certora

Security Assessment

Final Report

Sky – Diamond PAU

May 2026

Prepared for Spark Foundation

certora

Table of Contents

certora

Project Summary

Project Scope

Project Name	Repository Link	Commit Hash	Platform
Sky - Diamond PAU	https://github.com/sky-ecosystem/diamond-pau	fc2fe7f (initial) a84fe96 (fix)	Solidity

Project Overview

This document describes the security review of **Sky - Diamond PAU**. The work was undertaken from **April 14th, 2026** to **May 15th, 2026**.

The following contract list is included in our scope:

`src/*`

The team performed a manual audit of all the files in scope**.** During the manual audit, the Certora team discovered bugs in the code, as listed on the following page.

Protocol Overview

The system enables controlled interaction with various DeFi protocols while enforcing rate limits and maintaining custody of funds through the ALMProxy.

Assessment Methodology

Our assessment approach combines design level analysis with a deep review of the implementation to ensure that a protocol is secure, economically sound, and behaves as intended under realistic conditions.

At the design level, we evaluate the architecture, the economic assumptions behind the protocol, and the safety properties that should hold independently of a specific chain or environment. This process includes reviewing internal and cross protocol interactions, state transition flows, trust boundaries, and any mechanism that could be exploited to extract value, deny service, or alter core system behavior. At this stage, a focused threat modelling exercise helps identify key attack surfaces and adversarial capabilities relevant to the system. Design level issues often relate to incentive structures, governance implications, or systemic behavior that emerges under adversarial conditions.

Implementation analysis focuses on the concrete behavior of the code within the execution model of the target chain. This involves reviewing the correctness of logic, access control, state handling, arithmetic behavior, and the nuanced behaviors of the chain environment. Familiar classes of vulnerabilities such as reentrancy conditions, faulty permission checks, precision issues, or unsafe assumptions often surface at this layer. These findings require context aware reasoning that takes into account both the code and the architectural intent.

To support this analysis, the codebase is examined through repeated manual passes and supplemented by automated tools when appropriate. High-risk logic areas receive deeper scrutiny, invariants are validated against both design intent and actual implementation, and potential vulnerability leads are thoroughly investigated. Automated techniques such as static analysis, fuzzing, or symbolic execution may be used to complement manual review and provide additional insight.

Collaboration with the development team plays an important role throughout the audit. This helps confirm expected behaviors, clarify design assumptions, and ensure an accurate understanding of the protocol's intended operation. All findings are documented with clear reasoning, reproducible examples, and actionable recommendations. A follow up review is conducted to validate the applied fixes and verify that no regressions or secondary issues have been introduced.

certora

Threat and Security Overview

System Description

PAU is an on-chain treasury and market-making framework that lets an allocator drive funds held in a single custodial proxy into a whitelisted set of DeFi integrations. It is generalised into a diamond-inspired pattern. A controller is used per to dispatch calls into specialised facets. Funds always remain in the custody of the ALMProxy contract, and every state-mutating allocator action is gated by both a rate limit and (where applicable) a slippage floor.

The custody layer is the proxy. The proxy holds all funds and executes raw calls operations. The routing layer is the Controller together with ControllerSharedStorage, which acts as the entrypoint. It maps incoming selectors to (facet, delegateSelector) pairs.

Above this core sit facets that integrate with different protocols like Aave, Uniswap, Curve and others. They are used to generate yield, swap between assets, bridge assets and other actions.

Trust Boundaries

The DEFAULT_ADMIN_ROLE is fully trusted and is expected to be run by governance. It configures integrations, rate limits and other important values. On the other hand, The ALLOCATOR_ROLE is explicitly modelled as fully compromised at all times; every allocator-reachable code path must remain safe under that assumption.

External protocols sit on a spectrum. For the most part, they are considered to act as currently designed and are trusted to not conduct malicious actions such as upgrading to a malicious version.

Attack Vectors Considered

One of the vectors considered was a compromised allocator and the damage it can cause. The system has been examined against different actions an allocator could take in order to steal funds, lock funds, DoS the system or any other action that can cause the protocol to suffer in some way. Among the attacks considered are reentrancies, malicious inputs, actions at a specific timing (frontrun, backrun, sandwich), price manipulations and others.

Another vector considered was that all happy paths function correctly. Since the only role that can freely potentially cause damage is a compromised allocator, then that leaves 2 main big vectors - malicious actions by allocator as explained above and non-malicious, regular actions by admins/allocators/freezers/etc. The happy paths were thoroughly check to function exactly as intended as they are how the system will be interacted with for the most part.

Another vector considered was whether an unrelated party can cause damage to the system. This includes, but is not limited to, abusing missing/incomplete access controls, frontruns/backruns/sandwiches of different allocator actions to manipulate price or skew execution terms in some way, directly interacting with a third-party protocol through potentially permissionless functionality (i.e. claiming WEETH withdrawal on behalf of proxy which would stuck funds in the WETTHModule if it was a permissionless claim) to cause unintended and unhandled paths.

