDEEM` using `convertToAssets(shares)` evaluated at request time. `claimRedeem` only verifies the corresponding key exists, then calls `vault.maxWithdraw(proxy)` and pulls that amount — with no rate-limit accounting and no reconciliation against the request-time pre-charge. Async ERC-7540 vaults settle requests over multiple blocks at fluctuating exchange rates; the claimed asset amount routinely differs from the request-time `convertToAssets` value.

When claim exceeds pre-charge, the excess bypasses the redeem rate-limit entirely — the per-period exit cap is broken by the magnitude of fulfillment-window drift. When claim is less than pre-charge, the bucket is over-consumed by the shortfall until `RateLimits`' linear recharge restores it — operational drag, not a security-boundary failure. The excess direction is the security-relevant one.

A separate observation in the same facet: `LIMIT_7540_CLAIM_REDEEM` is referenced only via `_requireRateLimitExists` (`ERC7540Facet.sol:123`), never decremented. It functions as a presence-gated feature toggle, not an enforced budget — worth flagging as a dead rate-limit key whose configured `maxAmount`/`slope` have no effect.

Root cause

- `ERC7540Facet.requestRedeem` (`src/facets/erc7540/ERC7540Facet.sol`) — pre-charge via `convertToAssets(shares)`
- `ERC7540Facet.claimRedeem` — reads `vault.maxWithdraw(proxy)` and pulls without rate-limit accounting
- No state field stores the request-time pre-charge against which the claim could reconcile
- `MapleFacet.requestRedemption` exhibits the same pattern at lower severity (separate finding, 4.17)

Impact boundary

What is possible:

- Excess claim bypasses the redeem rate-limit (gap = `claimedAssets − convertToAssets(shares)` at request time) — the security-relevant direction
- Shortfall claim over-consumes the bucket by `convertToAssets(shares)_request − claimedAssets` until `RateLimits`' linear recharge restores it — operational, not a security boundary failure
- Excess-direction gap compounds over many requests in volatile-yield environments
- Defeats the rate-limit's purpose of bounding per-period redemption flows

What is not possible:

- Direct fund theft — assets still flow into the proxy
- Bypass of the request-time pre-charge — that decrement still applies

Attack economics:

- Capital required: 0 (compromised relayer; attack uses normal request/claim flow)
- Transactions: 2 per request (request, claim) — both permissionless to relayer
- Timing: Exchange-rate drift between request and claim — natural occurrence in async vaults
- Net profit/loss to attacker: Indirect — defeats rate-limit boundedness assumption
- Who can trigger: Compromised relayer

Reproduction recommendation

On any ERC-7540 vault with non-trivial yield accrual, request a redemption at exchange rate `R0`, advance time so the rate becomes `R1 > R0`, claim, observe `claimedAssets > convertToAssets(shares, R0)` and no rate-limit decrement on the gap.

4.5: No fee-to-amount ratio bound in `CCTPFacet.transferWithFee` and per-chunk `maxFee` reuse across burn-limit splits

Severity: Low

What happens

`CCTPFacet._transfer` splits a transfer into chunks of size `burnLimitsPerMessage(asset)` and reuses the relayer-supplied `maxFee` unchanged for every chunk. The only constraint on `maxFee` is an absolute admin-set `maxFeeCap` and the per-chunk `require(maxFee < transferAmount)`. There is no per-chunk or aggregate fee-to-amount ratio bound. The rate-limit decrement is taken on the gross bridged amount, with no accounting for fees absorbed at the CCTP layer.

`maxFee` is a ceiling on the per-message fee — Circle's fee oracle sets the **realized** fee, capped by `maxFee`. So a compromised relayer alone cannot directly drain funds via fees; what they can do is raise the ceiling such that, under any adverse condition where Circle's oracle returns a value close to `maxFee` (network congestion, fee-policy change, oracle anomaly), the protocol pays out aggregate fees scaled by `ceil(amount / burnLimit) × maxFee` instead of the bps-level fee a single-message transfer would see. The realized loss path is therefore two-step: aggressive relayer-chosen `maxFee` combined with adverse fee-oracle behavior. Even in normal operation, the rate-limit decrement is fee-inclusive (gross `amount`), so the budget treats fees as bridged value — accounting drift that is independent of any attack.

Root cause

- `CCTPFacet._transfer` while-loop (~`src/facets/cctp/CCTPFacet.sol:177–225`) — chunked split with reused `maxFee`
- Per-chunk gate: `require(maxFee < transferAmount)` — admits arbitrary `maxFee` up to `transferAmount − 1`
- Rate-limit decrement applied to gross amount on entry, not to net amount delivered
- No `maxFeeBps` or `maxAggregateFee` parameter

Impact boundary

What is possible (conditional on Circle fee-oracle behavior):

- Under adverse fee-oracle conditions (congestion, policy change, oracle compromise), a compromised relayer can structure transfers so that aggregate fees — bounded above by `ceil(amount / burnLimit) × maxFee` — consume a large fraction of the period's rate-limit budget
- Loss is realized as CCTP fee paid to Circle's fee recipient, not as transfer to an attacker
- Independent of any attack: rate-limit decrement is fee-inclusive (gross `amount`), so the budget over-counts delivered value by exactly the realized fee — a steady-state accounting gap

What is not possible:

- Direct theft of bridge funds — Circle controls the fee recipient
- A relayer-only drain via fees: the realized fee is set by Circle's oracle and capped by `maxFee`; the relayer raises the ceiling, but cannot force Circle to charge it
- Bypass of the absolute admin-set `maxFeeCap`

4.6: Unprotected UUPS `initialize()` on `WEETHModule` and `OTCBuffer` if proxy is not atomically initialized at deployment

Severity: Low

What happens

`WEETHModule.initialize` and `OTCBuffer.initialize` are external, permissionless on first call (idempotent under `Initializable`), and set both `DEFAULT_ADMIN_ROLE` and the internal `proxy` storage to caller-supplied values. If the UUPS proxy is deployed without atomic initialization (i.e. `initialize()` is not invoked in the same transaction as the proxy deployment, e.g. via `ERC1967Proxy(impl, initData)` constructor calldata), an attacker can front-run the legitimate `initialize()` call, seize admin and set `proxy` to an attacker address.

For `WEETHModule`, hijack does more than expose in-flight withdrawals at the moment of compromise. The EtherFi `LiquidityPool.requestWithdraw` flow mints the withdrawal NFT directly to the module address regardless of who controls `storage.proxy`, and `claimWithdrawal` authorizes solely on `msg.sender == storage.proxy` (`WEETHModule.sol:120`) — transferring the wrapped ETH to `msg.sender` on success. After hijack, every future legitimate withdrawal the protocol initiates lands at the module and can only be claimed by the attacker, while the legitimate `ALMProxy` is permanently locked out (its `msg.sender` no longer matches the hijacked `storage.proxy`). The seized `DEFAULT_ADMIN_ROLE` also unlocks `_authorizeUpgrade` (`WEETHModule.sol:193`), giving the attacker arbitrary UUPS upgrade authority on the module.

For `OTCBuffer`, hijack of admin gives access to `approve(asset, allowance)` (`OTCBuffer.sol:75`), which `forceApprove`s the stored proxy (= attacker) for any asset that lands at the buffer. The seized admin also unlocks `_authorizeUpgrade` (`OTCBuffer.sol:111`), so the buffer can be upgraded to arbitrary draining code regardless of the approve path.

The risk is exclusively a deployment-order hazard. Atomic proxy initialization (constructor calldata) eliminates it.

Root cause

- `WEETHModule.initialize` / `OTCBuffer.initialize` — external and permissionless on the proxy's first call. Both implementations *do* call `_disableInitializers()` in their constructors (`WEETHModule.sol:94`, `OTCBuffer.sol:50`), which locks the implementation contract against direct initialization; the residual risk is narrowly the **proxy's** first `initialize()` being front-run if the proxy is deployed without atomic init calldata
- `claimWithdrawal` authorizes solely on `msg.sender == proxy`
- No deployment script for these UUPS modules is in scope — `PAUFactory` deploys only ALMProxy/RateLimits/AccessControls/Controller, not the modules — so atomic proxy initialization must be verified against the actual deployment

Impact boundary

What is possible (if proxy not atomically initialized):

- Front-run `initialize()` between proxy deploy and legitimate init transaction
- Set `proxy = attacker` and `admin = attacker`
- Drain all WETH proceeds claimable through `WEETHModule.claimWithdrawal` for in-flight withdrawals
- Drain settlement tokens from `OTCBuffer` via attacker-authored approve/transfer paths

What is not possible:

- Exploitation if proxy is initialized atomically in its constructor (industry-standard UUPS pattern)
- Re-initialization after legitimate init — `Initializable` blocks subsequent calls

---

4.7: `LayerZeroFacet.transfer` sets `minAmountLD` to OFT-quoted value with no operator-configured ratio floor; strict `abi.decode` of quote returns can DoS the bridging path

Severity: Low

What happens

`LayerZeroFacet.transfer` sets `sendParams.minAmountLD = receipt.amountReceivedLD` — the value returned by `oft.quoteOFT()`. There is no governance-configured floor comparing `amountReceivedLD` to the source `amount`, so any haircut applied at the OFT (token-side fees, rebasing, intermediate hops) is silently accepted. The loss per call is bounded by the configured rate limit but is not bounded by an operator-set haircut tolerance.

