PUBLIC

Code Assessment

of the Diamond PAU v1.13 Smart Contracts

June 04, 2026

Produced for

Sky

by

CHAINSECURITY

Contents

1 Executive Summary

Dear all,

Thank you for trusting us to help Sky with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Diamond PAU v1.13 according to Scope to support you in forming an opinion on their security risks.

Sky implements Diamond PAU (Parallelized Allocation Units), a modular, facet based architecture that replaces the legacy ALM Controller. Integrations are deployed as standalone facet contracts and registered in a shared Beacon; each deployment's Controller syncs the relevant integration configs, routes allocator calls through its selector dispatch table, and relies on dedicated AccessControls and RateLimits contracts for authorization. Funds remain in the deployment's ALMProxy, and existing legacy ALMProxy instances can be reused by authorizing the new Controller as the proxy's controller.

The latest reviewed iteration, `v1.13.0-beta.0`, refactors the `PAUFactory` from an atomic full-system deployer into a set of per-component deployers that produce contracts with the expected bytecode where wiring is now performed by governance after deployment rather than atomically by the factory. The preceding `v1.12.0` performed the final adjustments preparing the codebase for release, and the trust model under which upgradeable integrations are treated as potentially malicious was finalized in an earlier release candidate.

The most critical subjects covered in our audit are functional correctness, asset solvency, access control, and external integrations. The general subjects covered are code consistency and documentation.

The most notable findings within this report include violations of functional correctness (e.g. Conflicting slippage constraints block UniswapV3Facet.removeLiquidity) and asset solvency (e.g. Dynamic Call Targets Can Bypass Integration Boundaries). These findings have been resolved. The remaining open findings, the medium-severity Permissionless asset inflows inflate rate limits and some lower-severity ones, have been acknowledged, with the exception of Reentrancy guard is per-Controller while the ALMProxy is shared, which was raised in the latest reviewed version.

In summary, we find that the codebase provides a good level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered, and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	4
• Code Corrected	3
• Risk Accepted	1
Low-Severity Findings	13
• Code Corrected	6
• Risk Accepted	6
• Acknowledged	1

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commits which are referenced throughout this report.

2.1 Scope

This review covers the changes introduced up to `v1.13.0-beta.0` of the Diamond PAU (Parallelized Allocation Units) codebase, a modular refactoring of the merged ALM Controller (`v1.11.0`) into a diamond proxy architecture where each integration is packaged as a separate facet contract, registered in a shared `Beacon`, and dispatched to by a per deployment `Controller`. Relative to `v1.12.0`, `v1.13.0-beta.0` refactors the `PAUFactory` from an atomic full-system deployer into a set of per-component deployers.

While most integrations have been reviewed in separate reports (see Spark ALM Controller v1.7.0 Review, v1.10.0 Diff Review and Grove ALM Controller Review v1.8.0), the Uniswap V4, WSTETH and OTCSwap integrations are assumed to be correct and secure since they are not audited by ChainSecurity. However, their diffs were reviewed as part of this scope.

The following files are in scope of this review:

```
src/
    ALMProxy.sol
    ALMProxyFreezable.sol
    AccessControls.sol
    Beacon.sol
    Controller.sol
    ControllerSharedStorage.sol
    PAUFactory.sol
    RateLimits.sol
    facets/
        Facet.sol
        IFacet.sol
        aave/
            AaveFacet.sol
            IAaveFacet.sol
        basin/
            BasinFacet.sol
            IBasinFacet.sol
        cctp/
            CCTPFacet.sol
            ICCTPFacet.sol
        centrifuge/
            CentrifugeFacet.sol
            ICentrifugeFacet.sol
        curve/
            CurveFacet.sol
            ICurveFacet.sol
        dai-usds/
            DAIUSDSFacet.sol
            IDAIUSDSFacet.sol
        erc4626/
            ERC4626Facet.sol
```

```
        IERC4626Facet.sol
    erc7540/
        ERC7540Facet.sol
        IERC7540Facet.sol
    ethena/
        EthenaFacet.sol
        IEthenaFacet.sol
    farm/
        FarmFacet.sol
        IFarmFacet.sol
    layer-zero/
        LayerZeroFacet.sol
        ILayerZeroFacet.sol
    maple/
        MapleFacet.sol
        IMapleFacet.sol
    merkl/
        MerklFacet.sol
        IMerklFacet.sol
    otc/
        OTCFacet.sol
        IOTCFacet.sol
        OTCBuffer.sol
        IOTCBuffer.sol
    pendle/
        PendleFacet.sol
        IPendleFacet.sol
    psm/
        PSMFacet.sol
        IPSMFacet.sol
    psm3/
        PSM3Facet.sol
        IPSM3Facet.sol
    spark-vault/
        SparkVaultFacet.sol
        ISparkVaultFacet.sol
    superstate/
        SuperstateFacet.sol
        ISuperstateFacet.sol
    transfer-asset/
        TransferAssetFacet.sol
        ITransferAssetFacet.sol
    uniswap-v3/
        UniswapV3Facet.sol
        IUniswapV3Facet.sol
        UniswapV3Utils.sol
    uniswap-v4/
        UniswapV4Facet.sol
        IUniswapV4Facet.sol
    usds/
        USDSFacet.sol
        IUSDSFacet.sol
    weeth/
        WEETHFacet.sol
        IWEETHFacet.sol
```

```
            WEETHModule.sol
            IWEETHModule.sol
        wrap-proxy-eth/
            WrapProxyETHFacet.sol
            IWrapProxyETHFacet.sol
        wsteth/
            WSTETHFacet.sol
            IWSTETHFacet.sol
    interfaces/
        IALMProxy.sol
        IALMProxyFreezable.sol
        IAccessControls.sol
        IBeacon.sol
        IController.sol
        IEnumerableIntegrations.sol
        IPAUFactory.sol
        IRateLimits.sol
    libraries/
        ApproveLib.sol
        RateLimitHelpers.sol
```

The assessment was performed on the source code files inside the Diamond PAU v1.13 repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	04 Mar 2026	6b0958f575d6fc99311df5326353f3202fba95b5	Initial Version
2	11 Mar 2026	220dbcfd6ebbf9e23f684aa9c29c8d76e14d447c	After Intermediate Report
3	13 Mar 2026	697a8dbb2cd5b46ae7b5c618e1d9c7efb112a83e	v1.11.0
4	06 Apr 2026	fea256adfc90d162c33917099985a69427a1129d	v1.12.0-beta.0
5	27 Apr 2026	dc54647a8fb8cd9fe313e95d149cb9c65d36e2c6	v1.12.0-rc.0
6	06 May 2026	fc2fe7fe5991b4f84f39ec88239b5ae4ffaacc8f	v1.12.0-rc.1
7	15 May 2026	ebcc56d45d1835d74915c2446ecc71cb61589c5f	v1.12-rc.2
8	27 May 2026	a84fe9616412e3b6d10ea64301a3481f7dcf9a05	v1.12.0
9	02 Jun 2026	fd5f09c5254ac7f1931a85e81a8d9036bd257704	v1.13.0-beta.0
10	04 Jun 2026	5c5ad6ae174bf467081ca82342ced2bd42a5c732	v1.13.0

For the solidity smart contracts, the compiler version `0.8.34` was chosen.

2.1.1 Excluded from scope

Any other files are out of scope.

The following are explicitly out of scope:

- Most integrations have been reviewed in separate reports (see Spark ALM Controller v1.7.0 Review, v1.10.0 Diff Review and Grove ALM Controller Review v1.8.0).
- Uniswap V4, WSTETH and OTCSwap integrations are assumed to be correct and secure.
- All 3rd-party protocols that the ALMProxy integrates with are out of scope. They are assumed to work honestly and correctly as documented.
- Dependencies and libraries e.g. OpenZeppelin.

2.2 System Overview

This system overview describes the latest received version (Version 9) of the contracts as defined in the Assessment Overview.

In the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Sky offers Diamond PAU (Parallelized Allocation Units), a modular architecture using the diamond proxy pattern. It replaces the legacy ALM Controller with a faceted approach where integrations are deployed as separate facet contracts that can be shared across several Liquidity Layers and chains. At the center of each deployment sits a single `Controller`, which serves as the entry point for allocator operations and routes each call to the facet its admin has wired up from the shared `Beacon` registry. Facets can be added, replaced, or removed over the lifetime of a deployment as its admin syncs updates from the `Beacon`.

2.2.1 Core Contracts

The PAU system is deployed per chain, and each deployment consists of its own dedicated set of the following core contracts: `ALMProxy`, `Controller`, `AccessControls`, and `RateLimits`. The `Beacon` is the single contract that may be shared across multiple deployments (it acts as the canonical registry of integration configurations). The `PAUFactory` is simply a convenience deployer for the individual core contracts.

While the typical deployment pairs a single `Controller` with a single `ALMProxy`, the architecture also supports backing one `ALMProxy` with several `Controller` contracts. In that topology each `Controller` can carry its own `AccessControls` and its own synced integration set, and either a dedicated `RateLimits` contract or one shared across the controllers, so governance can give different allocator groups intentionally different capabilities over the same custody account.

ALMProxy: Holds the deployment's funds and interacts with external protocols on its behalf. It exposes `doCall()`, `doCallWithValue()`, and `doDelegateCall()`, all restricted to the `CONTROLLER` role. The `CONTROLLER` role is granted to this deployment's `Controller` contract by the *admin* after deployment and can only be managed by the *admin*.

Controller: The single entry point for all allocator operations of a deployment. Inspired by EIP-2535, it uses dispatch-based routing via its `fallback()`: the incoming `msg.sig` (the `callSelector`) is looked up in a local `dispatches` mapping to resolve `(facet, delegateSelector)`, and the call is forwarded to `facet` via `delegatecall` with the first 4 bytes of calldata rewritten to `delegateSelector`. Integration configurations are synced from the `Beacon` via `updateIntegrations()` and can be removed via `removeIntegrations()`; both are admin-only and reentrancy-guarded. Shared state (`accessControls`, `proxy`, `rateLimits`, all pointing at this deployment's own contracts) is exposed to all facets through `ControllerSharedStorage`. Each facet

additionally uses its own ERC-7201 namespaced storage domain (tagged `v1`) to prevent collisions. The `beacon` reference is immutable on the Controller.

Beacon: Canonical repository of integration configurations shared by one or more Controllers. The *admin* (Sky governance) registers a facet together with a list of `(callSelector, delegateSelector)` wires via `setIntegration()` and removes an integration via `removeIntegration()`. Uniqueness is enforced on `callSelector` only. Two facets can share the same `delegateSelector` (the internal facet function) as long as they are exposed under different `callSelectors`. `callSelector` values that collide with hardcoded Controller selectors (`updateIntegrations`, `removeIntegrations`, `accessControls`, `beacon`, `proxy`, `rateLimits`, and the `IEnumerableIntegrations` view selectors) are rejected at registration. Controllers sync their local view from the Beacon, which allows them to opt into upgrades independently.