With the attacks considered, a few potential issues arose. A Medium severity issue which could cause the Uniswap V3 liquidity removal to DoS if accumulated fees grow sufficiently large, as well as multiple Low severity issues which are mostly related to malicious actions a compromised relayer can take to cause damage to the system.

certora

Findings Summary

The table below summarizes the findings of the review, including type and severity details.

Severity	Discovered	Confirmed	Fixed
Critical	-	-	-
High	-	-	-
Medium	1	1	1
Low	13	13	6
Informational	3	3	0
Total	**17**	**17**	**7**

Severity Matrix

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
			Likelihood	

certora

Detailed Findings

ID	Title	Severity	Status
M-01	UniswapV3Facet removeLiquidity can DoS due to accrued fees	Medium	Fixed
L-01	`minTokenOut` is either useless or blocking in an edge case	Low	Acknowledged
L-02	`OTCFacet` whitelist does not distinguish sendable and claimable assets	Low	Fixed
L-03	Swap proceeds can become mixed with low amounts in `OTCFacet`	Low	Acknowledged
L-04	Single-sided slippage check on `UniswapV3Facet.addLiquidity` can be bypassed via imbalanced target amounts	Low	Acknowledged
L-05	Compromised allocator can trade around Maple unrealized losses to harm ALM	Low	Fixed
L-06	Missing claim path for Aave's RewardsController incentive rewards	Low	Acknowledged
L-07	UniswapV3 expected amount logic is inaccurate	Low	Fixed
L-08	Compromised allocator can game stepwise share-price updates in Basin	Low	Acknowledged

L-09	Malicious allocator can grief the ALM by routing funds into a compromised pocket	Low	Acknowledged
L-10	`FarmFacet.claimReward` missing rate-limit whitelist gate	Low	Fixed
L-11	`CCTPFacet` design flaws enable fee griefing and stuck funds when Circle activates fees on standard transfers	Low	Fixed
L-12	Compromised allocator can front-run Centrifuge exchange rate decreases via sync vault deposits	Low	Acknowledged
L-13	FarmFacet `claimReward` rate limit check is not checked via withdraw	Low	Fixed
I-01	Aave slippage parameter is redundant	Informational	Acknowledged
I-02	Excess ETH from Centrifuge cross-chain transfers is not refunded to the allocator	Informational	Acknowledged
I-03	Hardcoded zero `remoteExtraGasLimit` in Centrifuge cross-chain share transfers	Informational	Acknowledged

Medium Severity Issues

M-01: UniswapV3Facet removeLiquidity can DoS due to accrued fees

Severity: Medium	Impact: Medium	Likelihood: Medium
Files: src/facets/uniswap-v3/UniswapV3Facet.sol	Status: Fixed	

Description:

The removeLiquidity function in UniswapV3Facet reuses the same min.amount0 and min.amount1 values for two conflicting purposes.

1. First, they are passed to Uniswap V3's `decreaseLiquidity`, which enforces that the returned principal (excluding fees) satisfies `principal >= min`.
2. Second, after calling collect (which sweeps both principal and all accumulated fees into the proxy), the function measures the total received amount via a before/after balance difference and runs `_checkSlippage` against it, enforcing `min >= (principal + fees) * maxSlippage / 1e18`.

These two constraints are incompatible when fees grow large relative to the principal. Combining them requires:

```
principal >= (principal + fees) * maxSlippage / 1e18
```

For a stablecoin pool with maxSlippage set to 0.995e18 (0.5% max deviation), the maximum tolerable fee-to-principal ratio is just ~0.5%. Once accumulated fees exceed that threshold, no valid min value satisfies both constraints simultaneously, and removeLiquidity always reverts.

This issue was previously identified in the Cantina v1.11.0 audit (finding 3.1.1, Medium) and the Unvariant v1.11 audit (M-01). It was fixed then by running the slippage check against the return values of decreaseLiquidity (principal only). The latest diff reintroduced the bug by migrating to a before/after balance-check pattern, which captures principal + fees in a single measurement and uses that combined amount for the slippage check.

Recommendations:

Go back to obtaining the principal amounts from the return value of decreaseLiquidity and use those for the slippage check, while keeping the balance-difference amounts for rate limit accounting.

Customer Response:

Fixed in PR#176.

Fix Review:

Fix confirmed.

Low Severity Issues

L-01: `minTokenOut` is either useless or blocking in an edge case

Severity: Low	Impact: Low	Likelihood: Low
Files: src/facets/pendle/PendleFacet.sol	Status: Acknowledged	

Description:

The PendleFacet computes a minTokenOut as follows:

```
uint256 minTokenOut = pyAmountIn * 1e18 / IYTLike(yt).pyIndexCurrent() - 5;
```

For most markets, this is useless as this is just how Pendle computes the SY amount out which represents shares (yield tokens) 1:1 when the SY is redeemed to the shares/yield tokens. Thus, this just computes exactly what we would get and then subtracts 5, it is essentially a check that would always pass.

In an edge case, this can be blocking, the following conditions have to be true:

- the SY tokens are not redeemed 1:1 for the yield token, i.e. `aTokens` as are multiplied by the Aave index.
- the index tracking the exchange rate is less than 1e18 (less than 1), i.e. some bad debt event (doesn't apply to Aave as the index there is non-decreasing).