As a related sub-issue on the same call site, both `quoteOFT` and `quoteSend` returns are decoded via strict `abi.decode` with no length guard. If an OFT is upgraded to return a different ABI shape, the decode reverts atomically and the bridging path DoSes for that route until the facet is upgraded.

Root cause

- `LayerZeroFacet.transfer` minAmountLD assignment (~`LayerZeroFacet.sol:194–203`) — `sendParams.minAmountLD = receipt.amountReceivedLD`
- `LayerZeroFacet.transfer` decode sites (~`LayerZeroFacet.sol:195–211`) — strict `abi.decode` of both quote returns
- No `maxHaircutBps` or equivalent governance parameter

Impact boundary

What is possible:

- Silent acceptance of OFT-side haircuts up to the rate-limit budget per period
- Bridging-path liveness DoS if an OFT changes its return ABI

What is not possible:

- Loss exceeding the rate-limit budget per period
- Cross-route contagion — DoS is per-OFT/per-route

4.8: Hard-required ETH refund in `LayerZeroFacet.transfer` reverts entire bridge for contract relayers with non-payable fallback

Severity: Low

What happens

After paying the LayerZero native fee, `LayerZeroFacet.transfer` attempts to refund any surplus `msg.value` to `msg.sender` via `msg.sender.call{value: refund}("")` and hard-requires `success`. A contract relayer with a non-payable fallback (or with a payable fallback that re-enters the Controller — blocked by `nonReentrant`) reverts the entire bridge transaction whenever the relayer overpays.

No funds are lost — revert atomicity returns the over-payment. The impact is operational liveness: contract-based relayers cannot use this code path with any over-payment.

Root cause

- `LayerZeroFacet.transfer` refund block (~`LayerZeroFacet.sol:219–226`) — `msg.sender.call{value: refund}("")` + `require(success)`
- No alternative refund recipient parameter

Impact boundary

What is possible:

- Contract relayer with non-payable fallback cannot bridge with any over-payment
- Force-precise-fee operational requirement on contract relayers

What is not possible:

- Fund loss
- Effect on EOA relayers (always payable)

4.9: Fee-inclusive withdraw rate-limit accounting in `UniswapV3Facet.removeLiquidity` charges fees against principal-only budget

Severity: Low

What happens

After `collect(type(uint128).max)` in `removeLiquidity`, `valueWithdrawn` is computed from the full post-collect balance delta — which includes both principal and accrued fees — and is charged against `LIMIT_UNISWAP_V3_WITHDRAW`. Project documentation explicitly contradicts this. From `docs/UNIV3_UNIV4_COMPARISON.md:15–16`:

> **Withdraw rate-limit source** | V3: Return values from `decreaseLiquidity()` only (not `collect()`) | V4: Actual balance deltas... **Withdraw fee inclusion** | V3: **Excluded** — fees are collected in a separate `collect()` call and not rate-limited

The V3 design is explicit that fees should not consume withdraw capacity (the explicit contrast against V4, which folds them in). The code uses the V4 pattern. Under normal operation the bucket is over-decremented by `fees` on every call; if accrued fees alone exceed the configured `maxAmount`, the call reverts via rate-limit underflow.

Root cause

- `UniswapV3Facet.removeLiquidity` (`src/facets/uniswap-v3/UniswapV3Facet.sol:456–460`) — `valueWithdrawn` computed from post-collect balance delta
- `docs/UNIV3_UNIV4_COMPARISON.md:15–16` — explicit design statement contradicted by code
- Shares a root cause with 4.1: both use post-collect balance delta rather than `decreaseLiquidity`'s principal-only return value. A fix to 4.1 that adopts principal-only accounting (measure delta after `decreaseLiquidity` and before `collect`, or capture the return value) **also fixes 4.9**. A fix to 4.1 by splitting `collect` into a standalone entry point leaves 4.9 unfixed.

Impact boundary

What is possible:

- Over-decrement of `LIMIT_UNISWAP_V3_WITHDRAW` by `fees` on every legitimate `removeLiquidity` call
- Rate-limit underflow when accrued fees exceed remaining `maxAmount`, blocking unwind until recharge
- **Forward-looking:** 4.1's slippage post-check today suppresses a compromised-relayer tiny-burn budget-drain — a `liquidity=1` micro-burn requires `min≈0` to pass Uniswap, but 4.1's post-check requires `min ≥ fees × S > 0`. If 4.1 is fixed by adopting principal-only slippage without also fixing

4.9's accounting, the drain becomes active

What is not possible:

- Fund theft
- Loss beyond budget exhaustion

4.10: Non-atomic `setLiquidityLowerTickBound` / `setLiquidityUpperTickBound` updates in `UniswapV3Facet` open transient out-of-policy mint window

Severity: Low

What happens

`UniswapV3Facet` exposes two independent admin setters for lower and upper liquidity tick bounds. A governance rebalance that loosens one side before tightening the other opens a transient superset window. A compromised relayer can front-run inside that window to mint positions whose ticks would fail under the final policy, and those positions persist because existing positions are never retroactively validated.

`UniswapV4Facet` already uses an atomic `setTickLimits` for both bounds; V3 lacks the equivalent.

Root cause

- `UniswapV3Facet.setLiquidityLowerTickBound` (`src/facets/uniswap-v3/UniswapV3Facet.sol:243`)
- `UniswapV3Facet.setLiquidityUpperTickBound` (`src/facets/uniswap-v3/UniswapV3Facet.sol:262`)
- No atomic `setTickLimits(low, high)` entry point on V3 (`UniswapV4Facet.setTickLimits` at `src/facets/uniswap-v4/UniswapV4Facet.sol:144` demonstrates the pattern is known to the team)
- No retroactive validation of existing positions on bound update (`_validateAddLiquidityParameters` at `UniswapV3Facet.sol:630-657` runs only at mint time)

Impact boundary

What is possible:

- Out-of-policy positions persist after governance update
- Bounded by deposit rate limits; reversible via `removeLiquidity` **conditional on 4.1 being fixed** — a position whose accrued fees exceed `(1-S)/S` of principal cannot be cleanly unwound while 4.1 is open

What is not possible:

- Direct fund loss

4.11: Unconditional `÷ secondsPerLiquidityCumulativesDelta` in `UniswapV3Utils.consult` reverts when zero in-range liquidity in TWAP window; result is discarded by callers

Severity: Low

What happens

`UniswapV3Utils.consult()` unconditionally divides by `secondsPerLiquidityCumulativesDelta`. Uniswap's `secondsPerLiquidityCumulative` accumulator only advances when in-range liquidity exists, so the delta is zero for any pool with zero in-range liquidity across the full TWAP window — the division then reverts with a Solidity 0.8 panic. Both `UniswapV3Facet._getPoolData` (`UniswapV3Facet.sol:604-607`) and `_getExpectedAmounts` (`UniswapV3Facet.sol:665-668`) call `consult` and explicitly discard the `harmonicMeanLiquidity` return value (commented `// ignore harmonicMeanLiquidity`). The computation exists solely to produce a value no caller reads, but its revert blocks `swap()` and `addLiquidity()`.

Fix: guard the division with a zero check, returning 0 for the harmonic mean.

```
harmonicMeanLiquidity = secondsPerLiquidityCumulativesDelta == 0
    ? 0
    : uint128(secondsAgoX160 / (uint192(secondsPerLiquidityCumulativesDelta) << 32));
```

Separately: `pool.observe()` itself reverts if `slot0.observationCardinality < 2` (Uniswap-side preflight requirement). This is an independent failure mode with a different remediation — pool admin must call `pool.increaseObservationCardinalityNext(...)` once. The facet does not surface this gracefully but the in-facet code fix above does not address it; treat the two as distinct concerns.

Root cause

- `UniswapV3Utils.consult` (`src/facets/uniswap-v3/UniswapV3Utils.sol:179-181`) — unguarded division by `secondsPerLiquidityCumulativesDelta`
- Both call sites discard the return value, so the dividing computation produces a value no caller uses

Impact boundary

What is possible:

- DoS of `swap()` and `addLiquidity()` for any pool with zero in-range liquidity across the entire TWAP window — common in newly-deployed pools before initial liquidity adds, or in long-dormant pools where all liquidity is out-of-range
- Adversarial setup is possible but expensive: a sufficiently liquid attacker could remove their own liquidity from a small pool across the TWAP window. Cost is high relative to a DoS-only outcome

What is not possible:

- Fund loss
- Persistent state corruption
- Bypass of slippage or rate-limit checks — these are downstream of `consult` and the revert blocks reaching them

---

4.12: Missing `poolKey.hooks == address(0)` enforcement in `UniswapV4Facet` allows hook-driven balance-delta manipulation that undercharges deposit rate limits

Severity: Low

What happens

`UniswapV4Facet` verifies `keccak256(abi.encode(poolKey)) == poolId` but never checks `poolKey.hooks == address(0)`. Passing empty `hookData` does not disable hooks in Uniswap v4 — the hooks address embedded in the `PoolKey` is what's invoked. For deposit operations, accounting uses ALMProxy before/after balance deltas: a hook that donates tokens to the proxy mid-call reduces the measured outflow and undercharges deposit rate limits. Project documentation explicitly identifies hookless pools as a hard requirement, but enforcement is operational only.

`DEFAULT_ADMIN_ROLE` is fully trusted, so exploitation requires governance to whitelist a hooked pool plus the hook to be malicious — both validators downgraded severity for that reason.