PAUFactory: Convenience factory for deploying the individual core contracts with the expected bytecode. Rather than a single atomic system deploy, it exposes one deployer per component: `deployALMProxy()`, `deployALMProxyFreezable()`, `deployRateLimits()`, `deployAccessControls()` (each constructing the contract with the caller-specified `admin` as `DEFAULT_ADMIN_ROLE`), and `deployController()` (taking the addresses of an already-deployed `AccessControls`, `ALMProxy`, and `RateLimits`, and wiring the immutable `Beacon` the factory was constructed with). The role wiring must be performed by the *admin* after deployment. The `Beacon` referenced by the factory is immutable.

AccessControls: Each Controller deployment has its own dedicated `AccessControls` contract (referenced via `ControllerSharedStorage.accessControls`), which holds the runtime role database consulted by that Controller's facets. It extends OpenZeppelin's `AccessControlEnumerable` and exposes a `setRoleAdmin(role, adminRole)` function, callable only by `DEFAULT_ADMIN_ROLE`, which records the role that acts as admin of a given role. Under `AccessControlEnumerable` semantics, the admin role of a given role can both grant and revoke that role. Facets do not implement access control themselves; role checks performed in a facet are delegated to the linked `AccessControls` contract. Roles such as `ALLOCATOR_ROLE` and `FREEZER_ROLE` are referenced by the facets and `ALMProxyFreezable`; which role can grant and revoke each of them (e.g. allowing `FREEZER_ROLE` to both grant and revoke `ALLOCATOR_ROLE`) is configured by the admin via `setRoleAdmin()`.

RateLimits: Each Controller deployment has its own dedicated `RateLimits` contract (referenced via `ControllerSharedStorage.rateLimits`), storing per-key rate limit configurations for that Controller's facets. Each key is a `keccak256` hash, typically derived as `PREFIX + encoded args` (e.g. `(token, pool)`). `setRateLimitData()` (admin-only) sets `maxAmount`, `slope`, `lastAmount`, and `lastUpdated`; convenience variants set an initially-full or unlimited limit. Only a `Controller` granted the `CONTROLLER` role by the admin can `triggerRateLimitDecrease()` / `triggerRateLimitIncrease()` (a single `RateLimits` contract may be shared by more than one such `Controller`). The current limit is computed as `min(lastAmount + slope * elapsed, maxAmount)`; a `maxAmount == type(uint256).max` encodes an unlimited limit.

2.2.2 ALMProxyFreezable

`ALMProxyFreezable` is a standalone, lighter-weight alternative to the full PAU stack, intended for low-stakes deployments where funds or critical authority are not held and an intermediary Controller is unnecessary. Each `ALMProxyFreezable` is its own self-contained deployment: there is no Controller, no facets, and no associated `RateLimits` or `AccessControls`. Allocators hold the `ALLOCATOR_ROLE` directly on the contract and invoke `doCall()` / `doCallWithValue()` without going through a Controller. A holder of the `FREEZER_ROLE` can revoke `ALLOCATOR_ROLE` via `removeAllocator()`. This contract is deployed independently.

2.2.3 Facets

A facet is a self-contained contract implementing one integration with an external protocol (Aave, Curve, CCTP, etc.). Each facet is deployed once and registered in the `Beacon`, and any number of Controllers can sync it into their dispatch tables. When a call reaches a Controller, it is forwarded to the matching facet, which runs in that Controller's context and routes any external protocol interactions through the deployment's `ALMProxy`.

A facet typically exposes three kinds of functions:

- *Allocator* functions that trigger liquidity allocations (swaps, deposits, withdrawals, bridging, ...).
- *Admin* functions that configure the facet for a given deployment (slippage bounds, tick ranges, recipients, whitelists, ...).
- View functions that expose the facet's configuration and rate-limit-key derivations.

Allocator and admin functions are gated by an `onlyRole` check that queries the `AccessControls` contract linked to the Controller in which the facet is executing, which holds the role database for that deployment. Per-deployment configuration written by admin functions lives in the facet's own ERC-7201 namespaced storage, so the same facet code can be reused across deployments while being tuned independently on each chain.

Facet operations are not rate-limited by default; whether a given operation consumes or requires a rate limit is implementation-specific. When used, rate-limit keys are constructed from a facet-specific `LIMIT_*` prefix combined with relevant identifiers (token, pool, destination, ...). Common patterns include reversible operations whose reverse direction replenishes the forward limit, and operations (e.g. cancels or claims) that do not consume any limit but require a corresponding limit to exist, using limit existence as a whitelist.

For AMM-style integrations (Curve, Uniswap V3, Uniswap V4) the rate-limit structure was standardized: each pool has an *aggregate* deposit and withdraw limit denominated in a normalized value across all assets, plus a *per-asset* deposit and swap limit. Liquidity operations consume the aggregate limit; swaps consume the per-asset swap limit; imbalanced add-liquidity operations also consume the per-asset deposit limit for the swapped portion.

The *admin* of a given deployment is responsible for wiring only those facets relevant to the chain it deploys to. The available facets are described below.

Arbitrary Token Transfer:

`transfer()`: transfers ERC-20 assets out manually, respecting the rate limit per `(asset, destination)` pair. Intended for integration with systems where direct token transfers are needed (e.g. Spark Vaults V2 return flow).

USDS (DSS Allocator):

`mint()` / `burn()` leverage the Sky Allocator vault to `draw()` or `wipe()` USDS. Each direction has its own rate limit (`LIMIT_USDS_MINT` and `LIMIT_USDS_BURN`); `mint()` decreases `LIMIT_USDS_MINT` and `burn()` decreases `LIMIT_USDS_BURN`. A burn additionally attempts to replenish `LIMIT_USDS_MINT` (best-effort: the increase is skipped if the mint limit is not configured). The vault address is admin-configurable via `setVault()`.

DSS LitePSM:

`swapUSDCToUSDS()` / `swapUSDSToUSDC()` leverage the DSS LitePSM to swap between USDC and USDS without fees (`sellGemNoFee()` / `buyGemNoFee()`). Each direction has its own rate limit (`LIMIT_USDS_TO_USDC` and `LIMIT_USDC_TO_USDS`); `swapUSDSToUSDC()` decreases `LIMIT_USDS_TO_USDC` and `swapUSDCToUSDS()` decreases `LIMIT_USDC_TO_USDS`. `swapUSDCToUSDS()` additionally attempts to replenish `LIMIT_USDS_TO_USDC` (best-effort: the increase is skipped if the limit is not configured). `swapUSDCToUSDS()` triggers PSM `fill()` when needed and chunks the swap if the buffered DAI is insufficient.

DAI/USDS Converter:

`swapDAIToUSDS()` / `swapUSDSToDAI()` support converting DAI to USDS and vice versa 1:1 via the DAI/USDS converter. Each direction has its own rate limit (`LIMIT_DAI_TO_USDS` and `LIMIT_USDS_TO_DAI`); the two are independent and the reverse direction does not replenish the forward limit. The facet has no admin configuration.

Spark PSM3:

`deposit()` / `withdraw()` enable depositing / withdrawing specific assets to / from a Spark PSM3. Each direction has its own per-asset rate limit (`LIMIT_PSM_DEPOSIT` and `LIMIT_PSM_WITHDRAW`); the two are independent and the reverse direction does not replenish the forward limit.

Circle CCTP:

1. `setDomainParameters()`: restricted to the *admin*; configures the mint recipient and the allowed fee cap rate band (`minFeeCapRate`, `maxFeeCapRate`, both expressed in basis points of the burn amount) for a destination domain when bridging USDC with CCTP (Circle's Cross-Chain Transfer Protocol). Recipient must be non-zero and `minFeeCapRate <= maxFeeCapRate <= 10_000`.
2. `transfer()`: leverages CCTP v2 to bridge USDC to a recipient (expected to be another ALMProxy) on a foreign domain. The *allocator* supplies a `feeCapRate` (basis points), validated to lie within the destination domain's configured `[minFeeCapRate, maxFeeCapRate]` band; for each chunked burn the fee passed to CCTP is bounded by `transferAmount * feeCapRate / 10_000`. The call is rate-limited per destination domain (`LIMIT_USDC_TO_DOMAIN`) and globally across all CCTP transfers (`LIMIT_USDC_TO_CCTP`).

Large transfers are automatically split into chunks bounded by the CCTP minter's `burnLimitsPerMessage`. The `DESTINATION_CALLER` is always `bytes32(0)`, allowing any allocator to finalize the message on the destination chain; the finality threshold is set to `MAX_FINALITY_THRESHOLD = 2_000` (standard / finalized).

ERC-4626:

`deposit()` / `withdraw()` / `redeem()` enable integrations with ERC-4626 vaults. Deposits and withdrawals / redemptions are rate-limited per vault on independent `LIMIT_4626_DEPOSIT` / `LIMIT_4626_WITHDRAW` keys; withdrawals and redemptions replenish the deposit limit, capped at `maxAmount`. The *admin* can set maximum exchange rate thresholds via `setMaxExchangeRate()`, which stores a `1e36 * assets / shares` bound. During deposit, the actual exchange rate is validated against the configured maximum, protecting against exchange rate manipulation. All three functions also accept slippage-protection parameters: `minSharesOut` for deposits, `maxSharesIn` for withdrawals, and `minAssetsOut` for redemptions. Balance deltas are computed explicitly rather than relying on returned values.

Aave:

`deposit()` / `withdraw()` enable integrations with Aave V3 pools. Deposits are rate-limited per `(underlyingAsset, pool, aToken)` on `LIMIT_AAVE_DEPOSIT`; withdrawals are rate-limited per `(pool, aToken)` on `LIMIT_AAVE_WITHDRAW`. Withdrawals additionally replenish the deposit limit (capped at `maxAmount`). The *admin* must configure a `maxSlippage` per `aToken` via `setMaxSlippage()`. After supplying assets to the Aave pool, the function verifies that the `aToken` balance delta is at least `amount * maxSlippage / 1e18`, protecting against unfavorable exchange rates.

Ethena (USDe / sUSDe):

1. `setDelegatedSigner()` / `removeDelegatedSigner()`: configure delegated signers on the Ethena minting contract. Both functions are gated only by existence of the corresponding `LIMIT_ETHENA_SET_DELEGATED_SIGNER` / `LIMIT_ETHENA_REMOVE_DELEGATED_SIGNER` keys (existence acts as a feature toggle; no amount is consumed). Assigned delegated signers must explicitly accept the delegation with `confirmDelegatedSigner()` on Ethena before they can sign `Orders` on behalf of the ALMProxy. An `Order` is an expirable intent to mint or redeem USDe with a fixed spending collateral and outcome. Minter and burner roles on the Ethena minting contract are required to submit the order transaction with the delegated-signer signature.