In such a case, the computed minTokenOut is higher than the actual tokens out as the facet code computes the SY tokens out and the actual tokens out are less than the SY tokens due to the index being under 1. Thus, until the index goes >= 1e18 (or possibly slightly less due to the subtraction of 5), this will always revert.

Recommendations:

Consider whether the slippage check is necessary as it simply recomputes what Pendle already computes and furthermore, there doesn't seem to be a way for a malicious actor to manipulate

anything in the process to cause an unfavourable redeem). It can also cause reverts in the edge case mentioned. Also consider whether the minAmountOut is necessary as well due to the fact that no one can maliciously intervene.

Customer Response:

Acknowledged.

L-02: `OTCFacet` whitelist does not distinguish sendable and claimable assets

Severity: Low	Impact: Low	Likelihood: Low
Files: src/facets/otc/OTCFacet.sol	Status: Fixed	

Description:

OTCFacet uses a single per-exchange asset whitelist to gate both directions of an OTC swap. The mapping used for this (assetWhitelisted) is read by both the allocator-facing entry points, namely send and claim.

Whitelisting an asset therefore enables it as both a sendable asset (transferred from the ALMProxy to the exchange) and a claimable asset (pulled from the buffer into the ALMProxy). The whitelist carries no notion of direction.

The current operational model assumes that every onboarded exchange supports bi-directional swaps, so whitelisting an asset for both sides is the desired behaviour. The issue only surfaces if an exchange ever onboards a one-directional venue, e.g. one that only accepts USDC in exchange for USDS but cannot accept USDS as an input.

In that scenario, both assets still need to be whitelisted (USDC so it can be sent, USDS so it can be claimed from the buffer), but the whitelist structure also lets a compromised allocator call send(exchange, USDS, amount). Since USDS is whitelisted, the call passes the gate, decrements the OTC rate limit, and the `ALMProxy` transfers USDS to an exchange that has no path to return it. The funds become stuck at the exchange address and can only be recovered if the specified exchange address has a recovery path for unexpected tokens.

This issue can only happen for exchanges that only support one-directional swaps, which contradicts the protocol's stated onboarding assumption. However, if future exchange integrations drop the bi-directional assumption, the structure as-is silently permits this attack.

Recommendations:

We recommend keeping the current single whitelist, but document the assumption explicitly in OPERATIONAL_REQUIREMENTS.md and reject onboarding of one-directional exchanges at the governance level.

If the protocol ever desires to onboard OTC exchanges that only support one-directional swaps, we recommend splitting the whitelist into two direction-specific mappings, for example, assetSendable and assetClaimable, and updating the rest of the facet to support these new mappings.

Customer Response:

Fixed in PR #172.

Fix Review:

Fix confirmed.

L-03: Swap proceeds can become mixed with low amounts in `OTCFacet`

Severity: Low	Impact: Low	Likelihood: Low
Files: src/facets/otc/OTCFacet.sol	Status: Acknowledged	

Description:

OTCFacet gates a new OTC send on isSwapReady, which requires the previous cycle to be considered closed, either by a direct claim or by a time-based recharge:

```
function isSwapReady(address exchange) public view override returns (bool) {
    // ...

    return
        getClaimWithRecharge(exchange) >=
        $.states[exchange].normalizedSent * maxSlippage / 1e18;
}
function getClaimWithRecharge(address exchange) public view override returns
(uint256) {
    // ...

    return
        state.normalizedClaimed +
        (block.timestamp - state.sentTimestamp) *
$.parameters[exchange].normalizedRate;
}
```

The recharge term is an absolute quantity in token units per second. The threshold on the right-hand side scales linearly with normalizedSent, i.e., with the size of the last send. The unlock time therefore shrinks in direct proportion to how small the previous send was. With the fork test configuration used by the test suite (rechargeRate = 1_000_000e18 / 1 days, maxSlippage = 0.9995e18), a full 10M send unlocks in about 10 days, but a 1,000 USDS drip unlocks in about 86 seconds, and a 100 USDS drip in about 9 seconds.

This behaviour, together with a lack of a minimum swap amount in send, allows a malicious allocator to cycle small sends that are recharged at the next block (i.e., isSwapReady returns true at the next block). This is an issue because we can have lots of pending swap proceeds while the facet still allows the allocator to start a new OTC swap.

The impact doesn't seem to be significant, but three effects are worth noting:

1. First, since `normalizedClaimed` is reset to zero on every send, buffer deposits from prior cycles (e.g., slow OTC desk settlement) are silently credited to whichever cycle happens to be active at claim time, so the per-cycle `isSwapReady` check no longer cleanly ties a send to its own return.

2. Second, `getState(exchange)` exposes only the latest cycle, so a monitor that watches `normalizedSent` sees the tiny drip value while the true outstanding exposure at the exchange is the sum of every unreturned cycle.

3. Third, if governance rotates the buffer via `setBuffer` while the exchange is still delivering the previous swap's proceeds, those proceeds land in the old buffer and are not reachable through the new buffer's `otcClaim` path. Those funds are recoverable, either by rotating `setBuffer` back to the old buffer and calling `otcClaim`, or by UUPS-upgrading the old buffer to add a rescue function, but neither recovery is automatic.