Root cause

- `UniswapV4Facet._requirePoolIdMatch` (~`src/facets/uniswap-v4/UniswapV4Facet.sol:760`) — verifies `poolId` integrity but not the `hooks` field
- Deposit rate-limit accounting via balance delta — manipulable by mid-call hook donation

Impact boundary

What is possible (with malicious whitelisted hook):

- A single hooked deposit undercharges the deposit rate-limit by the donation magnitude — mid-call donation inflates `endingBalance`, deflating the measured outflow that `_clampedSub` computes (`UniswapV4Facet.sol:470–475, 480–484`)
- Repeated across a recharge period, the realized aggregate outflow exceeds the configured budget by an unbounded multiple
- Withdraw path (`_decreaseLiquidity`, `UniswapV4Facet.sol:491–519`) is *not* undercharge-vulnerable — it uses raw `endingBalance - startingBalance` (no `_clampedSub`), so a mid-call hook that steals funds causes an underflow revert (liveness DoS), not under-accounting

What is not possible:

- Manifestation absent governance whitelisting a hooked pool — admin trust assumption blocks the precondition
- Direct fund theft

4.13: Missing `maxSlippage != 0` gate on `UniswapV4Facet` liquidity operations conflicts with `CODE_NOTES.md`'s disable-by-zeroing-slippage convention

Severity: Low

What happens

`UniswapV4Facet.swap()` requires `maxSlippage != 0`; the three liquidity functions (`mintPosition`, `increasePosition`, `decreasePosition`) do not. `CODE_NOTES.md` asserts the `maxSlippage != 0` gate covers all operations; `docs/UNIV3_UNIV4_COMPARISON.md` describes a deliberate V4 design that uses tick limits rather than slippage for liquidity operations. The two documents disagree.

A compromised relayer can deploy V4 liquidity when an operator has set `maxSlippage = 0` believing it disables V4 entirely. Tick limits and rate limits provide alternative disable levers, so the operational impact is bounded — but the documentation–code disagreement is a foot-gun on the **operator** side, not the relayer side: the operator's mental model ("zeroing `maxSlippage` disables V4 for this pool") is what's wrong, and which doc they read determines whether the model matches code.

Root cause

- `UniswapV4Facet.mintPosition` (~`UniswapV4Facet.sol:126`) — no `maxSlippage != 0` gate

- `UniswapV4Facet.increasePosition` (~`UniswapV4Facet.sol:182`) — no gate
- `UniswapV4Facet.decreasePosition` (~`UniswapV4Facet.sol:232`) — no gate
- Internal documentation conflict (`docs/CODE_NOTES.md:123` says all swap-and-liquidity ops require `maxSlippage != 0`; `docs/UNIV3_UNIV4_COMPARISON.md:20–21` documents V4 liquidity as tick-limited rather than slippage-gated, which matches the code)

Impact boundary

What is possible:

- Operator confusion leading to misconfigured "disabled" state that still admits liquidity operations
- Bounded by tick limits and rate limits

What is not possible:

- Direct fund theft

4.14: `OTCFacet` does not snapshot the active buffer at `send()`; admin rotation between send and claim orphans settlement proceeds at the old buffer

Severity: Low

What happens

`OTCFacet.send()` does not snapshot the active buffer address. `OTCFacet.claim()` always pulls from the current `parameters[exchange].buffer`. Because `setBuffer()` permits rotation whenever `isSwapReady()` is true (which time-based recharge can satisfy — see finding 4.3), governance can rotate the buffer before settlement proceeds arrive at the original buffer. Once rotated, settlement proceeds remain at the old buffer and are unreachable via the standard `claim()` path until governance rotates back.

This finding is causally enabled by 4.3. The `setBuffer` gate at `OTCFacet.sol:90–93` requires `sentTimestamp == 0 || getIsSwapReady(exchange)`. If 4.3 were fixed such that `getIsSwapReady` required `normalizedClaimed >= normalizedSent`, mid-swap rotation would be blocked and this finding disappears. Either fix path closes the surface — they are not independent issues but a finding pair.

No principal loss — the funds are stranded, recoverable by admin rotating back to the original buffer, **provided that buffer still holds the funds and has not been re-purposed or compromised in the interim**.

Root cause

- `OTCFacet.send()` (`OTCFacet.sol:136–167`) does not record the buffer in `State`
- `OTCFacet.claim()` (`OTCFacet.sol:181`) reads the live `parameters[exchange].buffer` at claim time rather than the buffer that was active at send time
- `OTCFacet.setBuffer()` (`OTCFacet.sol:90–93`) gate is satisfied by `isSwapReady`, which can be true via time-based recharge (finding 4.3)

Impact boundary

What is possible:

- Settlement proceeds stranded at old buffer after rotation
- Recovery requires active admin intervention (rotate back to old buffer); recovery is not automatic and depends on the old buffer remaining intact

What is not possible:

- Permanent loss (assuming the old buffer is recoverable by admin action)

4.15: OTC pending-swap state lives in Controller's ERC-7201 slot — Controller replacement zeroes pending state

Severity: Low

What happens

OTC pending state (`normalizedSent`, `sentTimestamp`, `normalizedClaimed`) is stored in the delegatecalling Controller's ERC-7201 slot. A replacement Controller starts with zeroed state. After admin replays the integration configuration, `getIsSwapReady()` immediately returns true, allowing a second `send()` or premature buffer rotation while the previous Controller still has a pending swap.

Rate limits still bound per-period throughput but do not enforce the one-outstanding invariant across Controller upgrades. Distinct from finding 4.3 — this is the admin-upgrade trigger path; finding 4.3 is the time-recharge trigger path.

Root cause

- OTC state stored in Controller's ERC-7201 slot, not in the OTCBuffer or in a Beacon-sync'd slot
- Controller replacement triggers state reset; admin replay re-enables `send()` without acknowledging the prior pending swap

Impact boundary

What is possible:

- Concurrent pending swaps across the upgrade boundary
- Premature buffer rotation in the new Controller

What is not possible:

- Manifestation absent an admin Controller-replacement event

4.16: `OTCFacet.setBuffer` does not verify the new buffer's stored `proxy` matches the Controller's ALMProxy

Severity: Low

What happens

`OTCFacet.setBuffer()` accepts any address without verifying the buffer's stored `proxy` matches the current ALMProxy. `OTCBuffer.approve()` targets only its initialized proxy, so claims via a mismatched buffer revert on allowance checks. With `normalizedRate == 0`, readiness never accrues, making buffer rotation impossible without admin intervention — and settlement tokens stranded at the mismatched buffer.

Root cause

- `OTCFacet.setBuffer()` does not call `IOTCBuffer(buffer).proxy()` and compare to Controller's ALMProxy

Impact boundary

What is possible:

- Stranded settlement tokens at mismatched buffer
- Recovery requires admin intervention

What is not possible:

- Manifestation absent an admin error in the `setBuffer` argument

4.17: Preview-based rate-limit charging in `MapleFacet.requestRedemption` (`convertToAssets(shares)`) without claim-time reconciliation

Severity: Low

What happens

Same defect class as finding 4.4 (ERC7540) but on Maple. `MapleFacet.requestRedemption` pre-charges `LIMIT_MAPLE_REDEEM` using `convertToAssets(shares)` at request time; Maple settles asynchronously and there is no claim-time reconciliation. If settlement assets exceed the preview (yield accrual, rounding), the delta bypasses the configured Maple redeem budget.

Root cause

- `MapleFacet.requestRedemption` — pre-charge via `convertToAssets`
- No reconciliation hook in claim path

Impact boundary

What is possible:

- Rate-limit under-enforcement on yield drift
- Funds remain in the proxy — accounting integrity defect, not custody loss

What is not possible:

- Fund theft

4.18: `MapleFacet.requestRedemption` does not require the cancel key to exist; relayer can enqueue an irreversible request

Severity: Low

What happens

`MapleFacet.requestRedemption` (`src/facets/maple/MapleFacet.sol:41-60`) decrements `LIMIT_MAPLE_REQUEST_REDEEM` and calls `requestRedeem` on the Maple token. It does not verify that the corresponding `LIMIT_MAPLE_CANCEL_REDEEM` key exists. `cancelRedemption` (`:63-82`)

gates hard on `_rateLimitExists(getCancelRedeemRateLimitKey(mapleToken))` — if the cancel key is unset, cancellation reverts with `MapleFacet/invalid-action`.

If governance configures only the request key and a compromised relayer submits a request, Maple holds the pending redemption against the proxy with no facet-accessible cancel path until governance adds the cancel key. Maple typically allows one pending request per owner, so this can also DoS subsequent `requestRedemption` calls.

Root cause

- `MapleFacet.requestRedemption` — missing cancel-key existence preflight against `getCancelRedeemRateLimitKey(mapleToken)`

Impact boundary

What is possible:

- Pending request trapped until governance adds the cancel key
- Single-pending-request DoS on further `requestRedemption` calls in the meantime

What is not possible:

- Fund theft (shares remain in the Maple position, redeemable directly via the underlying contract or by configuring the cancel key)

4.19: `ERC7540Facet.requestDeposit`/`requestRedeem` do not require the corresponding claim keys to exist

Severity: Low

What happens

`requestDeposit` and `requestRedeem` do not preflight the corresponding claim key. If governance later disables claim keys (or never sets them) while request keys remain active, a compromised relayer can accumulate assets/shares in pending ERC-7540 state with no facet-accessible claim path. Analogous to finding 4.18 on Maple.