2. `prepareMint()` / `prepareBurn()`: approve USDC / USDe to the Ethena minting contract, enabling a subsequent off-chain mint or burn flow. Rate-limited by `LIMIT_ETHENA_MINT` / `LIMIT_ETHENA_BURN`.
3. Once USDe is held, it can be staked into sUSDe using the generic ERC-4626 `deposit()`.
4. `cooldownAssets()` / `cooldownShares()`: initiate a withdrawal of sUSDe that requires a cooldown followed by an `unstake()` call, since instant withdrawal of sUSDe is currently blocked. These functions burn the shares and credit the USDe to the USDeSilo contract. Rate-limited by `LIMIT_ETHENA_COOLDOWN`.
5. `unstake()`: exits USDe to the ALMProxy once the `cooldownDuration` has passed. Gated only by existence of `LIMIT_ETHENA_UNSTAKE`.

Mint, burn, and cooldown are separately rate-limited. Actual mint/burn on Ethena is completed off-chain by delegated signers; this facet only manages approvals and signer delegation.

Superstate Minting:

`subscribe()`: allows minting USTB with USDC. Subject to a global rate limit (`LIMIT_SUPERSTATE_SUBSCRIBE`). USDC approvals are granted before calling the Superstate function. Redemptions are not supported, but USTB can be transferred out using the `transferAsset` facet if necessary.

Maple:

Maple deposits use the generic ERC-4626 deposit function. This facet handles the (non-ERC-4626) redemption flow:

1. `requestRedemption()`: creates a redemption request; rate-limited per maple token on the `convertToAssets(shares)` value (`LIMIT_MAPLE_REQUEST_REDEEM`).
2. `cancelRedemption()`: cancels an unfinalized redemption request; only requires that the cancel rate limit (`LIMIT_MAPLE_CANCEL_REDEEM`) is configured but does not consume it.

It is assumed that manual withdrawal is disabled for the ALMProxy on Maple, so that processed redemptions are pushed back to the ALMProxy automatically. If Maple is configured for manual withdrawal, *allocators* must trigger an explicit ERC-4626 `redeem()` on the ALMProxy to retrieve the tokens, which further requires the ERC-4626 withdrawal limit being configured for the maple pool.

Centrifuge:

The Centrifuge deposit/redeem request lifecycle itself (`requestDeposit`, `claimDeposit`, `requestRedeem`, `claimRedeem`) is handled by the generic ERC-7540 facet. The Centrifuge facet covers the Centrifuge-specific surface:

1. `cancelDepositRequest()` / `cancelRedeemRequest()`: cancel a pending deposit / redeem request.
2. `claimCancelDepositRequest()` / `claimCancelRedeemRequest()`: claim the assets / shares returned from a cancelled request once it is fulfilled. Balance deltas are computed explicitly and emitted in the events.
3. `transferShares()`: transfers share class tokens to a cross-chain recipient via a Centrifuge spoke. The recipient per destination (`centrifugeId`) must be explicitly configured by the *admin* via `setRecipient()`. The function is payable so that messaging fees can be forwarded. The rate-limit key is composed from `(token, centrifugeId, spoke)`, where `spoke` is fetched at call time from the vault's manager.

Cancel and claim operations only require that the corresponding `LIMIT_CENTRIFUGE_*` keys exist; they do not consume any rate limit. Cross-chain share transfers consume `LIMIT_CENTRIFUGE_TRANSFER`.

ERC-7540 (for Centrifuge):

`requestDeposit()` / `claimDeposit()` / `requestRedeem()` / `claimRedeem()` enable integrations with ERC-7540 async vaults (currently Centrifuge). `requestDeposit()` transfers assets into escrow and is rate-limited per `(asset, token)` on `LIMIT_7540_REQUEST_DEPOSIT`; `requestRedeem()` transfers shares into escrow and is rate-limited per token on `LIMIT_7540_REQUEST_REDEEM` denominated in `convertToAssets(shares)`. `claimDeposit()` / `claimRedeem()` finalize the request and retrieve the maximum claimable shares / assets respectively; they only require that the corresponding `LIMIT_7540_CLAIM_*` keys exist but do not consume them.

Curve StableswapNG:

`swap()` / `addLiquidity()` / `removeLiquidity()` enable swaps and liquidity management on Curve StableswapNG pools. Rate limits follow the standard AMM pattern: an aggregate deposit limit per pool, an aggregate withdraw limit per pool, and per-asset swap and deposit limits. Liquidity values are denominated in the aggregated underlying value of the operation (`Σ amount_i × stored_rates()[i] / 1e18`). `addLiquidity()` decrements the aggregate deposit limit by the aggregated value and additionally decrements the per-asset deposit limit for assets that were net-deposited and the per-asset swap limit for any portion that was effectively swapped due to imbalanced deposits. The *admin* configures per-pool slippage via `setMaxSlippage()`; swap slippage is bounded against `stored_rates()`-derived equivalent amounts, whereas `addLiquidity()` / `removeLiquidity()` bounds use `get_virtual_price()` to convert between LP tokens and underlying value.

Grove Basin:

`deposit()` / `withdraw()` enable integrations with Grove Basin liquidity pools. Both directions are rate-limited per `(basin, asset)` (`LIMIT_BASIN_DEPOSIT` / `LIMIT_BASIN_WITHDRAW`), with no replenishment. `deposit()` enforces an allocator-supplied `minSharesOut` against the share balance delta. `withdraw()` enforces `assetsWithdrawn * 1e18 >= sharesBurned * minConversionRate` against the actual amounts the basin returned.

LayerZero:

1. `setRecipient()`: restricted to the *admin*; sets the `LayerZeroRecipient` for a given `destinationEndpointId`. Recipient must be non-zero.
2. `transfer()`: triggers a `send()` on an OFT to bridge the OFT supported token to a `destinationEndpointId`. Fees are estimated via `quoteSend()` and attached; the function is payable, and the *allocator* pays the bridging fee. `minAmountLD` is set to `amount` rounded down to the granularity returned by the OFT's `decimalConversionRate()`, so the call tolerates only the rounding inherent in the OFT's decimal conversion. The OFT's `send()` is invoked with the proxy as the refund address, and any native value left in the controller after the call is swept to the proxy. The call is rate limited per `(token, oft, peer, destinationEndpointId)`.

Farm:

1. `deposit()` / `withdraw()`: deposit and withdraw a staking token to / from a SPK-compatible farm. Each direction is rate-limited per farm (`LIMIT_FARM_DEPOSIT` keyed by `(stakingToken, farm)`; `LIMIT_FARM_WITHDRAW` keyed by `(farm)`).
2. `claimReward()`: standalone reward claim. Gated only by existence of `LIMIT_FARM_CLAIM_REWARD` (no amount consumed).

Spark Vaults V2:

`take()` enables taking assets from a `SparkVault`. Rate-limited per SparkVault (`LIMIT_SPARK_VAULT_TAKE`). Funds can be returned using the `transferAsset` facet.

Pendle:

`redeem()`: allows the *allocator* to redeem matured Principal Tokens from expired Pendle markets. The market must have reached expiry. The redemption is rate-limited per `(pt, market)` on `LIMIT_PENDLE_PT_REDEEM` and decremented by the actual `tokenOutAmount` measured from the

proxy's balance delta. The minimum expected output passed to Pendle is computed internally from the yield token's `pyIndexCurrent()` with a 5-wei buffer to absorb rounding; an allocator-specified `minAmountOut` provides additional slippage protection against the measured balance delta.

Uniswap V3:

1. `swap()`: allows the *allocator* to execute token swaps through whitelisted Uniswap V3 pools. Rate-limited using a key derived from `(tokenIn, pool)`. A TWAP oracle is used to compute the price limit tick; admin-configured `maxTickDelta` bounds allowed price movement, and allocator-supplied `minAmountOut` bounds slippage.
2. `addLiquidity()`: mints a new position or adds to an existing position. New positions must fall within admin-configured tick bounds and their tick range must be aligned to the pool's tick spacing. For existing positions, the provided tick range must equal the position's stored tick range and the position must be owned by the ALMProxy. Deposit amounts are rate-limited under the standard AMM aggregate-and-per-asset pattern, and TWAP-based minimum-amount checks are applied using `maxSlippage`.
3. `removeLiquidity()`: decreases liquidity from existing positions and collects the resulting tokens (including accumulated fees) to the ALMProxy. Withdrawn amounts decrement the per-pool aggregate withdraw limit and the per-asset withdraw limit for each withdrawn asset. Slippage is checked against the collected token balance deltas. `maxSlippage` must be configured (non-zero) for the pool.

The *admin* configures pool parameters using `setMaxSlippage()`, `setMaxTickDelta()`, `setLiquidityLowerTickBound()`, `setLiquidityUpperTickBound()`, and `setTWAPSecondsAgo()`. The router and position manager are immutable facet parameters. Configuring these parameters effectively whitelists a pool for *allocator* use.

Only pools whose assets are intended to trade 1:1 (e.g. stablecoin pairs) are supported.

Merkl:

`toggleOperator()`: allows the *allocator* to toggle an operator address on the Merkl distributor, enabling or disabling that operator's ability to claim rewards on behalf of the ALMProxy. Gated only by existence of `LIMIT_MERKL_TOGGLE_OPERATOR` keyed by `(operator, distributor)`. The operator can only trigger claims; funds are always sent to the ALMProxy. The distributor is supplied by the allocator and included in the rate-limit key.

WrapProxyETH:

`wrapAll()`: wraps the ALMProxy's entire native ETH balance into WETH. No admin configuration; gated only by existence of the `LIMIT_WRAP_PROXY_ETH` rate-limit key.

EtherFi (weETH):

This facet enables the ALMProxy to deposit into and withdraw from EtherFi's liquidity pool. Withdrawal requests are represented as NFT receipts, which a companion `WEETHModule` contract receives and settles on behalf of the ALMProxy.

1. `deposit()`: converts WETH to weETH. The function unwraps WETH, deposits ETH into the EtherFi liquidity pool to receive eETH shares, and wraps the eETH into weETH. A `minSharesOut` parameter provides slippage protection. Rate-limited per `(eeth, liquidityPool)` on `LIMIT_WEETH_DEPOSIT`.
2. `requestWithdraw()`: initiates a withdrawal by unwrapping weETH to eETH and requesting a withdrawal from the EtherFi liquidity pool. The resulting NFT receipt is minted to the `WEETHModule` contract. A `minEETHShares` parameter is enforced during weETH unwrapping. Rate-limited per `(eeth, liquidityPool, weethModule)` on `LIMIT_WEETH_REQUEST_WITHDRAW`; the limit's existence acts as the module whitelist.
3. `claimWithdrawal()`: delegates to the `WEETHModule` to claim a finalized withdrawal; only requires that `LIMIT_WEETH_CLAIM_WITHDRAW` keyed by `(weethModule)` exists but does not consume it.

The `WEETHModule` is a UUPS-upgradeable contract that implements `onERC721Received` to hold EtherFi withdrawal request NFTs, and exposes a `claimWithdrawal()` that claims ETH from finalized requests, wraps it to WETH, and transfers it to the ALMProxy. Only the ALMProxy can call `claimWithdrawal()` on it. Only `DEFAULT_ADMIN_ROLE` can upgrade the implementation.