Recommendations:

The simplest fix is to add a minimum size check inside send that ensures a minimum amount of tokens that have to be sent for a single swap.

As an alternative, if allowing small sends is an intentional operational feature, the behaviour can be acknowledged and documented instead.

Customer Response:

Acknowledged.

L-04: Single-sided slippage check on `UniswapV3Facet.addLiquidity` can be bypassed via imbalanced target amounts

Severity: Low	Impact: Medium	Likelihood: Low
Files: src/facets/uniswap-v3/UniswapV3Facet.sol	Status: Acknowledged	

Description:

When validating an addLiquidity call, UniswapV3Facet computes an expected amount per token using the pool's TWAP tick and then requires each supplied min.amount to be at least expectedAmount * maxSlippage / 1e18. If the TWAP tick sits at or past one of the position's boundary ticks, the facet considers the opposite token economically irrelevant and sets its expected amount to zero. When that happens, _validateMinAmount enforces minAmount == 0 without any further bound:

```
function _validateMinAmount(uint256 minAmount, uint256 expectedAmount, uint256
maxSlippage)
    internal
    pure
{
    if (expectedAmount == 0) {
        require(minAmount == 0, "UniswapV3Facet/min-amount-below-bound");
        return;
    }

    // ...
}
```

The flaw is that the facet compares the allocator's intent against the TWAP tick, but the Uniswap V3 mints liquidity at the pool's real spot tick. A single-block swap can move spot many ticks without moving the day-long TWAP.

A compromised allocator can therefore pick ticks.lower = twapTick (keeping the facet in the single-sided branch), supply an imbalanced target where target.amount1 >> target.amount0, and

then sandwich the call by pushing spot close to ticks.upper before the deposit and reverting the swap after.

Uniswap V3 consumes most of target.amount1 at the manipulated spot while amount0 remains bounded by the allocator's own target.amount0, so the facet's slippage check passes. The freshly minted position is worth materially less once spot reverts to peg, with the difference accruing to the attacker.

The impact is bounded by how wide governance configures liquidityTickBounds for each pool. Under the test-base configuration (liquidityTickBounds = initTick +/- 1000, maxSlippage = 0.98e18), the empirical per-call extraction reaches roughly 248 bps of the USDT leg, refillable each rate-limit window. With narrow governance bounds (for example +/- 50 ticks), the per-call extraction drops a lot, and once MEV costs (swap fees paid to existing LPs plus back-run slippage) are accounted for, the vector becomes unprofitable in practice.

The protocol relies on the same governance-tightening mitigation for UniswapV4Facet, which has no TWAP branching at all and uses raw amount0Max / amount1Max caps bounded only by maxSlippage and operator-set tick ranges.

Recommendations:

The real mitigation to these kinds of attacks is the one already applied to V4, which requires governance to configure tight tick ranges per pool so the per-call extractable value stays below the attacker's cost of manipulation.

Consider removing the slippage validation on addLiquidity and documenting the tick-range expectation alongside the existing operational requirements for V4. Alternatively, acknowledge the issue and keep the check as defense in depth, noting in the threat model that its effective protection depends on liquidityTickBounds being kept narrow.

Customer Response:

Acknowledged.

L-05: Compromised allocator can trade around Maple unrealized losses to harm ALM

Severity: Low	Impact: Medium	Likelihood: Low
Files: src/facets/maple/MapleFacet.sol	Status: Fixed	

Description:

Maple Finance pools have a documented impairment lifecycle (see Maple's "Defaults and Impairments" docs): when a loan is flagged by the pool delegate, the pool's unrealizedLosses accumulator is incremented while totalAssets is left unchanged. The impairment is then resolved either by a recovery (loan repays, unrealizedLosses drops back to zero, no NAV change) or by a realization (the loss is moved from unrealizedLosses into totalAssets, permanently lowering the pool's NAV).

The asymmetry that matters here is in Maple's pricing functions. Deposits price shares using convertToShares, which uses the gross totalAssets and is therefore unchanged by an active impairment. Redemptions price shares using convertToExitAssets(shares), which is decreased during impairment. Neither the PAU MapleFacet nor the generic ERC4626Facet that PAU uses to deposit into Maple pools consults unrealizedLosses() before performing an action.

Because the threat model assumes the allocator can be compromised, and an active impairment is public information, a malicious allocator can deterministically pick the harmful side of the trade depending on what they expect to happen next:

1. **Impairment that will be reversed**: the allocator calls `MapleFacet.requestRedemption`. Once the cyclical withdrawal manager processes the request, the assets returned to the ALMProxy use the depressed `convertToExitAssets` rate (the impairment is still active at the time). When the loan later recovers and `unrealizedLosses` returns to zero, the rest of the pool's lenders absorb the recovery upside that ALM should have received. ALM's exit price is permanently locked in at the impaired level.