Root cause

- `ERC7540Facet.requestDeposit` / `requestRedeem` — missing claim-key existence preflight

Impact boundary

What is possible:

- Pending state accumulates with no claim path

What is not possible:

- Fund theft

4.20: `ERC7540Facet.claimDeposit` calls 2-arg `mint(shares, receiver)`, bricking claims on vaults that require 3-arg `mint(shares, receiver, controller)`

Severity: Low

What happens

`ERC7540Facet.claimDeposit` invokes `IERC4626.mint(shares, receiver)` — the 2-argument form. Some ERC-7540 vaults (those tracking per-controller pending state) require the 3-argument `mint(shares, receiver, controller)` form per ERC-7540's controller semantics. For such vaults, `claimDeposit` reverts and pending deposits are unclaimable through the facet.

This is a separate code-path defect from finding 4.4; the request-time pre-charge for deposits also discards the returned `requestId` rather than tracking it for later claim correlation.

Root cause

- `ERC7540Facet.claimDeposit` — 2-arg `mint` call, no per-controller dispatch
- `requestId` from request-time call is discarded

Impact boundary

What is possible:

- Deposits to controller-segregated ERC-7540 vaults are unclaimable

What is not possible:

- Fund theft (assets remain in the vault, redeemable directly)

4.21: `WEETHModule.claimWithdrawal` re-derives the EtherFi `WithdrawRequestNFT` at claim time from a live pointer chain; rotation of the NFT contract bricks claims for in-flight requests

Severity: Low

What happens

`WEETHModule.claimWithdrawal` re-derives the `WithdrawRequestNFT` address at claim time by walking the live `weETH → eETH → liquidityPool → withdrawRequestNFT` pointer chain. No binding is stored when the NFT is minted to the module. If EtherFi rotates the `WithdrawRequestNFT` contract while requests are outstanding, the module checks `isValid` / `isFinalized` against the *new* NFT for an ID that exists only on the *old* one — claims revert and the in-flight withdrawals become temporarily unclaimable.

There is no direct fund loss. Recovery requires a `WEETHModule` UUPS upgrade that pins or stores the original NFT address per request.

Root cause

- `WEETHModule.claimWithdrawal` — NFT address resolved at claim time, not stored at mint time
- `onERC721Received` is non-storing

Impact boundary

What is possible:

- In-flight withdrawals stuck across an EtherFi NFT rotation event

What is not possible:

- Fund theft (UUPS upgrade restores liveness)

4.22: `FarmFacet.claimReward` gate-checks but does not decrement; `FarmFacet.withdraw` calls `_claimReward` unconditionally

Severity: Low

What happens

Two related defects in `FarmFacet`:

(a) `claimReward` gate-checks the existence of the claim-reward key but never decrements it, allowing unlimited standalone claims against a configured budget.

(b) `withdraw` calls `_claimReward` unconditionally without checking the claim-reward key at all. Disabling claims by zeroing that key has no effect on the withdraw path. Worse, if the farm's reward subsystem reverts (e.g. paused), the unconditional reward claim blocks principal exit — coupling principal liveness to reward subsystem health.

Because claimed rewards flow inward to the proxy, there is no principal-loss vector; the harm is governance being unable to independently gate or budget reward claims relative to withdrawals, plus the principal-exit DoS coupling.

Root cause

- `FarmFacet.claimReward` (~`src/facets/farm/FarmFacet.sol:72–81`) — no decrement
- `FarmFacet.withdraw` (~`:84–101`) — unconditional `_claimReward`
- `FarmFacet._claimReward` (~`:131–142`) — no key check

Impact boundary

What is possible:

- Governance budget bypass on standalone claims
- Principal-exit DoS when reward subsystem reverts

What is not possible:

- Fund theft

4.23: `SuperstateFacet.subscribe` has no min-out and no governance-configured exchange-rate floor

Severity: Low

What happens

`SuperstateFacet.subscribe` decrements the rate limit, approves USDC, and calls `ustb.subscribe` with no post-call balance-delta check and no governance-configured exchange-rate bound. This deviates from the `ERC4626Facet` pattern (`maxExchangeRates`) documented in the threat model as the required defense.

Per-tx loss is bounded by `LIMIT_SUPERSTATE_SUBSCRIBE`; there is no on-chain redemption path to unwind a bad fill. USTB fees are currently documented as zero and the integration is opt-in, making this a low-likelihood defensive-coding gap.

Root cause

- `SuperstateFacet.subscribe` (~`src/facets/superstate/SuperstateFacet.sol:58-69`) — no min-out, no exchange-rate bound

Impact boundary

What is possible:

- Bad fill bounded by per-tx rate-limit budget; no on-chain unwind path

What is not possible:

- Loss exceeding the rate limit

4.24: `ERC4626Facet.redeem` has no governance exit-rate floor

Severity: Low

What happens

`ERC4626Facet` deposits are guarded by a governance-configured `maxExchangeRate` ceiling (`src/facets/erc4626/ERC4626Facet.sol:45`, enforced at `:121`). The redeem path has no symmetric **minExchangeRate** floor: `minAssetsOut` is supplied by the relayer (`ERC4626Facet.sol:158-181`), and the withdraw rate-limit is decremented by `assets` actually received — not by a value-equivalent of the shares burned.

The structural consequence: on a vault whose exchange rate has degraded, a compromised relayer can burn shares notionally worth $X (pre-loss) and receive $Y << $X. The withdraw budget — sized in asset-denominated terms to bound dollar-equivalent exit pace — consumes only $Y, so more shares can be exited per period than the rate-limit sizing assumed. This holds for any degraded exchange rate, not only at the extreme rounding-to-zero boundary.

Root cause

- `ERC4626Facet.redeem` — no admin-set exit-rate floor parameter (asymmetric with the deposit-side `maxExchangeRate` ceiling)
- Withdraw rate-limit decrement is denominated in received assets, not in shares × floor-rate

Impact boundary

What is possible:

- Compromised relayer redeems shares at a degraded exchange rate while consuming less budget than the share value warrants
- Cumulative effect: exit pace per period can exceed the rate-limit's intended dollar-equivalent cap by the magnitude of the rate degradation

What is not possible:

- Loss unbounded by share holdings; redeem only burns the shares the proxy already holds
- Manifestation absent a degraded exchange rate at the vault — the issue is dormant otherwise

---

4.25: Missing remote-peer (`peers[dstEid]`) binding in `LayerZeroFacet.transfer` allows misroute on external OFT peer rotation

Severity: Informational

What happens

The facet whitelists routes by `(token, oft, destinationEndpointId)` but never reads or pins the OFT's `peers[dstEid]` mapping, which determines the actual destination contract that receives the transfer on the remote chain. If the OFT's admin rotates the peer for that `dstEid` after governance enables the route, a compromised relayer can send transfers that pass all local checks but land at an unintended (or malicious) destination OApp.

Loss per period is bounded by the configured rate limit; exploitation requires both a compromised relayer and an external OFT peer rotation event.

Root cause

- `LayerZeroFacet.transfer` (~`LayerZeroFacet.sol:140-229`) — no `oft.peers(dstEid)` read or pin
- Rate-limit key derivation (~`LayerZeroFacet.sol:247`) — keyed on `(token, oft, dstEid)`, not on the resolved peer

Impact boundary

What is possible:

- Misroute to an attacker-controlled remote OApp after peer rotation
- Loss bounded by rate-limit budget per period

What is not possible:

- Misroute without an external peer-rotation event
- Bypass of the per-tx rate-limit decrement

4.26: Documentation drift from the "Bespoke rate limits" refactor

Severity: Informational

What happens

Commit `0f3a1ef` (2026-05-06, "Bespoke rate limits") restructured rate-limit key handling across several facets without updating `docs/RATE_LIMITS.md`, `docs/SECURITY.md`, or `docs/CODE_NOTES.md`. Three stale claims remain in those docs whose pre-refactor language no longer describes how the code behaves; an operator following any of them can reach a partial-config state that the doc implies is impossible.

(a) `docs/SECURITY.md:69` — Centrifuge cancel/claim-cancel keys. Doc claims setting the deposit rate limit to 1 unblocks cancel and claim paths. Pre-refactor (commit `54cbc5b`, 2026-03-12), `CentrifugeLib` used the shared ERC-7540 `LIMIT_DEPOSIT` key as the presence check for both cancel paths and the doc was correct. The refactor replaced that single key with four dedicated keys (`_LIMIT_CANCEL_DEPOSIT`, `_LIMIT_CLAIM_CANCEL_DEPOSIT`, `_LIMIT_CANCEL_REDEEM`, `_LIMIT_CLAIM_CANCEL_REDEEM`, `CentrifugeFacet.sol:97–105`), each presence-checked individually via `_requireRateLimitExists` at `CentrifugeFacet.sol:146`, `:166`, `:193`, `:213`. The "set the deposit rate limit to 1" lever has no effect under current code — there is no generic deposit rate-limit key on `CentrifugeFacet`. The four independently-checked keys also admit asymmetric configuration: with the cancel key set but the claim-cancel key absent, `cancel*Request` succeeds and `claimCancel*Request` reverts; Centrifuge's `REQUEST_ID = 0` semantics (`CentrifugeFacet.sol:150`, `:197`) then block new requests until governance adds the missing key.