The following integrations are assumed to be correct and secure since they are not audited by ChainSecurity (only diffs reviewed):

Uniswap V4:

1. `swap()`: swaps an input token to an output token using a Uniswap V4 pool. `maxSlippage` is used for slippage protection across normalized token amounts.
2. `mintPosition()`: mints a new position in a Uniswap V4 pool. The tick range must fall within the admin-defined tick limits.
3. `increasePosition()`: adds liquidity to an existing position (owned by the ALMProxy), respecting the admin-defined tick limits.
4. `decreasePosition()`: removes liquidity from an existing position.

Swaps are rate-limited per `(poolId, tokenIn)` in token-native units. Liquidity operations use the standard AMM aggregate-and-per-asset pattern: aggregate deposit and withdrawal limits are keyed by `poolId` and denominated in 18-decimal normalized units, while per-asset deposit and withdraw limits are keyed by `(poolId, token)` and use token-native units. The *admin* configures tick limits and slippage using `setTickLimits()` and `setMaxSlippage()`. The facet uses Permit2 for approvals, the Universal Router for swaps, and the PositionManager for liquidity operations.

WSTETH:

1. `deposit()`: deposits WETH to Lido and receives wstETH; rate-limited globally (`LIMIT_WSTETH_DEPOSIT`).
2. `requestWithdraw()`: requests a withdrawal from wstETH; rate-limited by equivalent stETH amount (`LIMIT_WSTETH_REQUEST_WITHDRAW`).
3. `claimWithdrawal()`: claims a finalized withdrawal request, wraps the returned ETH to WETH, and transfers it to the ALMProxy. Requires `LIMIT_WSTETH_CLAIM_WITHDRAW` to exist but does not consume it.

OTC Swap:

The `OTCBuffer` is a UUPS-upgradeable AccessControl contract that holds counter-assets during OTC swaps. Its *admin* configures per-asset approvals to the ALMProxy via `approve()` on the buffer, so that the ALMProxy can pull claimed assets back. The buffer has no allocator-facing surface.

The facet itself exposes:

1. `setBuffer()`, `setMaxSlippage()`, `setRechargeRate()`: *admin*-only configuration of an OTC exchange (the counter-buffer, max slippage per exchange, and recharge rate). `setBuffer()` is blocked while a swap is pending and not yet ready.
2. `send()`: starts an OTC swap by transferring the input asset directly to the exchange when the swap is ready. Rate-limited per `(asset, exchange)` on `LIMIT_OTC_SEND`, denominated in the asset's native units; configuring the limit acts as the per-asset whitelist for that exchange.
3. `claim()`: claims the full buffer balance of the output asset back to the ALMProxy via `transferFrom`. Gated only by existence of `LIMIT_OTC_CLAIM` keyed by `(asset, exchange)`.
4. `isSwapReady()`: returns whether the sum of already-claimed amount plus `elapsed * rechargeRate` exceeds `sent * maxSlippage / 1e18`.

2.3 Trust Model

2.3.1 Core Infrastructure

Beacon: The Beacon is a shared entity used by all Controllers. The *admin* (Sky governance) is fully trusted, otherwise they can register arbitrary facets that, once synced by a Controller via `updateIntegrations()`, are reachable via `delegatecall` from the Controller and can fully compromise the ALMProxy. Each Controller independently decides when to sync, so a Beacon compromise only affects a Controller after it performs a sync.

ALMProxy: The *admin* (Star's proxy) is fully trusted, otherwise they can set up controllers and trigger any calls with the privilege of the ALMProxy. The *controller* is also trusted. In addition, the ALMProxy requires several governance-configured roles to operate (e.g. `bud` on DSS LitePSM for fee-less swaps, `wards` on AllocatorVault to `draw()`/`wipe()` USDS, and sufficient AllocatorBuffer allowance to move minted USDS).

Controller: The *admin* (Star's proxy) is fully trusted, otherwise they can configure facet storage maliciously and steal funds. The *allocator* is semi-trusted and can cause limited damage, dependent on the facets (see below). The *freezer* is also semi-trusted and can temporarily DoS the controller in the worst case, but is expected to pause malicious allocators. The freezer-allocator relationship is enforced by the AccessControls contract once the admin has configured `FREEZER_ROLE` as the revoker for `ALLOCATOR_ROLE` via `setRoleRevoker()`.

AccessControls: The admin (Star's proxy, holder of `DEFAULT_ADMIN_ROLE`) is fully trusted. It can grant or revoke any role (including `ALLOCATOR_ROLE` and `FREEZER_ROLE`) and, via `setRoleAdmin(role, adminRole)`, can always set which role acts as admin of another. AccessControls inherits OpenZeppelin's AccessControlEnumerable, so the admin of a role can both grant and revoke this role for addresses.

RateLimits: The *admin* (Star's proxy) is fully trusted to configure the limit data and *controller* correctly.

ALMProxyFreezable: Inherits the same role-based trust model. The *admin* (Star's proxy) is fully trusted; the *allocator* is semi-trusted and can execute arbitrary calls through the proxy (there is no intermediary Controller, hence no rate-limit or facet-level slippage enforcement); the *freezer* is semi-trusted and can DoS by removing allocators. This variant is not intended to hold funds or critical authority.

PAUFactory: The `deploy*` functions are permissionless, so anyone can invoke the factory to instantiate the individual core contracts. A contract produced by the factory is **not** trusted by virtue of the factory itself; only deployments performed by the officially designated actors (e.g. Sky/Star governance) are to be trusted. Each deployment's trust model is determined by the `admin` passed to the respective deployer and by the `Beacon` the factory was constructed with. Note that governance is additionally responsible for wiring the contracts.

Before initializing the contracts, the governance should always carefully examine whether the deployed contracts match the expectations.

2.3.2 Integrations

Integrated contracts and external systems are partially trusted:

- Rate limits should limit damage within governance-set bounds (e.g. in case of exploits in external systems) and are the root of the trust model.
- Other mechanisms exist additionally (e.g. governance-defined maximum slippage). These typically restrict unreasonable token outflows that an allocator could trigger (e.g. in swaps), and also protect against malicious integrations (e.g. ERC-4626 deposits). These mechanisms are integration-specific.

Trust in the integration targets varies per contract; we outline each below. Note that underlying tokens are trusted.

Sky ecosystem contracts: Trusted to behave as expected and not to be maliciously configured. Rate limits and other mechanisms nonetheless apply (mostly) to restrict the allocator, preventing unexpectedly large liquidity flows that would introduce economic risk. The following facets fall in this category:

- `DAIUSDSFacet`
- `PSMFacet`
- `PSM3Facet`
- `FarmFacet`
- `SparkVaultFacet`
- `USDSFacet`
- `OTCFacet` / `OTCBuffer`
- `BasinFacet`

Trusted integrations: Treated similarly to Sky ecosystem contracts due to immutability: external, but assumed immutable or not subject to malicious updates. For forks and similar, governance should take special care to ensure immutability:

- `CurveFacet`
- `PendleFacet`
- `UniswapV3Facet`
- `UniswapV4Facet`
- `TransferAssetFacet`
- `MerklFacet`

Untrusted integrations: Upgradeable or otherwise mutable contracts whose governance may act maliciously and attempt to bypass rate limits or pull unexpected assets. Rate limits and other restrictions still apply, but under a wider threat model.

- `CCTPFacet`
- `SuperstateFacet`
- `EthenaFacet`
- `WSTETHFacet`
- `WrapProxyETHFacet`
- `AaveFacet`
- `CentrifugeFacet`
- `ERC4626Facet`
- `ERC7540Facet`
- `MapleFacet`
- `LayerZeroFacet`
- `WEETHFacet` / `WEETHModule`

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

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

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Open Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	1

- Permissionless Asset Inflows Inflate Rate Limits Risk Accepted

Low-Severity Findings	7

- Reentrancy Guard Is per-Controller While the ALMProxy Is Shared Risk Accepted
- Centrifuge Deposit / Redemption Can Be DoSed by Cancellation Risk Accepted
- Centrifuge Tranche Token Price May Change Between Request Submission and Execution Risk Accepted
- Maple Redemption Can Be DoSed Risk Accepted
- Over-reduced Limit in Maple Redemption Risk Accepted
- Relayer Can DoS SUSDE Unstaking Risk Accepted
- Revoking Unused Approval Acknowledged

5.1 Permissionless Asset Inflows Inflate Rate Limits

Security Medium Version 7 Risk Accepted

CS-SKYDPAU-045

`ERC4626Facet.withdraw()`, `ERC4626Facet.redeem()` and `AaveFacet.withdraw()` increase deposit rate limits by the balance delta created on the corresponding withdrawal:

```
uint256 startingBalance = IERC20Like(asset).balanceOf(proxy);
IALMProxy(proxy).doCall(target, /* withdraw / redeem call */);
uint256 received = IERC20Like(asset).balanceOf(proxy) - startingBalance;
_decreaseRateLimit(withdrawKey, received);
_tryIncreaseRateLimit(depositKey, received);
```

Any increase of the proxy's measured-token balance between the two `balanceOf` snapshots is attributed to the integration withdrawal, regardless of where it originated. Since both reads happen within the same call, an unrelated inflow only inflates `received` if it lands during the `doCall`. For example, a malicious integration that claims the proxy's permissionlessly-claimable funds mid-withdrawal.

Below, an example with LayerZero v2 (any external funds source with permissionless fund claiming on behalf of ALM Proxy is suitable for the example):

1. Allocator bridges 10M (`1e24`) USDS through the OFT back to Ethereum Mainnet.
2. Allocator calls `ERC4626Facet.withdraw(vault, 1)` to withdraw 1 wei from a USDS vault. `vault.withdraw(1)` is invoked accordingly, which:
    1. Transfers 1 wei USDS to the proxy.
    2. Executes `endpoint.lzReceive` to finalize the bridging of funds. 10M USDS are received.
3. The asset delta is ~10M USDS (`1e24+1`), even though only 1 wei comes from `vault`. The vault's rate limit is maliciously increased.
4. Allocator deposits into the vault up to the inflated allowance, which the vault then takes.

The configured max amount only caps a single attack. It can be circumvented by bridging repeatedly, refilling the limit on demand and repeating the attack quickly.

In summary, any of the ALMProxy's funds held in external protocols that are permissionlessly claimable can be claimed by a malicious integration to inflate the rate limit by an unreasonable amount. Through repetition, a given token can be drained quickly.

Risk accepted:

Sky stated:

```
We're just going to acknowledge permissionless asset inflows inflating rate limits for now.
At best it means we received a windfall, at worst it looks like allocators can "amalgamate"
rate limits (i.e. a deposit rate limit can be inflated if the withdrawal ends up receiving
more assets due to an lzReceive, for example, which consumed some layer zero facet rate limit)
```

The risk is primarily relevant when the allocator is also malicious.