1. **Impairment that will be realized**: the allocator calls `ERC4626Facet.deposit` against the Maple pool token. Shares are minted using the undiscounted `totalAssets` price (the

impairment has not yet been realized, so the pool still books the loaned principal at full value). When the default is realized, `totalAssets` drops by the loss amount, and ALM's freshly minted shares immediately lose value.

Recommendations:

Block the allocator from depositing into or requesting a redemption from any Maple pool whose unrealizedLosses are non-zero.

A simpler operational alternative is for governance to zero the LIMIT_MAPLE_REDEEM and LIMIT_4626_DEPOSIT keys for the Maple pool token as soon as an impairment is announced, and only restore them after resolution. This avoids a code change but pushes the burden onto a governance process that has to react quickly and stay paused for what might be months at a time.

Customer Response:

Documented in PR #167.

L-06: Missing claim path for Aave's RewardsController incentive rewards

Severity: Low	Impact: Low	Likelihood: Medium
Files: src/facets/aave/AaveFacet.sol	Status: Acknowledged	

Description:

AaveFacet exposes only deposit and withdraw. There is no entry point that calls RewardsController.claimRewards / claimAllRewards on behalf of the proxy, so any incentives accrued on aTokens held by the proxy cannot be claimed through the allocator flow.

This is not hypothetical. On-chain inspection of the mainnet RewardsController (0x8164Cc65827dcFe994AB23944CBC90e0aa80bFcb) at audit time shows an active emission on aEthUSDS-Core (~0.02 aEthUSDS/s), with `distributionEnd` rolling forward in 2-day extensions. Any USDS the proxy supplies into that market today accrues rewards that no facet can claim.

Recommendations:

Add an allocator-callable function to AaveFacet that routes RewardsController.claimRewards (or claimAllRewards) through ALMProxy.doCall with the proxy as recipient.

Customer Response:

Acknowledged.

L-07: UniswapV3 expected amount logic is inaccurate

Severity: Low	Impact: Low	Likelihood: Low
Files: src/facets/uniswap-v3/UniswapV3Facet.sol	Status: Fixed	

Description:

When computing the expected amounts, the UniswapV3 facet has some mistakes which cause the amounts to be computed incorrectly. The facet does this in _getExpectedAmounts:

```
if (twapTick <= ticks.lower) {
    expectedAmount0 = UniswapV3Utils.getAmount0Delta(
        sqrtRatioLowerX96,
        sqrtRatioUpperX96,
        expectedLiquidity,
        false
    );
}
```

However, the actual Uniswap logic does the following:

```
if (_slot0.tick < params.tickLower) {
    // current tick is below the passed range; liquidity can only become in range
by crossing from left to
    // right, when we'll need _more_ token0 (it's becoming more valuable) so user
must provide it
    amount0 = SqrtPriceMath.getAmount0Delta(
        TickMath.getSqrtRatioAtTick(params.tickLower),
        TickMath.getSqrtRatioAtTick(params.tickUpper),
        params.liquidityDelta
    );
```

Firstly, the sign is < instead of <=. Secondly, false is provided as the last input in the facet logic, however Uniswap provides true (liquidity over 0):

```
liquidity < 0
    ? -getAmount0Delta(sqrtRatioAX96, sqrtRatioBX96, uint128(-liquidity),
false).toInt256()
    : getAmount0Delta(sqrtRatioAX96, sqrtRatioBX96, uint128(liquidity),
true).toInt256();
```

Recommendations:

Consider removing the = sign in the check and provide true as the roundUp input.

Customer Response:

Fixed in PR #169.

Fix Review:

Fix confirmed.

L-08: Compromised allocator can game stepwise share-price updates in Basin

Severity: Low	Impact: Medium	Likelihood: Low
Files: src/facets/basin/BasinFacet.sol	Status: Acknowledged	

Description:

The GroveBasin share price doesn't always drift continuously, it sometimes jumps in discrete steps when `_completeRedeem` runs or when the rate of an asset in the pool is updated. These step events are observable (mempool or scheduled), and the threat model treats the allocator as compromisable.

A compromised allocator can game each share update by frontrunning it and causing the maximum harm to the proxy's funds:

1. **UP jump (yield denial)**: The allocator can withdraw before the share price rises, and redeposit after. The proxy isn't in the basin during the appreciation, so it gets none of it; the forfeited yield is split pro-rata among the remaining shareholders, which will only be the fee receiver and the `address(0)` (due to the first vault deposit).

2. **DOWN jump (loss amplification)**: The allocator can maximize the proxy's position right before a loss is recognized, withdraw immediately after, causing the proxy to eat an outsized share of the loss. Though this amplification usually won't be significant because the proxy is always the majority shareholder in the basin.

Recommendations:

Given the limited impact, we recommend acknowledging the issue and monitoring the allocator's behaviour to ensure a malicious allocator is replaced before causing significant harm.

Customer Response:

Acknowledged.

L-09: Compromised allocator can amplify exposure to a failing external yield strategy

Severity: Low	Impact: Medium	Likelihood: Low
Files: src/facets/basin/BasinFacet.sol	Status: Acknowledged	

Description:

A GroveBasin holds three assets (collateral, credit, and swap token). When the basin is configured to use the optional "pocket", the swap token is routed through that to an underlying yield strategy.