(b) `docs/RATE_LIMITS.md:170–171` — Aave/ERC4626 withdraw deposit-key precondition. Doc claims withdraws require `maxAmount > 0` on the deposit key. Code uses `_tryIncreaseRateLimit` (`src/facets/aave/AaveFacet.sol:151`, `src/facets/erc4626/ERC4626Facet.sol:152`, `:177`), which silently no-ops when the deposit key's `maxAmount == 0` (`src/facets/Facet.sol:54–58`); the withdraw path is gated only by the withdraw key. The refactor's `test_withdrawERC4626_zeroDepositRateLimit` (`test/mainnet-fork/ERC4626.t.sol:317`) codifies the relaxed behavior. Pre-refactor, `_increaseRateLimit` would have hard-reverted via `RateLimits.triggerRateLimitIncrease`'s `require(maxAmount > 0)` (`src/RateLimits.sol:103`), matching the doc at the time. Harm is the mildest of the three — an operator's first "pause via deposit key" attempt simply has no effect and is observable on the first try.

(c) `docs/RATE_LIMITS.md:114–115` and `docs/CODE_NOTES.md:170–171` — WSTETH/WEETH claim key. Both docs claim `claimWithdrawal` presence-checks the request key. Code presence-checks a dedicated claim-side key — `LIMIT_WSTETH_CLAIM_WITHDRAW` (`src/facets/wsteth/WSTETHFacet.sol:146, :175`) and `makeAddressKey(LIMIT_WEETH_CLAIM_WITHDRAW, weethModule)` (`src/facets/weeth/WEETHFacet.sol`). Pre-refactor presence checks referenced the request key, matching the docs. This sub-case is the most operationally consequential: an operator following the stale docs configures only `LIMIT_WSTETH_REQUEST_WITHDRAW` / `LIMIT_WEETH_REQUEST_WITHDRAW`; `requestWithdraw` succeeds (shares enter Lido/EtherFi's queue), but `claimWithdrawal` later reverts with `WSTETHFacet/invalid-action` / `WEETHFacet/invalid-action`. Unlike (b), no immediate signal — the keys named in the docs exist as valid rate-limit keys, just not the ones the claim path checks. Withdrawals are stuck pending until governance adds the claim key (single admin tx).

Root cause

- Commit `0f3a1ef` (2026-05-06) restructured rate-limit key handling in `CentrifugeFacet`, `AaveFacet`/`ERC4626Facet`, and `WSTETHFacet`/`WEETHFacet` (via new key constants, `_tryIncreaseRateLimit`, and dedicated claim-side presence checks) without updating `docs/RATE_LIMITS.md`, `docs/SECURITY.md:69`, or `docs/CODE_NOTES.md:170–171`
- No CI gate or PR template check ties rate-limit code changes to docs updates

Impact boundary

What is possible:

- (a) Operator-driven asymmetric Centrifuge config produces a stuck-pending-cancel state that blocks new requests until governance adds the missing key
- (b) Operator forms an incorrect mental model that zeroing the Aave/ERC4626 deposit key suspends withdraws — observable on first attempt, self-correcting
- (c) Operator configures only the WSTETH/WEETH request key; subsequent `claimWithdrawal` reverts until governance adds the claim key; shares already in Lido/EtherFi's external queue are stuck pending — recoverable, not lost

What is not possible:

- Fund loss — all three sub-cases preserve principal: Centrifuge positions remain claimable directly via the vault; Aave/ERC4626 withdraws are unaffected; WSTETH/WEETH shares remain claimable through Lido/EtherFi's queue once the claim key is added
- Manifestation absent admin misconfiguration — under the documented trust model (`DEFAULT_ADMIN_ROLE` fully trusted) the partial-config preconditions shouldn't exist on a correctly-operated deployment

---

4.27: `FarmFacet.withdraw` charges the relayer-supplied amount against the withdraw budget rather than the measured staking-token balance delta

Severity: Informational

What happens

`FarmFacet.withdraw` decrements the withdraw rate limit by the relayer-supplied `amount` *before* the call, rather than measuring the actual staking-token balance delta afterwards (the convention followed by `AaveFacet.withdraw` and `ERC4626Facet.redeem`). If a whitelisted farm returns more tokens than requested (share-based semantics, bonus, upgrade), the withdraw budget is undercharged.

Exploitation requires governance to whitelist a non-exact-amount farm. The currently integrated Synthetix-style farm is exact-amount, making this a low-likelihood defensive-coding gap.

Root cause

- `FarmFacet.withdraw` (~`:84–101`) — pre-charges `amount` rather than measuring balance delta
- Deviates from the project-wide convention

Impact boundary

What is possible:

- Withdraw budget undercharge if a non-exact-amount farm is whitelisted

What is not possible:

- Direct fund theft

4.28: Unchecked zero `stored_rates` in `CurveFacet._validateSwapMinAmountOut`

Severity: Informational

What happens

If a whitelisted Curve pool returns `rates[inputIndex] == 0`, the pre-swap `equivalentAmountOut` collapses to 0 and the slippage guard passes unconditionally — admitting an arbitrarily bad fill. If `rates[outputIndex] == 0`, a division-by-zero DoS results. Zero rates in a zero-rated coin also under-weight aggregate value in liquidity checks.

Legitimate StableSwap-NG pools return non-zero rates by design; exploitation requires a malformed or misbehaving whitelisted pool.

Root cause

- `CurveFacet._validateSwapMinAmountOut` (~`src/facets/curve/CurveFacet.sol:414–441`) — no zero-rate guard
- `addLiquidity` (~`:134`) and `removeLiquidity` (~`:209`) similarly affected

Impact boundary

What is possible (with malformed whitelisted pool):

- Slippage bypass on input-side zero
- DoS on output-side zero

What is not possible:

- With well-formed StableSwap-NG pools

4.29: Last burn-limit chunk in `CCTPFacet.transferWithFee` reverts when remainder ≤ `maxFee`

Severity: Informational

What happens

When the total transfer amount is split by Circle's `burnLimitsPerMessage`, the final remainder chunk can be `≤ maxFee`. The per-chunk `require(maxFee < transferAmount)` then reverts the entire transaction atomically. No funds move and no partial state persists; the only impact is liveness of the `transferWithFee` path.

The behavior is acknowledged in a code comment as a known limitation.

Root cause

- `CCTPFacet._transfer` while-loop (~`CCTPFacet.sol:211–224`) — per-chunk gate

Impact boundary

What is possible:

- Liveness DoS when transfer amount lands at a problematic remainder

What is not possible:

- Fund loss

4.30: No admin sweep on `WEETHModule` — WETH accidentally routed to the module is stranded until UUPS upgrade

Severity: Informational

What happens

`WEETHModule.claimWithdrawal` (`src/facets/weeth/WEETHModule.sol:119–145`) reads `ethReceived = address(this).balance` — i.e. the **module's total ETH balance**, not the per-claim delta — wraps that full amount to WETH, then transfers exactly the wrapped amount to the proxy. Two consequences fall out of that asymmetry:

- **Stray ETH self-sweeps.** Any ETH accidentally routed to the module gets opportunistically wrapped on the next legitimate `claimWithdrawal` call and forwarded to the proxy. No admin action needed.
- **Stray WETH is stranded.** The `IERC20Like(weth).transfer(msg.sender, ethReceived)` line transfers only the just-wrapped amount, not the module's WETH balance. Any WETH that arrives at the module by other paths (e.g. a misconfigured `TransferAsset` route) is invisible to every code path on the module.

There is no admin `sweep`/`rescueToken` function. WETH (and any non-WETH ERC-20 that lands here by misrouting) is recoverable only via UUPS upgrade adding a sweep.

This requires a trusted governance misconfiguration plus a compromised relayer to reach any material amount; impact is recoverable.

Root cause

- `WEETHModule.claimWithdrawal` — wraps the total ETH balance (so stray ETH self-sweeps) but transfers only the per-call wrapped amount of WETH (so pre-existing WETH balance is not forwarded)
- No `sweep`/`rescueToken` admin function — recovery requires UUPS upgrade

Impact boundary

What is possible:

- Stranded WETH recoverable only via UUPS upgrade

What is not possible:

- Stranding absent an upstream governance routing error sending stray ERC-20s to the module
- Loss — funds are recoverable via UUPS upgrade

4.31: `IEnumerableIntegrations` struct-layout coupling between `Beacon`, `Controller`, and facets has no schema-version handshake

Severity: Informational

What happens

`Controller.updateIntegrations` (`src/Controller.sol:91–117`) consumes `IBeacon(beacon).getConfigs(ids)` via a typed return whose layout is defined by `IEnumerableIntegrations.Config` / `Wire` / `Dispatch` (`src/interfaces/IEnumerableIntegrations.sol`). The Controller's `beacon` address is set immutably at construction, so a Beacon swap is itself a Controller redeployment — there is no live runtime path for the pair to drift. The realistic failure path is a **Controller redeployment compiled against a modified `IEnumerableIntegrations` struct while pointing at the pre-existing Beacon** (e.g. interface evolved between versions): the typed-return decode reverts, breaking `updateIntegrations` for that deployment. Facets that store interface-defined structs in ERC-7201 slots face an analogous upgrade hazard if the layout changes without bumping the slot.

The same coupling extends off-chain — monitoring tools and governance scripts that decode `Beacon.integrations()` returns must be recompiled in lockstep with on-chain layout changes. This is a deployment-discipline concern, not a code bug.

All failure modes fail closed; no principal-loss path exists; all preconditions require trusted-governance mistakes.