5.2 Reentrancy Guard Is per-Controller While the ALMProxy Is Shared

Design | Low | Version 9 | Risk Accepted

CS-SKYDPAU-044

Facets are reached through `delegatecall` from the `Controller`, so the `nonReentrant` guard's storage lives in the `Controller` and the lock is scoped to a single `Controller`. The `ALMProxy` that custodies the funds holds no guard of its own.

As of Version 9 the documentation endorses backing a single `ALMProxy` with more than one `Controller`. The sibling Controllers then each have their own independent guard while sharing the same custody account, so an operation in flight on Controller A can be interleaved with one on Controller B even though both operate the same `ALMProxy`.

That is similar to Permissionless asset inflows inflate rate limits, except the inflow no longer needs a permissionless external source: a sibling Controller can be deliberately re-entered to move assets into the shared proxy mid-measurement. For example:

1. The allocator calls `ERC4626Facet.withdraw` on Controller A, which snapshots the proxy balance and enters its `doCall` to an untrusted `vault`.
2. The `vault` re-enters Controller B and pulls the same asset into the proxy. A's lock is set but B's is free, so the call proceeds.

3. Control returns to A, which attributes B's inflow to its own withdrawal, over-decrementing its withdraw limit and over-incrementing its deposit limit.

The same interleaving defeats other check-then-act invariants that rely on the single-Controller guard (slippage measurements, position-ownership checks, OTC swap readiness, pending-request lifecycles, ...).

The corruption works whether the Controllers share or dedicate their `RateLimits`, since the manipulated quantity is the shared `ALMProxy` balance, not rate-limit storage.

This requires a single `ALMProxy` shared by multiple Controllers, and a re-entering call that reaches a sibling Controller's state-changing function. Damage stays bounded by the configured rate limits and slippage, but a per-operation cap can be circumvented by repetition. This is a non-default configuration whose exploitability depends entirely on how the sibling Controllers and their roles are wired.

Risk accepted:

Sky is aware and accepts the risk. The reentrancy scenarios have been documented in `ARCHITECTURE.md`.

5.3 Centrifuge Deposit / Redemption Can Be DoSed by Cancellation

Security | Low | Version 1 | Risk Accepted

CS-SKYDPAU-001

The *relayers* (renamed to *allocators* in Version 6) can request to cancel pending deposits / redemptions on Centrifuge and claim them later once fulfilled. Note that in case there is a pending deposit cancellation, no new deposit can be made (same for redemption). Consequently, a compromised *relayer* can DoS new deposit requests by triggering a deposit cancellation (same for redemption) until the existing pending deposit is cancelled or fulfilled.

Risk accepted:

Sky is aware of the risk.

5.4 Centrifuge Tranche Token Price May Change Between Request Submission and Execution

Correctness | Low | Version 1 | Risk Accepted

CS-SKYDPAU-002

When requesting a redemption from a Centrifuge ERC-7540 vault, the rate limit is decreased by an estimation of the withdrawable assets (`convertToAssets(shares)`) based on the latest tranche token price.

However, the tranche token price may change between the redemption request submission and execution, hence the actual withdrawable assets after execution may not match the rate limit decreased at submission time.

5.5 Maple Redemption Can Be DoSed

Security | Low | Version 1 | Risk Accepted

CS-SKYDPAU-005

Maple redemption can be DoSed by a compromised *relayer* (renamed to *allocator* in Version 6) in two ways:

1. Each user can have at most 1 redemption request in MapleWithdrawalManager. Hence a compromised *relayer* can keep triggering dust redemptions and block the legitimate redemptions from honest *relayers*. In this case, the honest *relayers* have to cancel the dust redemptions first before triggering a legitimate one.
2. Requesting a maple redemption will consume rate limit, whereas cancelling a redemption will not recharge the limit. Consequently, if the whole rate limit is consumed by a compromised *relayer*, other *relayers* will not be able to trigger future redemptions.

Risk accepted:

Sky has added a test (`Attacks.t.sol`, `test_attack_compromisedRelayer_delayRequestMapleRedemption`) to demonstrate the mitigation: if a malicious *relayer* delays redemption, the *freezer* can remove the compromised *relayer* and revert to the governance *relayer*. This prevents the compromised *relayer* from continuing the attack, allowing the governance *relayer* to cancel and submit the legitimate request.

5.6 Over-reduced Limit in Maple Redemption

Correctness | Low | Version 1 | Risk Accepted

CS-SKYDPAU-006

In `requestMapleRedemption()`, the redemption limit will be reduced given the conversion rate between the shares and the assets with `convertToAssets()`.

In MaplePool, function `convertToAssets` assumes the pool holds `totalAsset()` without unrealized loss. However, when the redemption is processed with `processRedemptions()`, the withdrawable amount takes the unrealized loss into consideration.

Consequently, there would be a discrepancy between the rate limit decrease and the actual received tokens in the event of unrealized loss.

5.7 Relayer Can DoS SUSDE Unstaking

Security | Low | Version 1 | Risk Accepted

CS-SKYDPAU-007

In Ethena, two steps are required to convert sUSDe to USDe:

- A cooldown must be initiated first, which (1) burns the shares and credits the USDe to the USDeSilo contract (2) resets the `cooldownEnd` to be `cooldownDuration` from current `block.timestamp`. Note that step (2) will extend any existing cooldown asset to another `cooldownDuration`.
- When the `cooldownEnd` is reached, the sUSDe can be unstaked and the USDe will be credited to a specified receiver.

Consequently, a malicious *relayer* (renamed to *allocator* in Version 6) can keep triggering new cooldowns with as little as 1 wei asset to block previous exits from sUSDe to USDe, hence DoS the sUSDe to USDe conversion.

Note: Sky was aware of this issue and had reported to us before the audit. In case a malicious *relayer* DoSes the sUSDe `unstake()`, `FREEZER_ROLE` calls `removeAllocator()` on the `ALMProxy`, which revokes *relayer* role from the offending account.

5.8 Revoking Unused Approval

Design | Low | Version 1 | Acknowledged

CS-SKYDPAU-008

The *freezer* can remove *relayers* (renamed to *allocators* in Version 6) from the MainnetController which prevents *relayers* from triggering any more interactions or funds transfers from the ALMProxy.

However, since the Ethena integration requires actions from several external parties (see Allowance For Ethena Minter May Not Be Consumed), the actual transfers of underlying assets to mint or redeem USDe may happen even after the *relayer* is disabled.

Acknowledged:

Sky acknowledged the issue and decided not to change the code since Ethena is fully trusted.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Open Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	3

- Conflicting Slippage Constraints Block UniswapV3Facet.removeLiquidity Code Corrected
- ERC4626 withdraw Rate Limits Use the Input Amount Instead of the Asset Balance Delta Code Corrected
- Residual Asset Cross-Contamination Code Corrected

Low-Severity Findings	6

- Dynamic Call Targets Can Bypass Integration Boundaries Code Corrected
- Rate Limit Updates Not Defensive Code Corrected
- OTCFacet.claim Has No Rate Limit Gate Code Corrected
- Inconsistent Swap Rate Limit Decrease for Curve Code Corrected
- Lack of Rate Limits Code Corrected
- LayerZero Dangling Approvals Code Corrected

Informational Findings	15

- CurveFacet.removeLiquidity Validates Against Post-Execution get_virtual_price Code Corrected
- AccessControls.getRoleRevoker Returns Only Role Admin Code Corrected
- AccessControls Inherits ReentrancyGuard But Never Uses It Code Corrected
- ICurvePoolLike.coins Not Declared view Code Corrected
- WrapProxyETHFacet.wrapAll Short-Circuits Before The Rate-Limit Gate Code Corrected
- Dangling Approvals Code Corrected
- OTCFacet.send Shares One Rate Limit Across Whitelisted Assets Code Corrected
- PSMFacet Couples Both Swap Directions Under One Rate Limit Code Corrected
- Leftover optimizer_runs = 1 in foundry.toml Code Corrected
- RelayerRemoved Emitted Unconditionally Code Corrected
- Code Redundancy Code Corrected
- Inconsistency Code Corrected
- Interface Redundancy Code Corrected
- Typos Code Corrected
- msg.value Validation in transferTokenLayerZero Code Corrected

6.1 Conflicting Slippage Constraints Block `UniswapV3Facet.removeLiquidity`

Correctness | Medium | Version 6 | Code Corrected

CS-SKYDPAU-019

`UniswapV3Facet.removeLiquidity()` enforces two slippage constraints on the same `min.amount0` and `min.amount1` arguments. Once enough trading fees have accrued on the position, no value of these arguments satisfies both constraints simultaneously and the position can no longer be removed.

The function calls `decreaseLiquidity()`, then `collect()`, and then runs `_checkSlippage()` against the proxy balance delta:

```
require(minAmount >= (amount * maxSlippage) / 1e18, ...);
```

The measured `amount` here is `Principal + Fees`, so the facet imposes:

```
min.amount0 >= (Principal + Fees) * maxSlippage / 1e18
```

The same `min.amount0` is also forwarded to `NonfungiblePositionManager.decreaseLiquidity()` as `amount0Min`. Uniswap V3 enforces that the principal released by `decreaseLiquidity()` is at least `amount0Min`, and `decreaseLiquidity()` only operates on principal (fees are claimed separately by `collect()`). This gives:

```
min.amount0 <= Principal
```

The two constraints together require `(Principal + Fees) * maxSlippage / 1e18 <= min.amount0 <= Principal`, which has no solution when `Fees > Principal * (1e18 - maxSlippage) / maxSlippage`. For `maxSlippage = 0.99e18`, the position becomes unwithdrawable once accrued fees exceed roughly `1.01%` of the principal. Once fees on the position have accumulated past this threshold, the `ALLOCATOR_ROLE` can no longer remove the position via this facet and the principal is frozen.

This issue existed in a prior version of the controller and has been reintroduced.

Code corrected:

The issue is resolved by calling `collect()` to sweep accrued fees before measuring the proxy balance delta. The facet then calls `decreaseLiquidity()` followed by a second `collect()` to withdraw the principal. Because the fees are already extracted, the delta evaluated by `_checkSlippage()` consists only of the withdrawn principal.