The three audited pockets supply USDT to Aave V3, supply USDT to a MetaMorpho vault, and route USDC through the Sky PSM, respectively. The pocket's reported balance feeds directly into the basin's totalAssets, so any step-down in the underlying yield source becomes a step-down in the basin share price. This step-down would only apply to external yield sources like Aave V3 and MetaMorpho, not USDC in the Sky PSM.

BasinFacet exposes a generic deposit/withdraw on the allocator surface. A compromised allocator can concentrate the ALM's exposure into the swap-token leg right before a predictable degradation of the pocket's underlying strategy. The "compromised pocket" trigger is any predictable adverse event in the underlying strategy: bad debt about to be realized in a Morpho Blue market backing a MetaMorpho vault, an Aave reserve becoming illiquid for an extended period (similar to recent stress incidents like Kelp DAO), etc. This "compromised pocket" risk is the same deposit/withdraw risk as other facets that directly integrate with external yield strategies.

When the underlying strategy of a pocket realizes a loss, the ALM absorbs the bad-debt fraction on the entire concentrated swap-token exposure rather than only on the pre-attack swap-token holdings. When the underlying strategy becomes illiquid rather than insolvent (the Aave case), the harm is instead a liveness loss: the deposited tokens in that pocket cannot be withdrawn

until the underlying re-liquefies, which can be hours to days, or longer if the underlying issue is persistent.

This is not specific to Basin, any facet that deposits into an external yield strategy (e.g., AaveFacet, ERC4626Facet) carries the same risk of a compromised allocator amplifying exposure before a predictable adverse event.

Recommendations:

Implement the following mitigations:

1. When considering the rate-limit risk mitigation strategy, ensure the external yield strategy is included and rate limits are set at levels Governance is comfortable with for those external yield sources.
2. Monitor each pocket's underlying yield source off-chain (Morpho vault market exposures, Aave reserve utilization, etc) and revoke rate limits on the swap token when the strategy becomes risky.
3. Document this risk in the Basin onboarding docs so governance explicitly accounts for external yield strategy health before onboarding exposure to those strategies directly or through a Basin Pocket.

Customer Response:

Acknowledged, will add documentation for this.

L-10: `FarmFacet.claimReward` missing rate-limit whitelist gate

Severity: Low	Impact: Low	Likelihood: Low
Files: src/facets/farm/FarmFacet.sol	Status: Fixed	

Description:

FarmFacet.claimReward does not enforce a rate-limit whitelist check. The function accepts any farm address and directly calls _claimReward without verifying whether the farm is associated with a configured rate-limit key.

In contrast, both FarmFacet.deposit and FarmFacet.withdraw invoke _decreaseRateLimit, which reverts if no rate-limit key is set, effectively enforcing the whitelist requirement. claimReward lacks this safeguard entirely.

As a result, a compromised allocator could call claimReward on arbitrary addresses, breaking the "whitelist enforced" invariant described in the THREAT_MODEL document.

That said, no concrete exploit leading to significant impact has been identified. The only observed consequence is the potential for incorrect or misleading event emissions.

Recommendations:

Add a rate-limit existence check at the beginning of claimReward, consistent with the pattern used by other facets.

Customer Response:

Fixed in 0f3a1ef.

Fix Review:

Fix confirmed.

L-11: `CCTPFacet` design flaws enable fee griefing and stuck funds when Circle activates fees on standard transfers

Severity: Low	Impact: Medium	Likelihood: Low
Files: src/facets/cctp/CCTPFacet.sol	Status: Fixed	

Description:

CCTPFacet.transferWithFee allows a allocator to specify maxFee, which is the ceiling that Circle's Iris attestation service can charge on the destination chain. The facet validates maxFee <= maxFeeCap once per call, then enters a chunking loop where each depositForBurn receives the same maxFee independently. This design has several compounding problems that surface the moment Circle introduces non-zero fees for standard (finality=2000) transfers.

All vectors are dormant today because Circle doesn't charge fees for standard transfers today, but it will activate the fee switch in the future, according to their documentation.

The sub-issues share a single precondition (Circle's fee switch activation for standard transfers) and interact with each other:

1. **maxFeeCap should be in percentage terms**.

The check maxFee <= maxFeeCap is denominated in USDC, independent of the transfer amount. A allocator can bridge maxFeeCap+1 with maxFee=maxFeeCap, which indicates CCTP that the allowed maximum fee can be nearly 100% of the transfer amount. The cap offers no proportional protection: a 100 USDC cap permits 99.99% fee on a 100.000001 USDC transfer but only 0.01% on a 1M USDC transfer.

1. **The chunking loop multiplies the effective cap.**

When amount exceeds burnLimitsPerMessage, the facet splits into N chunks. Each chunk independently passes the same maxFee to depositForBurn, and Circle treats each as a separate burn message with its own fee ceiling. The effective per-call fee cap becomes ceil(amount / burnLimit) * maxFeeCap.

A single transferWithFee(50_000e6, 100e6, base) call with a 10k burn limit produces 5 chunks, each authorizing 100 USDC in fees, for a total of 500 USDC. This exceeds the admin's configured 100 USDC cap by 5x without violating any on-chain check.