Root cause

- `src/interfaces/IEnumerableIntegrations.sol` — no schema version field; struct evolution has no compile-time signal

- `Controller` (`src/Controller.sol:94`) — typed-return decode of `Beacon.getConfigs` relies on the local layout matching whatever Beacon was compiled with
- `Beacon` is non-upgradeable and `Controller.beacon` is immutable — the realistic failure vector is Controller redeployment against a stale Beacon, not parallel upgrades

Impact boundary

What is possible:

- Operational DoS during a misordered multi-contract upgrade

What is not possible:

- Fund loss

4.32: `BasinFacet` deposit/withdraw lack a governance-configured exchange-rate guard

Severity: Low

What happens

`BasinFacet` shares the missing-governance-floor pattern of §4.23 (Superstate) and §4.24 (ERC-4626 redeem); what distinguishes it is that Basin prices conversions through external rate providers, so the exposure is an in-staleness-window oracle print rather than an open-ended haircut (see *What is not possible* below). `BasinFacet.deposit` gates only on the allocator-supplied `minSharesOut` (`require(shares >= minSharesOut)`, `src/facets/basin/BasinFacet.sol:73`), and `BasinFacet.withdraw` gates only on the allocator-supplied `minConversionRate` (`require(assetsWithdrawn * 1e18 >= sharesBurned * minConversionRate)`, `:104–107`). There is no governance-configured exchange-rate ceiling/floor analogous to `ERC4626Facet`'s `maxExchangeRate` or `AaveFacet`'s `maxSlippage`. Because `ALLOCATOR_ROLE` is compromisable under the documented trust model, an attacker controlling an allocator can pass `minSharesOut = 0` / `minConversionRate = 0` and accept whatever conversion Grove Basin returns. If Basin's rate providers temporarily value assets adversely (or value a non-1:1 credit token below stablecoins), the proxy realizes a principal haircut. Loss is bounded by the configured per-`(asset, basin)` deposit/withdraw rate limits, but there is no additional governance slippage budget — the same defensive layer the comparable deposit facets carry.

Each `BasinFacet` call handles a single asset (`IBasinFacet`), so a deposit-one-token / withdraw-another round trip is two separately rate-limited calls, not one — the per-call rate limit bounds each leg independently.

Root cause

- `BasinFacet.deposit` (`src/facets/basin/BasinFacet.sol:52–79`) — only allocator `minSharesOut` (`:73`); no governance exchange-rate storage
- `BasinFacet.withdraw` (`:82–112`) — only allocator `minConversionRate` (`:104–107`)
- Asymmetry with sibling facets: `ERC4626Facet` carries `maxExchangeRates` (`src/facets/erc4626/ERC4626Facet.sol:45`, enforced `:121`) and `AaveFacet` carries `maxSlippages` (`src/facets/aave/AaveFacet.sol:50`, enforced `:123`); `BasinFacet` declares neither

Impact boundary

What is possible:

- A compromised allocator accepts an adverse Basin conversion, realizing a principal haircut bounded by the configured per-`(asset, basin)` rate limit per period
- A defense-in-depth gap relative to the `ERC4626`/`Aave` deposit paths, which apply a governance exchange-rate guard on top of the allocator's mins

What is not possible:

- Loss exceeding the configured rate-limit budget per period
- A single-call cross-asset arbitrage — the facet is single-asset per call, so cross-asset value transfer requires two separately rate-limited calls
- Manifestation absent a fresh adverse Grove Basin conversion at execution time — `GroveBasin` prices deposit/withdraw through rate providers (`ChronicleRateProvider`/`FixedRateProvider`) and reverts on a stale rate (`_getConversionRate` → `StaleRate`), so the exposure is an in-staleness-window oracle print that values the credit token below par, not an arbitrary rate; `GroveBasin` applies no internal deposit/withdraw min of its own, so the facet-level governance floor is the only missing guard

---

4.33: `PendleFacet.redeem` computes router `minTokenOut` with a hardcoded `1e18` scale, reverting redemption on non-18-decimal yield-token markets

Severity: Low

What happens

`PendleFacet.redeem` computes the router-side `minTokenOut` as `pyAmountIn * 1e18 / IYTLike(yt).pyIndexCurrent() - 5` (`src/facets/pendle/PendleFacet.sol:133`). PT/YT and `pyIndexCurrent()` are 18-decimal in Pendle V2, so this expression is an 18-decimal-scaled SY amount. `_createSimpleTokenOutput` (`:183–197`) sets both `tokenOut` and `tokenRedeemSy` to the SY's yield token with no swap, so the router

enforces `minTokenOut` against `netTokenOut` measured in that yield token's **native** decimals. For a 6-decimal yield token (e.g. aUSDC/aUSDT), `netTokenOut` is ~`1e6`-scaled while `minTokenOut` is ~`1e18`-scaled — an impossible internal minimum that makes the router revert deterministically, blocking redemption for that market. The facet's own post-call check `require(tokenOutAmount >= minAmountOut)` on the balance delta (`:155`) is what actually protects value, so there is no underfilled-acceptance path.

An in-code NOTE (`:114–116`) explicitly warns "DO NOT use for markets with non-standard SYs … (Non-standard SYs: ePENDLE, mPENDLE, aTokens (aUSDC, aUSDT))" — it documents the hazard and shifts responsibility to governance but does not gate it, and the deterministic revert applies to any standard-SY market whose yield token is non-18-decimal. A secondary dust case: for `pyAmountIn` small enough that `pyAmountIn * 1e18 / pyIndexCurrent() < 5`, the `- 5` underflows (Solidity 0.8 panic) before the router call; the project's own `test_redeemPendlePT_amountTooSmall` confirms this. It is avoidable by batching a larger redemption.

Root cause

- `PendleFacet.redeem` (`src/facets/pendle/PendleFacet.sol:118–163`) — `minTokenOut = pyAmountIn * 1e18 / pyIndexCurrent() - 5` (`:133`) assumes 18-decimal scaling and a raw 5-unit buffer
- `_createSimpleTokenOutput` (`:183–197`) — points the router minimum at the final yield token, which the router measures in native decimals
- In-code NOTE (`:114–116`) documents the non-standard-SY hazard but does not enforce it; post-call balance-delta check (`:155`) bounds value but does not prevent the router-side revert

Impact boundary

What is possible:

- Deterministic redemption DoS for any onboarded market whose yield token is non-18-decimal — recoverable only by reconfiguration or a facet fix
- Dust-amount underflow revert when `pyAmountIn * 1e18 / pyIndexCurrent() < 5`, avoidable by batching

What is not possible:

- Underfilled acceptance or principal loss — the post-call `tokenOutAmount >= minAmountOut` check (`:155`) operates on the real balance delta
- Manifestation on the currently-tested 18-decimal-yield markets (sUSDe, USDe, stETH)
- Cross-market contagion — the failure is confined to the misconfigured market

---

4.34: `ERC4626Facet.deposit`'s `maxExchangeRate` ceiling does not catch Maple's impairment asymmetry, so deposits into an impaired pool socialize pre-existing losses

Severity: Low

What happens

§4.24 treats the deposit path as the *guarded* side of `ERC4626Facet` — protected by a governance-configured `maxExchangeRate` ceiling (`src/facets/erc4626/ERC4626Facet.sol:45`, enforced at `:121`) — and the redeem path as the gap. That ceiling is real, but for one vault class it guards the wrong quantity. `deposit` enforces only `shares >= minSharesOut` (relayer-supplied, can be `0`, `:118`) and `_getExchangeRate(shares, amount) <= maxExchangeRates[token]` (`:121`), where the rate is *assets-in / shares-minted* — a deposit-time price.

Maple ERC-4626 pools price the two legs asymmetrically during impairment: deposits mint shares off gross `totalAssets` (`convertToShares`), while exits value shares off the loss-aware `convertToExitAssets`, reduced by `unrealizedLosses`. In an impaired pool the minted `assets/shares` therefore stays ≈ 1 and clears the `maxExchangeRate` ceiling, even though the shares' immediate exit value is already below the assets paid in. A compromised allocator can deposit up to the remaining `LIMIT_4626_DEPOSIT(asset, token)` capacity (decremented at `:106`) into the impaired pool, converting the pool's pre-existing impairment into an immediate principal loss to the proxy. All Maple deposits route through this generic path — `MapleFacet` implements only the redemption queue — so no Maple-specific guard intercepts it.

The gap is documented rather than latent: `docs/OPERATIONAL_REQUIREMENTS.md` records that `ERC4626Facet` does not check `unrealizedLosses()` on a Maple pool during deposit and that fresh deposits can socialize pre-existing losses onto the proxy, with mitigation assigned to governance reducing the deposit rate limit once impairment is posted.

Root cause

- `maxExchangeRate`, the deposit path's only governance guard, bounds the deposit-time *mint* price (gross `assets/shares`), not the loss-aware *exit* value of the minted shares — so a pool whose deposit and exit prices have diverged passes the check
- No `unrealizedLosses() == 0` precondition or exit-value (`convertToAssets`) floor on the deposit path; Maple deposits use this generic facet rather than a Maple-specific guarded entry point

Impact boundary

What is possible:

- A compromised allocator deposits into an actively-impaired Maple pool, socializing the pool's pre-existing losses onto the proxy up to the configured `LIMIT_4626_DEPOSIT(asset, token)` capacity
- Repeated across recharge windows, the aggregate socialized loss tracks the deposit budget's replenishment until governance reduces the limit

What is not possible:

- Loss unbounded by the deposit rate limit — the budget caps how much underlying can enter the impaired pool per window
- Manifestation absent an active Maple impairment with deposits still enabled — dormant for healthy pools and for any pool whose deposit limit governance has already zeroed
- Any effect on ordinary symmetrically-priced vaults — the gap is specific to Maple's deposit/exit pricing asymmetry, and `maxExchangeRate` continues to bound those normally

---

4.35: `ERC7540Facet` async deposit/claim has no min-out or governance-configured exchange-rate floor

Severity: Low

What happens

`ERC7540Facet.requestDeposit` approves the asset, decrements the double-keyed `(token, asset)` request-deposit rate limit by `amount` (`src/facets/erc7540/ERC7540Facet.sol:75`), and transfers the assets into the vault's request escrow (`:78–81`) — the assets leave proxy custody at request time. `claimDeposit` then presence-checks the claim key (`:91`) and mints `maxMint(proxy)` shares (`:94–97`). Neither path carries a `minSharesOut` check or a governance-configured exchange-rate bound. This is the async analogue of the synchronous `ERC4626Facet.deposit`, which the report elsewhere (§4.24, §4.34) treats as the *guarded* reference: it enforces both `shares >= minSharesOut` and `_getExchangeRate(...) <= maxExchangeRates[token]` (`src/facets/erc4626/ERC4626Facet.sol:118`, `:121`). The ERC-7540 path has no equivalent of either.