6.2 ERC4626 `withdraw` Rate Limits Use the Input Amount Instead of the Asset Balance Delta

Security | Medium | Version 6 | Code Corrected

CS-SKYDPAU-026

`ERC4626Facet.withdraw()` decreases the withdraw rate limit and refills the deposit rate limit using the input `amount` parameter, without verifying how many assets the `ALMProxy` actually received from the vault:

```
function withdraw(address token, uint256 amount, uint256 maxSharesIn)
    external
    ...
    returns (uint256 shares)
{
    _decreaseRateLimit(getWithdrawRateLimitKey(token), amount);

    address proxy = _getSharedControllerStorage().proxy;

    uint256 startingShares = IERC20Like(token).balanceOf(proxy);

    IALMProxy(proxy).doCall(
        token,
        abi.encodeCall(IERC4626Like.withdraw, (amount, proxy, proxy))
    );

    shares = startingShares - IERC20Like(token).balanceOf(proxy);

    require(shares <= maxSharesIn, "ERC4626Facet/shares-burned-too-high");

    _tryIncreaseRateLimit(
        getDepositRateLimitKey(token, IERC4626Like(token).asset()), amount
    );

    ...
}
```

Only the share balance delta is measured; the asset side is never checked. If a compromised or malicious vault transfers back fewer assets than `amount` while still burning shares, the deposit rate limit for that `(asset, vault)` pair is refilled by `amount` regardless of what the `ALMProxy` actually received.

A compromised `ALLOCATOR_ROLE` can use this against a compromised vault to inflate the deposit allowance for that pair, then deposit the `ALMProxy`'s independently held `asset` balance into the same vault, draining assets that the deposit rate limit is meant to cap.

`redeem()` in the same facet measures the asset balance delta on the `ALMProxy` after the external call and feeds that delta into both rate limit updates, so it is not affected.

Code corrected:

The balance delta of the asset is now tracked and the amount of assets actually received is used to update the rate limit.

6.3 Residual Asset Cross-Contamination

Security | Medium | Version 5 | Code Corrected

CS-SKYDPAU-038

In Version 5, the trust model was tightened: upgradeable systems and class-level integrations (e.g. ERC-4626) are now considered maliciously mutable. Sky specified that no rate-limit violation should be possible in such cases. For example, if an ERC-4626 vault whose underlying was USDS is upgraded to use USDC, the integration must not be able to pull USDC through it. The remediation keys rate limits on `(integration, token)` pairs rather than on the integration alone.

Note that in various instances the result of two identical staticcalls without any external interaction can change due to gas introspection or to changes in observable state.

Below is a list of violations:

1. `AaveFacet`: Aave could be upgraded so that `UNDERLYING_ASSET_ADDRESS` returns different values across calls. This means the expected underlying could be used for the rate limit while an unexpected one could be used for the approval given in `deposit`. The same applies to `withdraw()`, where `amountWithdrawn` could be inflated by fake tokens so that the deposit rate limit is improperly increased.
2. `ERC4626Facet`: Similarly, `asset()` could be manipulated so that `deposit()` pulls an incorrect token, or so that `redeem` computes the `assets` balance delta against an incorrect token, manipulating the increase/decrease rate-limit logic.
3. `ERC7540Facet`: Similarly, `asset()` could be manipulated so that `requestDeposit()` pulls an incorrect token.
4. `WEETHFacet`: Similarly, the `weETH` contract could lie about the underlying `eETH`, so that an unexpected token is pulled or the rate-limit accounting is bypassed.

Note that for other facets similar scenarios can occur, but the integrations are trusted, non-upgradeable, or otherwise mitigated. For instance, `CentrifugeFacet.transferShares` lets `baseManager`, `spoke()`, `poolId()` and `scId()` be manipulated so that an unexpected token is pulled, but no approvals are given and Centrifuge could perform such an action at any time anyway.

Code corrected:

In Version 6 addresses derived from integrated contracts are read once at the start of each function and reused for the rate limit key, approvals, balance measurements, and the call. The rate limit accounting and the operation always reference the same value.

6.4 Dynamic Call Targets Can Bypass Integration Boundaries

Design | Low | Version 5 | Code Corrected

CS-SKYDPAU-040

Some facets derive external call targets from integrated contracts instead of binding them directly to the configured action. This can cause the proxy to interact with an unexpected external system while rate-limit accounting is still attributed to the originally supplied integration.

The clearest example is `AaveFacet`. The `POOL` return value might be manipulated, leading to unexpected `supply` / `withdraw` calls. This is most relevant when multiple Aave-like protocols are supported at once. Because `withdraw` replenishes the deposit limit for the supplied `aToken` based on the observed underlying received, this can let one configured pool replenish limits for another configured integration. For example, a compromised Aave `aToken` could route `withdraw` to a Spark pool, causing Spark liquidity to be withdrawn while replenishing the compromised Aave deposit limit. A compromised relayer (renamed to allocator in Version 6) could then use the replenished limit to push more underlying into the broken or malicious Aave integration than intended.

Other examples include:

#. `FarmFacet`: `getReward()` is called on an arbitrary `farm` target, leading to unforeseeable consequences. #. `CentrifugeFacet`: `transferShares` invokes `crosschainTransferShares` on an arbitrary `spoke()` if the implementation is malicious. Centrifuge is not expected to behave maliciously, but a compromised or non-canonical spoke could lead to unforeseeable consequences. #.

`WEETHFacet`: The `liquidityPool()` is an unvalidated contract on which both `deposit` (with ETH value) and `requestWithdraw` are called, with potentially unforeseeable consequences.

Note that other facets could also expose dynamic call targets, but the relevant integrations are assumed to be fully trusted or immutable.

Code corrected:

In Version 6 addresses derived from integrated contracts are read once at the start of each function and reused for the rate limit key, approvals, and the call. The rate limit accounting and the operation always reference the same value. This was applied to `AaveFacet` (`pool`, `underlying`), `CentrifugeFacet` (`spoke`) and `WEETHFacet` (`eeth`, `liquidityPool`).

`FarmFacet.claimReward()` now requires a rate limit to be configured for the `farm`.

6.5 Rate Limit Updates Not Defensive

Design | Low | Version 5 | Code Corrected

CS-SKYDPAU-039

Some rate limits could be more defensive. For example, decreases could use the larger of the intended amount and the actual amount changed, while increases can use the smaller of the intended amount and the actual amount changed. For example:

- `AaveFacet`: In `withdraw`, the rate-limit decrease might be too small. While the intended decrease was `amount`, a malicious implementation could return a smaller `amountWithdrawn` (actual balance delta), so the limit is consumed by less than intended. The same applies to the corresponding rate-limit increase.

Note that similar considerations apply to other facets, but the integrated contracts are trusted, non-upgradeable, or otherwise mitigated. A few notable cases:

- Asynchronous flows leave less room for defensive accounting. `MapleFacet` relies on `convertToAssets` (the upgradeable `manager`), which could under-report on redemptions. The same applies to `ERC7540Facet` (also via `convertToAssets`) and `WSTETHFacet` (via `getStETHByWstETH`, where `stETH` is upgradeable).
- The integration behind `LayerZeroFacet` could burn more than `amount` from the proxy directly (in burn-and-mint mode, allowance does not apply), and the integration behind `WEETHFacet` could misreport underlying balances. However, in those cases they could do far worse directly with the underlying tokens, since those funds are already under their control.

Code corrected:

`AaveFacet` now uses pre- and post-balance delta as amountWithdrawn.

6.6 `OTCFacet.claim` Has No Rate Limit Gate

Design | Low | Version 5 | Code Corrected

CS-SKYDPAU-027

`OTCFacet.claim()` is gated only by the per asset whitelist managed via `setIsWhitelisted`; no rate limit is decreased or required to exist. This is inconsistent with claim and withdraw style operations elsewhere in the controller, which are either throttled (`ERC4626Facet.withdraw`, `AaveFacet.withdraw`) or toggleable via a rate limit existence check

(`CentrifugeFacet.claimCancelDepositRequest`). `OTCFacet.claim` cannot therefore be disabled or throttled per `(exchange, asset)` pair through the rate limit configuration alone.

Code corrected:

In Version 7 `OTCFacet.claim()` is gated by the existence of a rate limit at `getClaimRateLimitKey(exchange, asset) = makeAddressAddressKey(_LIMIT_CLAIM, asset, exchange)`. This adopts the `CentrifugeFacet.claimCancelDepositRequest` toggle pattern named in the original finding, so the admin can disable a specific `(exchange, asset)` claim path by not configuring the corresponding limit. Throttling was not added: the claim amount is whatever the buffer returned (`IERC20Like(asset).balanceOf(buffer)`), not an allocator-chosen quantity, so a per-call value cap would bound how much the exchange can return rather than how much value flows out of the controller.

6.7 Inconsistent Swap Rate Limit Decrease for Curve

Design Low Version 1 Code Corrected

CS-SKYDPAU-041

When adding liquidity to Curve, a swap can occur. Note that the rate limit adjustment is inconsistent with the adjustment in the swap function. Consider the following example:

1. Assume that swapping 50 token A returns 49 token B.
2. When using the swap function, the rate limit is reduced by 50.
3. Assume that when adding liquidity with 100 token A and 0 token B, the internal swap swaps so that 50 token A and 49 token B are added.
4. The swap performed is effectively the swap from 1.
5. However, the rate limit adjustment will be the average of the input and output deltas and will thus be 49.5.

Ultimately, there can be swap rate limit discrepancies between swap and adding liquidity. However, note that this is intended according to Sky.

Code corrected:

In Version 6, `addLiquidity` now identifies, per token, the net amount swapped in by comparing the deposited amount to the underlying share of the minted LP (`_getSwappedInAmounts`). For the token whose deposit exceeds that share (the input side of the implicit swap), the swap rate limit is decreased by exactly that excess, matching the decrease applied by `swap()`.

6.8 Lack of Rate Limits

Design Low Version 1 Code Corrected

CS-SKYDPAU-043

The LayerZero integration might pay fees in native tokens. However, the outbound amount of native tokens is not rate limited. Thus, a *relayer* (renamed to *allocator* in Version 6) colluding with an endpoint operator could drain the native token balance.

Code corrected:

The native fee is capped by the `msg.value`, hence must be provided by the allocator.

6.9 LayerZero Dangling Approvals

Correctness | Low | Version 1 | Code Corrected

CS-SKYDPAU-004

Pending approvals might remain. Namely, the `OFTReceipt` contains the `amountSentLD` and the `amountReceivedLD` amounts where `amountSentLD` corresponds to the amount actually debited from the user. Hence, `send()` could potentially pull less tokens than approved (e.g. LayerZero dust removal) and pending approvals could exist. Optimally, the approvals should be revoked to prevent dangling approvals.

Note that pending approvals might introduce a corner case if ZRO are held and bridged. Namely, a dangling approval could allow for a `lzTokenFee > 0` to be collected for ZRO OFTs.

Code corrected:

As of Version 7, Sky revokes all potentially dangling approvals.

6.10 `CurveFacet.removeLiquidity` Validates Against Post-Execution `get_virtual_price`

Informational | Version 7 | Code Corrected

CS-SKYDPAU-046

`CurveFacet.removeLiquidity` reads `get_virtual_price()` inside `_validateRemoveLiquidity` after `remove_liquidity` has already executed. For a pro-rata withdrawal Curve's integer arithmetic rounds in the pool's favor, so the post-call value is greater than or equal to the value that held when the allocator chose `minWithdrawAmounts`. The right-hand side of the slippage check grows by that small amount, making the check more restrictive than intended.