1. **No minimum-fee floor enables stuck funds.**

When Circle activates source-side minFee for standard transfers, but the actual feeExecuted for the destination is higher, a compromised allocator can set maxFee to the source floor. The source burns USDC successfully, but the actual fee exceeds the maximum allowed by the allocator. The destination then reverts with "*Fee exceeds max fee*" and the funds are locked on the source chain.

This requires that the value of minFee on the source chain be slightly lower than the actual fee executed on the destination chain, but this already happens today for fast transfers, so we can assume it will hold similarly for standard transfers when the fee switch is activated there.

1. **maxFeeCap is global instead of per-domain.**

• Different destination chains have different fee percentages (link), ranging from 1 bps to 14 bps. A single global cap cannot simultaneously be tight enough to protect against low-fee routes while permitting transfers on higher-fee routes. This forces governance to set the cap to the maximum across all routes, leaving cheaper routes under-protected.

Recommendations:

• For issues 1 and 2, express the fee cap as a basis-point fraction of amount rather than an absolute value.

• For issue 3, add a per-domain minimum floor on `maxFee` set by governance. This prevents an allocator from deliberately underbidding and causing the funds to be stuck on the destination chain.

• For issue 4, the fee caps (minimum and maximum) should be configured per destination domain rather than using a single range for all domains (chains).

Customer Response:

Fixed in PR #174.

Fix Review:

Fix confirmed.

L-12: Compromised allocator can front-run Centrifuge exchange rate decreases via sync vault deposits

Severity: Low	Impact: Medium	Likelihood: Low
Files: src/facets/centrifuge/CentrifugeFacet.sol	Status: Acknowledged	

Description:

Centrifuge vaults can operate in two modes regarding deposits: synchronous (ERC-4626) and asynchronous (ERC-7540). When a Centrifuge vault is configured as a sync vault, deposits are processed through the ERC4626Facet, which grants shares immediately at the current on-chain exchange rate. In contrast, async vaults process deposits through the ERC7540Facet.

A compromised allocator can exploit the sync deposit path by timing deposits into a Centrifuge vault just before an exchange rate decrease is applied on the spoke. This is particularly relevant for junior tranches, where exchange rate volatility is higher due to their first-loss position. If the allocator is also personally invested in the same vault, they have a direct financial incentive to deposit ALM funds before the rate drops: the additional capital absorbs a proportional share of the impending loss, effectively socializing the allocator's personal losses across the PAU system.

The impact is bounded by the PAU rate limits configured for the vault's deposit key and by the vault's own maxReserve cap. While neither of these limits prevents the attack entirely, they constrain the maximum amount of capital that can be deployed in a single rate limit window.

Recommendations:

Configure Centrifuge integrations to use vaults only with asynchronous deposits (ERC-7540). If sync vaults must be used, restrict them to senior tranches where exchange rate decreases are less frequent and less severe.

Customer Response:

Acknowledged.

L-13: FarmFacet `claimReward` rate limit check is not checked via withdraw

Severity: Low	Impact: Low	Likelihood: Low
Files: src/facets/farm/FarmFacet.sol	Status: Fixed	

Description:

In FarmFacet, the claimReward function verifies that a claim-reward rate limit has been configured for the given farm before proceeding:

```
function claimReward(address farm)
    external
    override
    nonReentrant
    onlyRole(ALLOCATOR_ROLE)
    returns (uint256 reward)
{
    require(_rateLimitExists(getClaimRewardRateLimitKey(farm)),
"FarmFacet/invalid-action");
    return _claimReward(farm);
}
```

However, the withdraw function also claims rewards by calling the same internal _claimReward helper at the end of its execution, without checking whether the claim-reward rate limit exists:

```
function withdraw(address farm, uint256 amount)
    external
    override
    nonReentrant
    onlyRole(ALLOCATOR_ROLE)
    returns (uint256 reward)
{
    _decreaseRateLimit(getWithdrawRateLimitKey(farm), amount);

    IALMProxy(_getSharedControllerStorage().proxy).doCall(
```

```
        farm,
        abi.encodeCall(IFarmLike.withdraw, (amount))
    );

    emit FarmWithdraw(farm, amount);

    return _claimReward(farm);
}
```

This means the allocator can bypass the claimReward rate limit existence check by calling withdraw instead. If a farm has its withdraw rate limit configured but not its claim-reward rate limit, rewards can still be claimed through the withdrawal path.

Recommendations:

Move the _rateLimitExists check for the claim-reward key inside the _claimReward helper function. Alternatively, skip the reward claim inside withdraw when the claim-reward rate limit has not been set up for the given farm.

Customer Response:

Fixed in PR #170.

Fix Review:

Fix confirmed.