Because the assets are committed when `requestDeposit` escrows them and the share count is fixed by the vault at settlement, a claim-time min-out would be structurally weak — the exposure is set at request time, not at `claimDeposit`. The operative gap is therefore the absence of any governance NAV/exchange-rate floor on the async entry, mirroring §4.23 (Superstate), §4.24 (ERC-4626 redeem), and §4.32 (Basin). On an onboarded async vault that charges an entry/subscription fee or settles at an adverse batch NAV, a compromised allocator can submit deposits that settle unfavorably and take a principal haircut that no on-chain guard rejects. Loss per window is bounded by the configured `(token, asset)` request-deposit rate-limit budget.

Root cause

- `ERC7540Facet.requestDeposit` (`src/facets/erc7540/ERC7540Facet.sol:64–87`) — escrows `amount` of the asset with no min-shares-out and no governance exchange-rate bound
- `ERC7540Facet.claimDeposit` (`:90–100`) — mints `maxMint(proxy)` with no floor on the settled shares; asymmetric with the `minSharesOut` + `maxExchangeRate` guards on the synchronous `ERC4626Facet.deposit` path

Impact boundary

What is possible:

- A compromised allocator submits async deposits that settle at an adverse NAV or entry fee, realizing a principal haircut bounded by the `(token, asset)` request-deposit rate-limit budget per window; no governance floor rejects the unfavorable fill
- Repeated across recharge windows, the aggregate socialized loss tracks the deposit budget's replenishment until governance reduces the limit

What is not possible:

- Loss exceeding the request-deposit rate-limit budget per window
- Manifestation absent an adverse settlement — dormant for async vaults that settle at par
- Theft to a third party — escrowed assets and minted shares remain in proxy custody

---

4.36: Strict `abi.decode` of mutable-external return data DoSes withdrawal/cooldown paths on any upstream return-shape change

Severity: Informational

What happens

Three facets strict-`abi.decode` the raw return bytes of an external protocol call routed through `IALMProxy.doCall`, with no tolerant-decode wrapper:

- `WSTETHFacet.requestWithdraw` decodes the Lido Withdrawal Queue return as `(uint256[])` (`src/facets/wsteth/WSTETHFacet.sol:114–123`)
- `WEETHFacet.requestWithdraw` decodes the EtherFi liquidity-pool return as `(uint256)` (`src/facets/weeth/WEETHFacet.sol:189–194`)
- `EthenaFacet.cooldownAssets`/`cooldownShares` decode the sUSDe cooldown return (`src/facets/ethena/EthenaFacet.sol:171`, `:192`)

Each call site carries an in-code NOTE acknowledging the external contract is mutable and that decoding the return value is the only way to obtain the request ID / amount. If the external protocol keeps the same selector and inputs but changes the **return shape** (a different arity, a scalar instead of an array, or empty bytes on success), `abi.decode` reverts and the entire transaction — including the preceding approval and rate-limit decrement — rolls back atomically. The effect is a temporary, governance-recoverable DoS of that one path (deposits and `claimWithdrawal` remain available); there is no fund-loss vector. This is the same strict-decode house style the report already documents on the LayerZero quote path (§4.7); §4.36 captures the withdrawal/cooldown instances.

A secondary accounting asymmetry sits inside `EthenaFacet`: `cooldownAssets` decrements the cooldown rate limit by the **input** `usdeAmount` before the decode (`:167`), while `cooldownShares` decrements by the **decoded** `assets` afterward (`:200`). The two paths charge the cooldown budget on different

bases, but neither can bypass the separate burn/transfer/bridge limits that bound actual value egress.

Root cause

- `WSTETHFacet.requestWithdraw` (`:112–123`), `WEETHFacet.requestWithdraw` (`:187–194`), `EthenaFacet.cooldownAssets`/`cooldownShares` (`:160–174`, `:183–195`) — strict `abi.decode` on mutable-external returndata with no length/shape guard
- A deliberate, NOTE-documented trade-off (the request ID/amount is only obtainable from the return value) — it explains the code but does not prevent the revert
- `EthenaFacet` cooldown rate-limit charged on input (`:167`) vs decoded output (`:200`) — inconsistent basis within one facet

Impact boundary

What is possible:

- Per-path liveness DoS if Lido / EtherFi / Ethena ships a backwards-incompatible return-ABI change while preserving the selector and inputs — recoverable by a governance facet rewire/upgrade
- Cooldown-budget accounting drift between `cooldownAssets` and `cooldownShares`

What is not possible:

- Fund loss — the revert is atomic, preserving approvals and rate-limit budget
- Attacker-triggered manifestation — the trigger is an external protocol's ABI change, outside allocator/relayer control
- Egress beyond configured limits from the Ethena cooldown mischarge — actual value movement is bounded by separate burn/transfer/bridge keys

---

5. Remediation Status

This appendix records the remediation status of each Section 4 finding; each **Fixed** finding was verified against the audited source. 31 findings were raised against the initial release candidate (`fc2fe7f`, `v1.12.0-rc.1`): **10 are Fixed and 21 Acknowledged** (risk-accepted under the documented trust model). A further **5 findings** (§4.32–§4.36) were identified in later release candidates through `a84fe96` (`v1.12.0-rc.4`) and have likewise been **Acknowledged**. Across all 36 findings: **10 Fixed, 26 Acknowledged**. Verbatim Spark responses and PR links are included in the per-finding detail.

#	Severity	Finding	Status
4.1	Medium	Fee-inclusive slippage post-check in `UniswapV3Facet.removeLiquidity` deterministically reverts on positions with accrued fees	Fixed
4.2	Medium	Approval granted before non-static `quoteOFT`/`quoteSend` in `LayerZeroFacet.transfer` allows OFT-side drain of just-approved allowance	Acknowledged
4.3	Medium	Time-based recharge in `OTCFacet.isSwapReady` allows multiple concurrent in-flight swaps, breaking the one-outstanding-swap-per-exchange invariant	Acknowledged
4.4	Medium	Request-time pre-charge without claim-time reconciliation in `ERC7540Facet` bypasses redeem rate-limit on exchange-rate drift	Acknowledged
4.5	Low	No fee-to-amount ratio bound in `CCTPFacet.transferWithFee` and per-chunk `maxFee` reuse across burn-limit splits	Fixed
4.6	Low	Unprotected UUPS `initialize()` on `WEETHModule` and `OTCBuffer` if proxy is not atomically initialized at deployment	Acknowledged
4.7	Low	`LayerZeroFacet.transfer` sets `minAmountLD` to OFT-quoted value with no operator-configured ratio floor; strict `abi.decode` of quote returns can DoS the bridging path	Fixed
4.8	Low	Hard-required ETH refund in `LayerZeroFacet.transfer` reverts entire bridge for contract relayers with non-payable fallback	Fixed
4.9	Low	Fee-inclusive withdraw rate-limit accounting in `UniswapV3Facet.removeLiquidity` charges fees against principal-only budget	Fixed
4.10	Low	Non-atomic `setLiquidityLowerTickBound` / `setLiquidityUpperTickBound` updates in `UniswapV3Facet` open transient out-of-policy mint window	Acknowledged
4.11	Low	Unconditional `÷ secondsPerLiquidityCumulativesDelta` in `UniswapV3Utils.consult` reverts when zero in-range liquidity in TWAP window; result is discarded by callers	Acknowledged
4.12	Low	Missing `poolKey.hooks == address(0)` enforcement in `UniswapV4Facet` allows hook-driven balance-delta manipulation that undercharges deposit rate limits	Acknowledged
4.13	Low	Missing `maxSlippage != 0` gate on `UniswapV4Facet` liquidity operations conflicts with `CODE_NOTES.md`'s disable-by-zeroing-slippage convention	Fixed