In contrast, `addLiquidity` samples `virtualPrice` before the `doCall` and reuses that snapshot inside its validator, which is the consistent ordering.

The per-call increase of `get_virtual_price()` is tiny relative to the headroom provided by `maxSlippage`, so legitimate calls are not expected to revert because of this alone.

Code corrected:

The virtual price is now cached before the operations.

6.11 `AccessControls.getRoleRevoker` Returns Only Role Admin

Informational | Version 6 | Code Corrected

CS-SKYDPAU-036

`AccessControls.revokeRole()` is gated by `onlyRoleAdminOrDefaultAdmin`, which permits both the role admin and any holder of `DEFAULT_ADMIN_ROLE` to revoke a role:

```
require(
    hasRole(roleAdmin, msg.sender) || hasRole(DEFAULT_ADMIN_ROLE, msg.sender),
    NotRoleAdminOrDefaultAdmin(msg.sender, role, roleAdmin)
);
```

However, the public view `getRoleRevoker(role)` returns only `getRoleAdmin(role)`:

```
function getRoleRevoker(bytes32 role) external view override returns (bytes32) {
    return getRoleAdmin(role);
}
```

The function name suggests it returns the set of roles whose holders can revoke `role`, but it omits `DEFAULT_ADMIN_ROLE`. Off-chain consumers querying `getRoleRevoker` to determine who can revoke a given role receive an incomplete answer.

Code corrected:

The logic has been removed as part of a functional change in Version 7.

6.12 `AccessControls` Inherits `ReentrancyGuard` But Never Uses It

Informational | Version 6 | Code Corrected

CS-SKYDPAU-029

`AccessControls` imports and inherits OpenZeppelin's `ReentrancyGuard`. However, none of the contract's external functions are decorated with the `nonReentrant` modifier, and the inherited `_status` slot is never read or written.

Code corrected:

The reentrancy guard is not inherited anymore.

6.13 `ICurvePoolLike.coins` Not Declared `view`

Informational | Version 6 | Code Corrected

CS-SKYDPAU-030

In `CurveFacet`, the local `ICurvePoolLike` interface declares `coins` as a non-view function.

```
function coins(uint256 index) external returns (address);
```

The actual Curve StableswapNG `coins(uint256)` is a public storage getter and has no side effects.

Code corrected:

`coins` is now declared as view function.

6.14 `WrapProxyETHFacet.wrapAll` Short-Circuits Before The Rate-Limit Gate

Informational **Version 6** **Code Corrected**

CS-SKYDPAU-032

`WrapProxyETHFacet.wrapAll()` returns early when the proxy holds no ETH, before consulting the rate-limit existence check:

```
uint256 ethAmount = proxy.balance;

if (ethAmount == 0) return;

require(_rateLimitExists(wrapRateLimitKey()), "WrapProxyETHFacet/invalid-action");
```

When `ethAmount == 0` the function succeeds silently without verifying that `LIMIT_WRAP_PROXY_ETH` has been configured.

The behavior has no direct fund impact: with a zero balance there is nothing to wrap, no event is emitted, and no external call is made. The deviation matters as a design-consistency observation: among every allocator-gated function in the codebase, `wrapAll()` is the only entry point that can complete successfully without its rate-limit-existence gate ever being evaluated. Every other `LIMIT_*`-existence-only action checks the gate unconditionally at the top of the function.

Code corrected:

The rate limit check is now performed at the beginning of the function.

6.15 Dangling Approvals

Informational **Version 5** **Code Corrected**

CS-SKYDPAU-033

Dangling approvals could exist when an integration does not consume the full approved amount. While they are bounded by rate-limit consumption, it would be more defensive to zero out approvals when they remain unconsumed. Below, some potentially dangling approvals are listed:

1. `AaveFacet`: A malicious implementation could leave part of the approval unconsumed for various reasons (e.g. not pulling sufficient tokens).
2. `ERC4626Facet`: As above.
3. `ERC7540Facet`: As above.
4. `CCTPFacet`: As above.
5. `SuperstateFacet`: As above.
6. `WSTETHFacet`: As above; the withdrawal queue may not consume the full approval.
7. `LayerZeroFacet`: No consumption of approval given (funds may not be pulled even when `approvalRequired()` returns `true`). See also LayerZero Dangling Approvals.
8. `USDEFacet`: Dangling approvals might persist due to the way the integration's flow consumes approvals.

Note that the same pattern exists in other facets (e.g. `PendleFacet`, `CurveFacet`, `PSM3Facet`), but the relevant integrations are trusted, non-upgradeable, or otherwise mitigated.

Code corrected:

The code has been adjusted. All operations approving are resetting the approvals to 0. The only exception is `EthenaFacet` that keeps the approvals due to how the underlying mechanism works.

6.16 `OTCFacet.send` Shares One Rate Limit Across Whitelisted Assets

Informational **Version 5** **Code Corrected**

CS-SKYDPAU-034

`OTCFacet.send()` consumes a single rate limit per exchange (`getSwapRateLimitKey(exchange)`), denominated in 18-decimal normalized units, regardless of which whitelisted asset is being sent. The *admin* whitelists assets per exchange via `setIsWhitelisted()`; all of those assets share the same per-exchange budget when they are sent.

```
uint256 normalizedSent = _toNormalizedAmount(assetToSend, amount);
_decreaseRateLimit(getSwapRateLimitKey(exchange), normalizedSent);
```

The accounting treats every whitelisted asset as 1:1 with the shared 18-decimal unit. Two consequences follow:

1. If any whitelisted asset for an exchange depegs from its expected price, the rate limit consumption no longer reflects the value sent. A depegged asset can be sent at the same normalized cost as a pegged one.
2. The *admin* cannot independently bound the volume of each whitelisted asset under a single exchange. Pausing one asset for that exchange while keeping others active requires removing the asset from the whitelist rather than adjusting a per-asset rate limit.

This deviates from the per-(integration, token) keying used in other facets where multiple tokens can flow through a single integration (e.g. `AaveFacet`, `ERC4626Facet`, `PSM3Facet`, `BasinFacet`).

Code corrected:

In Version 7 the send rate limit is keyed per `(asset, exchange)` (`makeAddressAddressKey(_LIMIT_SEND, asset, exchange)`) and decremented by the raw `amount` in the asset's native units, rather than by a single per-exchange limit consumed in 18-decimal normalized units. The admin configures a separate budget for each `(asset, exchange)` pair, and a depegged asset can no longer draw down a budget shared with pegged assets.

6.17 `PSMFacet` Couples Both Swap Directions Under One Rate Limit

Informational **Version 5** **Code Corrected**

CS-SKYDPAU-035

PSMFacet governs both swap directions under a single bidirectional key, LIMIT_USDS_TO_USDC: swapUSDSToUSDC() consumes the limit and swapUSDCToUSDS() replenishes it. As a consequence, the *admin* cannot independently throttle or pause one direction:

- Setting maxAmount = 0 on LIMIT_USDS_TO_USDC blocks swapUSDSToUSDC() but also makes swapUSDCToUSDS() revert in triggerRateLimitIncrease(), since the rate limit must be configured for the increase to apply.
- Similarly, lowering the maxAmount to throttle USDS → USDC also caps how much USDC → USDS replenishment is observable at once.

This deviates from facets that expose independent per-direction limits (e.g. PSM3Facet, where LIMIT_PSM_DEPOSIT and LIMIT_PSM_WITHDRAW are independent and the reverse direction does not replenish the forward limit). In a hypothetical USDC depeg scenario, the PSM3Facet-style separation would allow blocking USDS to USDC while still permitting USDC to USDS; the PSMFacet design does not.

Code corrected:

The code has been adjusted to introduce a rate-limit for the USDC-to-USDS direction. Note that:

- swapUSDSToUSDC(): only decreases its directional rate-limit.
- swapUSDCToUSDS(): decreases its directional rate-limit but also replenishes the rate-limit of the other direction.

6.18 Leftover optimizer_runs = 1 in foundry.toml

Informational | Version 4 | Code Corrected

CS-SKYDPAU-024

foundry.toml sets optimizer_runs = 1. Setting optimizer_runs to a low value optimizes for deployment cost and typically produces smaller bytecode. In the previous design with a monolithic controller this helped keep the contract below the size limit. In the current architecture each facet is deployed separately and is well below the size limit, so this choice of optimization no longer seems necessary.

Code corrected:

foundry.toml now sets optimizer_runs = 200.

6.19 RelayerRemoved Emitted Unconditionally

Informational | Version 4 | Code Corrected

CS-SKYDPAU-025

Both ALMProxyFreezable.removeRelayer and AccessControls.removeRelayer call _revokeRole and emit RelayerRemoved unconditionally. When the target address does not currently hold the RELAYER role, _revokeRole returns false and no state change occurs, and OpenZeppelin's underlying RoleRevoked event is not emitted, yet RelayerRemoved is still emitted.

Code corrected:

Both functions now require `_revokeRole` to return `true`, reverting when the target does not hold the RELAYER role and preventing the spurious `RelayerRemoved` emission. Note relayer is renamed to allocator in Version 6.

6.20 Code Redundancy

Informational **Version 1** **Code Corrected**

CS-SKYDPAU-009

1. `ApproveLib` is imported but never directly used in MainnetController.
2. Duplicated functions: RateLimitHelpers library block duplicates the free functions.

Code corrected:

Sky reduced redundancy.

6.21 Inconsistency

Informational **Version 1** **Code Corrected**

CS-SKYDPAU-011

Below are still-existing inconsistencies from versions before Version 4:

1. ApproveLib: Most libraries embed the ApproveLib for token approval logic, while some libraries use customized approval logic (i.e. PSMLib, CCTPLib and WSTETHLib). As of Version 4, they carry explicit comments. However, `WSTETHFacet.requestWithdraw` still issues a raw `IALMProxy.doCall(wsteth, ... approve ...)` without `ApproveLib`, without a justifying comment.
2. Rate Limit Invocation: In some Libs, rate limit operation is inlined, while in others it is wrapped into an internal function. *Note that the pattern of using internal functions is that as soon as multiple rate limit interactions with equal "key types" exist, an internal function is used.*
3. Constants Naming: operation keys are defined as constants in respective libraries. Most constant names are defined without integration name, while some constant names still contain the integration name (`LIMIT_USDE_BURN`, `LIMIT_USDE_MINT`, `LIMIT_SUSDE_COOLDOWN`, `LIMIT_USDS_TO_USDC`). *Note that this is expected due to preexisting configurations.*
4. Sanity Checks: some integration-related addresses (pendleRouter, merklDistributor, uniswapV3PositionManager, uniswapV3Router) are not configured in the constructor hence staying 0 before they are configured. It is not checked in the UniswapV3Lib that the respective addresses (uniswapV3Router, uniswapV3PositionManager) are non-zero before the logic execution.
5. Function Arguments: Named argument syntax is used in some integrations but not others (positional arguments), leading to inconsistent readability and robustness across different integrations.
6. External Function librarification: Not all external functions are librarificated, in particular:
    1. Some configuration setters are not librarificated (e.g. `setPendleRouter()`, `setMintRecipient()`).
    2. The access control related functionalities are not librarificated. (e.g. `grantRole()`, `revokeRole()`, `removeRelayer()`).