Informational Severity Issues

I-01: Aave slippage parameter is redundant

Files:

src/facets/aave/AaveFacet.sol

Description:

AaveFacet.deposit performs a post-supply slippage check based on the aToken balance delta on the proxy:

```
uint256 aTokenBalance = IERC20Like(aToken).balanceOf(proxy);

IALMProxy(proxy).doCall(
    pool,
    abi.encodeCall(IPoolLike.supply, (underlying, amount, proxy, 0))
);

uint256 newATokens = IERC20Like(aToken).balanceOf(proxy) - aTokenBalance;

require(newATokens >= amount * maxSlippage / 1e18,
"AaveFacet/slippage-too-high");
```

In practice, the supply function on Aave always returns aTokens 1:1 with the underlying (within ±2 wei of rounding), so the check can never observe negative slippage on a successful deposit, except for the mentioned dust. The maxSlippage parameter therefore does no real work as a slippage guard, its only operative effect is the maxSlippage == 0 revert at line 98, which acts as a per-aToken deposit pause.

Recommendations:

Preferred: remove the slippage check and the maxSlippage mapping/setter entirely. Pausing deposits is already expressible via the rate-limit configuration (LIMIT_DEPOSIT.maxAmount == 0), which is the standard mechanism across PAU.

Alternatively, acknowledge the redundancy and leave the code as-is.

Customer Response:

Acknowledged.

I-02: Excess ETH from Centrifuge cross-chain transfers is not refunded to the allocator

Files:

src/facets/centrifuge/CentrifugeFacet.sol

Description:

The CentrifugeFacet.transferShares function allows the allocator to initiate a cross-chain share transfer via the Centrifuge Spoke. The allocator sends ETH along with the call to cover cross-chain messaging fees. The entire msg.value is forwarded to the ALMProxy, which then calls crosschainTransferShares on the Spoke.

If the Spoke's underlying cross-chain messaging layer consumes less ETH than the full msg.value, any refund is sent back to the caller of crosschainTransferShares, which is the ALMProxy. The facet does not implement any mechanism to return this excess ETH to the allocator (link).

This is inconsistent with the LayerZeroFacet, which handles the same scenario correctly. The impact is that any overpayment by the allocator accumulates in the ALMProxy instead of being returned.

Recommendations:

Add a refund mechanism to CentrifugeFacet.transferShares that returns excess ETH to the allocator after the cross-chain call completes, matching the pattern already used in LayerZeroFacet.

Alternatively, acknowledge the issue and document it so allocators are aware of it.

Customer Response:

Acknowledged. The ALMProxy can use wrapProxyETH to wrap it into WETH and then transfer it to any address necessary if gas is really needed that bad, but this is unexpected.

I-03: Hardcoded zero `remoteExtraGasLimit` in Centrifuge cross-chain share transfers

Files:

src/facets/centrifuge/CentrifugeFacet.sol

Description:

In CentrifugeFacet, the transferShares function initiates a cross-chain transfer of vault shares by calling crosschainTransferShares on the Centrifuge Spoke contract. This function accepts a remoteExtraGasLimit parameter that specifies additional gas to allocate for execution on the destination chain beyond the default gas limit.

```
    // Initiate cross-chain transfer via the specific spoke address.
    IALMProxy(_getSharedControllerStorage().proxy).doCallWithValue{value:
msg.value}(
        spoke,
        abi.encodeCall(
            ISpokeLike.crosschainTransferShares,
            (
                centrifugeId,
                ICentrifugeV3VaultLike(token).poolId(),
                ICentrifugeV3VaultLike(token).scId(),
                recipient,
                amount,
>>              0            // @audit hardcoded `remoteExtraGasLimit`
            )
        ),
        msg.value
    );
```

The facet hardcodes this parameter to 0, meaning every cross-chain share transfer relies entirely on the default gas budget provided by the Centrifuge messaging layer. If a destination chain's share token includes transfer hooks or additional logic that consumes more gas than the default allowance, the cross-chain message could fail on the remote side.

In practice, this is unlikely to be an issue as long as the vault shares being transferred are standard ERC-20 tokens without gas-intensive hooks. However, if a future Centrifuge pool

integration uses shares with custom transfer logic, the current implementation provides no way to accommodate the extra gas requirement.

Recommendations:

If no vault shares with gas-intensive hooks are planned to be used, acknowledge this limitation and document it as an operational constraint.

Otherwise, allow the admin role to configure a fixed remoteExtraGasLimit value per centrifugeId (similar to the existing recipients mapping), and pass it to the `crosschainTransferShares` call instead of the hardcoded zero.

Customer Response:

Acknowledged.

Disclaimer

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline and is widely used by developers and security researchers during audits and bug bounties.

Certora also provides architectural design reviews, where researchers analyze system design and trust boundaries early to identify structural risks and reduce complexity before implementation.

Certora conducts off-chain audits covering backend services, APIs, frontend, and mobile applications, focusing on vulnerabilities in authentication, data handling, key management, and wallet-related workflows.

Certora also offers Penetration Testing and Red Teaming to simulate real-world attacks and uncover exploitable weaknesses across infrastructure, applications, and business logic.

In addition, Certora provides operational security (OpSec) assessments, reviewing key management, access controls, and operational processes to strengthen overall security posture.

Finally, Certora delivers ongoing monitoring and incident response, enabling continuous threat detection and prevention, alert triage, and coordinated response to active incidents.