#	Severity	Finding	Status
4.14	Low	`OTCFacet` does not snapshot the active buffer at `send()`; admin rotation between send and claim orphans settlement proceeds at the old buffer	Acknowledged
4.15	Low	OTC pending-swap state lives in Controller's ERC-7201 slot — Controller replacement zeroes pending state	Acknowledged
4.16	Low	`OTCFacet.setBuffer` does not verify the new buffer's stored `proxy` matches the Controller's ALMProxy	Acknowledged
4.17	Low	Preview-based rate-limit charging in `MapleFacet.requestRedemption` (`convertToAssets(shares)`) without claim-time reconciliation	Acknowledged
4.18	Low	`MapleFacet.requestRedemption` does not require the cancel key to exist; relayer can enqueue an irreversible request	Acknowledged
4.19	Low	`ERC7540Facet.requestDeposit`/`requestRedeem` do not require the corresponding claim keys to exist	Acknowledged
4.20	Low	`ERC7540Facet.claimDeposit` calls 2-arg `mint(shares, receiver)`, bricking claims on vaults that require 3-arg `mint(shares, receiver, controller)`	Acknowledged
4.21	Low	`WEETHModule.claimWithdrawal` re-derives the EtherFi `WithdrawRequestNFT` at claim time from a live pointer chain; rotation of the NFT contract bricks claims for in-flight requests	Acknowledged
4.22	Low	`FarmFacet.claimReward` gate-checks but does not decrement; `FarmFacet.withdraw` calls `_claimReward` unconditionally	Fixed
4.23	Low	`SuperstateFacet.subscribe` has no min-out and no governance-configured exchange-rate floor	Acknowledged
4.24	Low	`ERC4626Facet.redeem` has no governance exit-rate floor	Acknowledged
4.25	Informational	Missing remote-peer (`peers[dstEid]`) binding in `LayerZeroFacet.transfer` allows misroute on external OFT peer rotation	Acknowledged
4.26	Informational	Documentation drift from the "Bespoke rate limits" refactor	Fixed
4.27	Informational	`FarmFacet.withdraw` charges the relayer-supplied amount against the withdraw budget rather than the measured staking-token balance delta	Fixed
4.28	Informational	Unchecked zero `stored_rates` in `CurveFacet._validateSwapMinAmountOut`	Acknowledged
4.29	Informational	Last burn-limit chunk in `CCTPFacet.transferWithFee` reverts when remainder ≤ `maxFee`	Fixed
4.30	Informational	No admin sweep on `WEETHModule` — WETH accidentally routed to the module is stranded until UUPS upgrade	Acknowledged
4.31	Informational	`IEnumerableIntegrations` struct-layout coupling between `Beacon`, `Controller`, and facets has no schema-version handshake	Acknowledged

The following additional findings were identified in later release candidates (`c741e32` / `v1.12.0-rc.3` and `a84fe96` / `v1.12.0-rc.4`).

#	Severity	Finding	Status
4.32	Low	`BasinFacet` deposit/withdraw lack a governance-configured exchange-rate guard	Acknowledged
4.33	Low	`PendleFacet.redeem` computes router `minTokenOut` with a hardcoded `1e18` scale, reverting redemption on non-18-decimal yield-token markets	Acknowledged
4.34	Low	`ERC4626Facet.deposit`'s `maxExchangeRate` ceiling does not catch Maple's impairment asymmetry, so deposits into an impaired pool socialize pre-existing losses	Acknowledged
4.35	Low	`ERC7540Facet` async deposit/claim has no min-out or governance-configured exchange-rate floor	Acknowledged
4.36	Informational	Strict `abi.decode` of mutable-external return data DoSes withdrawal/cooldown paths on any upstream return-shape change	Acknowledged

Remediation detail:

- **4.1** — Fixed in sky-ecosystem/diamond-pau#176. The facet now pre-collects accrued fees into the proxy *before* snapshotting the starting balance, so the measured `decreaseLiquidity` + `collect` delta is principal-only and the slippage check compares principal against principal. Fees still reach the proxy but are excluded from both the slippage check and the withdraw rate-limit; no separate `collect()` entry point was required. The same change resolves 4.9.
- **4.2** — Spark response: *"Ack — if an OFT is malicious, they can steal the funds through approvals. This is part of the risk model and is mitigated with rate limits and freezer functionality."*
- **4.3** — Spark response: *"Ack — this is a known part of the design, and should be handled with adequately small recharge rates in configuration. This design can be revisited in a later release."*
- **4.4** — Spark response: *"Ack — this is known functionality and expected."*
- **4.5** — Fixed in sky-ecosystem/diamond-pau#174. `transferWithFee` was replaced by a unified `transfer(amount, destinationDomain, feeCapRate)` that takes a relayer-supplied basis-points rate, bound-checked against a per-domain admin range `[minFeeCapRate, maxFeeCapRate]`. Per-chunk `maxFee` is now computed proportionally (`transferAmount × feeCapRate / 10_000`) instead of reusing a flat

`maxFee`, so the aggregate fee ceiling is bounded by `amount × maxFeeCapRate / 10_000` regardless of chunking, and the absolute `maxFeeCap` was removed. The gross-amount rate-limit decrement is unchanged — an intentional outflow-exposure budget rather than a residual defect. The same refactor resolves 4.29.

- **4.6** — Spark response: *"Ack — this is known, an uninitialized contract will not be onboarded."*
- **4.7** — Fixed in sky-ecosystem/diamond-pau#173. `LayerZeroFacet.transfer` now enforces a min-receive check on the bridged amount, so a haircut below the floor reverts rather than being silently accepted, and the separate `quoteOFT` call was dropped. The strict `abi.decode` hardening was separately acknowledged; it is a temporary-DoS robustness item.
- **4.8** — Fixed in sky-ecosystem/diamond-pau#190. The refund-to-`msg.sender` call and its hard `require(success)` were removed entirely, eliminating the DoS by construction. The flow now sweeps any residual Controller ETH to the proxy via a new `_sweepETH()` helper and sets `refundAddress = proxy` on `oft.send`, so over-payment — whether by the relayer or as an OFT-side refund — lands in protocol custody; a new public `quoteTransfer` lets allocators pre-compute the exact native fee. Tradeoff: an over-paying relayer no longer receives the excess back, which is acceptable for the documented protocol-funded-relayer pattern.
- **4.9** — Fixed in sky-ecosystem/diamond-pau#176 by the same change as 4.1: `valueWithdrawn` is now derived from the principal-only delta, so the withdraw rate-limit is no longer charged for collected fees, matching `docs/UNIV3_UNIV4_COMPARISON.md`.
- **4.10** — Spark response: *"Acknowledge, this is not an issue — these values are always updated atomically or close to with review."*
- **4.11** — Spark response: *"Acknowledge, this wouldn't really happen in practice and can be addressed with a small addition of liquidity by an external actor."*
- **4.12** — Spark response: *"Ack — hooks can be onboarded safely, they just have to be manually reviewed."*
- **4.13** — Fixed in sky-ecosystem/diamond-pau#183, which corrected the V4 slippage description in `LIQUIDITY_OPERATIONS.md`. Though `CODE_NOTES.md`'s "Slippage Checks" section still states that all liquidity operations require `maxSlippage != 0` (the V4 liquidity functions do not enforce it)
- **4.14** — Spark response: *"Ack — buffer is upgradeable so funds can always be recovered."*
- **4.15** — Spark response: *"Ack — controller is not supposed to be upgraded anymore, and if it was OTC swaps would be paused."*
- **4.16** — Spark response: *"Ack — this will be enforced operationally."*
- **4.17** — Spark response: *"Ack — this is a known piece of functionality and expected."*
- **4.18** — Spark response: *"Not an issue, want to maintain ability to configure just per function."*
- **4.19** — Spark response: *"Same thing, not an issue."*
- **4.20** — Spark response: *"If this is desired functionality it can be added in a subsequent release."*
- **4.21** — Spark response: *"Ack — the contract is assumed not to change, can be updated if needed."*
- **4.22** — Fixed in sky-ecosystem/diamond-pau#170. `FarmFacet.withdraw` no longer unconditionally depends on the reward-claim path, so a reverting or disabled reward claim can no longer brick withdrawals — the DoS coupling is closed. The claim-reward rate-limit key remains presence-gated (a boolean enable flag) rather than a decremented budget; this is non-impactful since reward tokens flow into the proxy, but operators should configure that key as an enable flag, not a budget.
- **4.23** — Spark response: *"Ack — this is not needed for this release, can be added later if desired."*
- **4.24** — Spark response: *"This is intentional, we don't want to block withdrawals in case of a loss heavy event where we need to exit fast."*
- **4.25** — Spark response: Backlog.
- **4.26** — Fixed in sky-ecosystem/diamond-pau#183, which added the rate-limit-key documentation the refactor left behind: the Centrifuge cancel / claim-cancel keys in `docs/SECURITY.md` — including the stuck-state failure mode where a configured cancel key without its matching claim-cancel key wedges the `REQUEST_ID = 0` integration until governance adds the missing key — and a "Try-Increase (Not Gate-Check)" section in `docs/RATE_LIMITS.md` covering the Aave/ERC-4626 deposit-key withdraw precondition.
- **4.27** — Fixed in sky-ecosystem/diamond-pau#186. `FarmFacet.withdraw` now charges the measured staking-token balance delta against the withdraw budget instead of the relayer-supplied amount, matching the project-wide balance-delta accounting convention.
- **4.28** — Spark response: *"This would be enforced operationally during onboarding."*
- **4.29** — Fixed in sky-ecosystem/diamond-pau#174. The per-chunk `require(maxFee < transferAmount)` gate that caused the last-chunk revert was removed as part of the proportional per-chunk `maxFee` refactor (see 4.5).
- **4.30** — Spark response: *"Ack — can recover with an upgrade."*
- **4.31** — Spark response: *"This would be prevented with testing prior to deployment/activation."*
- **4.32** — Spark response: *"Ack, will not be adding for this release."*
- **4.33** — Spark response: *"Ack, will not onboard non18 decimal PTs."*
- **4.34** — Spark response: *"Ack, this is already documented."*
- **4.35** — Spark response: *"Ack, will not be adding for this release."*
- **4.36** — Spark response: *"Ack, assuming constant return shape."*