3. Auto generated Getters are not librarificated.
4. Events of some integrations are refactored to libraries while others stay in the controller (e.g. `PendleRouterSet`).

7. Interfaces: Most contracts define minimal interfaces to use, while PendleLib, TransferAssetLib and MapleLib import OZ interfaces.
8. Address Passing: Most libraries accept the target integration contract as argument, while some libraries hardcode the integration contract addresses (e.g. WEETHLib, UniswapV4Lib).
9. The named storage location for `WEETHModule` and `OTCBuffer` was not migrated to the convention used by the other contracts. It would be preferable to follow the `sky.pau.storage` convention.

Below is a non-exhaustive list of inconsistencies included in Version 4:

1. Role Naming: `ALMProxyFreezable` declares role constants as `FREEZER` and `RELAYER`, while the rest of the codebase uses the `<ROLE>_ROLE` suffix (e.g. `FacetBase.RELAYER_ROLE`, `FacetBase.DEFAULT_ADMIN_ROLE`).
2. Event Emission Location: `PSMFacet.swapUSDCToUSDS` emits `PSMSwapUSDCToUSDS` mid-function while every other facet emits the corresponding event at the end of the call.
3. View vs Pure on Rate-Limit Key Getters: `IUSDEFacet` declares `LIMIT_USDE_BURN()`, `LIMIT_USDE_MINT()` and `LIMIT_SUSDE_COOLDOWN()` as `external view`, while all other facet interfaces declare their `LIMIT_*` getters as `external pure`.
4. Event Naming (Facet suffix): Most events follow the `<IntegrationName><Action>` convention (e.g. `AaveDeposit`), while `TransferAssetFacetTransfer` is the only event carrying "Facet" in the name. The consistent name would be `TransferAssetTransfer`.
5. Event Naming (Updated vs Set): Most admin events use the `Set` suffix (e.g. `UniswapV3MaxSlippageSet`, `UniswapV3MaxTickDeltaSet`, `UniswapV4TickLimitsSet`), while `UniswapV3Facet` has three admin events with the `Updated` suffix (e.g. `UniswapV3LowerTickUpdated`).
6. Approval Reset to Zero: `UniswapV3Facet` and `UniswapV4Facet` reset token approvals to `0` after every operation (swap, addLiquidity, increase/decrease), while no other facet clears residual approvals. Note that this is not a functional bug since `ApproveLib` supports non-zero-to-non-zero transitions, but is an inconsistent defense-in-depth posture.
7. `DEFAULT_ADMIN_ROLE` functionality lacks sanity checks (e.g. boundary checks in `setMaxSlippage()`, `UniswapV3Facet` missing various `pool != 0` checks and similarly for `UniswapV4Facet`).

Code corrected:

Most of the above has been addressed. Below is a list of remaining and acceptable inconsistencies:

- Pre-Version 4 #5: *Mostly resolved*: Still not uniform across internal calls but readability has been addressed.
- Version 4 #2: *Unaddressed*: However, the position of the event emission has no impact here.
- Version 4 #7: *Mostly resolved*: More checks have been added.

6.22 Interface Redundancy

Informational Version 1 Code Corrected

CS-SKYDPAU-042

Some interfaces are defined multiple times across different contracts (e.g. IERC20Like, IERC4626Like).

6.23 Typos

Informational **Version 1** **Code Corrected**

CS-SKYDPAU-012

1. In WSTETHLib: `IERC20ike`.
2. In USDSLib: `ApprobeLib`.
3. In MainnetController: `cancelation`.

Code corrected:

All the typos have been corrected.

6.24 `msg.value` Validation in `transferTokenLayerZero`

Informational **Version 1** **Code Corrected**

CS-SKYDPAU-021

The relayer (renamed to allocator in **Version 6**) attaches `msg.value` to calls to `transferTokenLayerZero()` to pay for the LayerZero V2 fees. Note that there might be two scenarios:

- The relayer does not provide sufficient value (e.g. pricing changed between transaction sending and arrival). Then, if the controller does not hold the relevant native token delta, the call reverts. However, that works as expected.
- Similarly, the relayer might provide a `msg.value` that is too high due to similar reasons. In such cases the native token could be stuck in the controller.

Ultimately, the second scenario could lead to native tokens in the controller. Typically, they will not be used. However, technically the relayer could reuse them for future LayerZero V2 fees. However, refunding the relayer with remaining delta might be more meaningful.

Code corrected:

The excess Ether will now be refunded back to the message sender.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 `UniswapV3Facet.addLiquidity` Queries `token0` / `token1` Twice

Informational **Version 6** **Acknowledged**

CS-SKYDPAU-031

`UniswapV3Facet.addLiquidity()` reads the pool's `token0` and `token1` twice along the same call path:

1. In `addLiquidity()` itself, to set the position manager approvals and to key the rate-limit decreases.
2. In `_mintLiquidity()`, to populate the `MintParams` passed to the position manager.

The two staticcalls are independent. If a pool returned different addresses between the two reads (e.g. via gas-introspection-based branching or any other observable-state change), the approvals and rate-limit decreases would be attributed to one pair while the position manager would mint against another, mirroring the cross-contamination class described in Residual Asset Cross-Contamination.

Uniswap V3 pool contracts are trusted to function correctly per the system overview, and a legitimate pool's `token0` / `token1` are immutable, so this scenario is not expected to occur in practice.

Acknowledged:

Sky states:

```
Acknowledged, hitting stack too deep, and gas savings are inconsequential.
```

7.2 `UniswapV3Facet` Slippage Helpers Use Different Arithmetic

Informational **Version 6** **Acknowledged**

CS-SKYDPAU-037

`UniswapV3Facet` defines two internal helpers that compute the same slippage threshold `expectedAmount * maxSlippage / 1e18`:

1. `_checkSlippage()` uses native `uint256` arithmetic.
2. `_validateMinAmount()` uses `FullMath.mulDiv()`.

Both helpers carry an inline comment stating they are "effectively the same as" the other. The native multiplication can overflow on very large amounts where `FullMath.mulDiv()` would not, so the two helpers are not strictly equivalent for arbitrary inputs, even though in practice both call sites operate on values bounded well below `2**128`.

7.3 Documentation Inconsistencies

Informational **Version 4** **Specification Partially Changed**

CS-SKYDPAU-023

Several statements across the `docs/` folder do not match the implementation. They are grouped below by source file.

SECURITY.md:

1. `doCallWithValue` is described as accessible only to `DEFAULT_ADMIN_ROLE` (governance). In practice, `ALMProxy.doCallWithValue()` is gated by the `CONTROLLER` role and `ALMProxyFreezable.doCallWithValue()` by the `Allocator` role; neither uses `DEFAULT_ADMIN_ROLE`. The follow-up claim that the function is "governance-controlled and does not introduce attack vectors for compromised relayers" is therefore also incorrect for the freezable variant, where a compromised relayer (renamed to allocator in **Version 6**) can call it directly.
2. The wrapping helper is referred to as `wrapAllProxyETH`. The actual function on `WrapProxyETHFacet` is `wrapAll`.

ARCHITECTURE.md:

1. It is claimed that all contracts in the repository inherit and implement OpenZeppelin's `AccessControl`. `Controller`, `PAUFactory`, `ControllerSharedStorage` and the facets (via `FacetBase`) do not inherit `AccessControl` directly; they rely on an external `AccessControls` contract for role checks.
2. "Reentrancy protection across all facet calls" is listed among the Controller's key characteristics, suggesting the Controller itself enforces it. `Controller.fallback()` applies no reentrancy guard; protection is instead implemented per function at the facet level, via `FacetBase`'s inherited `ReentrancyGuard`.
3. `FacetBase` is described as "inheriting `ControllerSharedStorage` and `ReentrancyGuard`, providing the `onlyRole` modifier". The `onlyRole` modifier is defined directly in `FacetBase`; neither `ControllerSharedStorage` nor `ReentrancyGuard` supplies it.
4. The illustrations under sections *General Call Flow* and *Example: Minting USDS* are outdated: they still show examples using `MainnetController`, label the caller "ALM Planner" instead of "Allocator", and depict the call to `RateLimits` without the new corresponding call to `AccessControls`.

LIQUIDITY_OPERATIONS.md:

1. All Uniswap V3 operations are said to require `maxSlippage` to be configured. This holds for `addLiquidity` and `removeLiquidity`, but `UniswapV3Facet.swap()` does not read `maxSlippage`; it validates through `tickDelta`, TWAP bounds and `minAmountOut` instead.
2. The same claim is made for Uniswap V4. Only `UniswapV4Facet.swap()` enforces `maxSlippage`; `mintPosition`, `increasePosition` and `decreasePosition` rely on caller-supplied `amount0Max/Min` and `amount1Max/Min` bounds.
3. PSM3 rate-limit-decreasing functions are referred to as `depositPSM` and `withdrawPSM`. `PSM3Facet` exposes them as `deposit` and `withdraw`.

WEETH_INTEGRATION.md:

1. The three entry points are referenced as `Controller.depositToWeETH`, `Controller.requestWithdrawFromWeETH` and `Controller.claimWithdrawalFromWeETH`. The functions on `WEETHFacet` are named `deposit`, `requestWithdraw` and `claimWithdrawal`.

DEVELOPMENT.md:

1. The ERC-7201 namespace pattern is documented as `sky.pau.storage.<FacetName>` (with the example `sky.pau.storage.AaveFacet`). All facets actually use the suffixed form `sky.pau.storage.<FacetName>.v1`.

CODE_NOTES.md:

1. The errors `ZeroAccessControls`, `ZeroProxy` and `ZeroRateLimits` declared in `IController` are not listed in the error table.

Cross-cutting:

1. Role constants are defined inconsistently: `AccessControls` uses `FREEZER_ROLE` / `RELAYER_ROLE` while `ALMProxyFreezable` uses `FREEZER` / `RELAYER` (both hash to the same value).
2. Several events declared in the public interfaces are not referenced anywhere in `docs/`: `IRateLimits.RateLimitDataSet`, `IRateLimits.RateLimitDecreaseTriggered`, `IRateLimits.RateLimitIncreaseTriggered`, `IAccessControls.RelayerRemoved` and `IALMProxyFreezable.RelayerRemoved`.

Additional inconsistencies identified in Version 5:

1. ARCHITECTURE.md: References to `FacetBase` were not updated when the base contract was renamed to `Facet` in the source.
2. WEETH_INTEGRATION.md: The rate limit key descriptions were not updated when `WEETHFacet._getRequestWithdrawRateLimitKey` started keying by `(eETH, weethModule)` via `makeAddressAddressKey`; the doc still describes the key as keyed by the `weETHModule` address only and references `makeAddressKey(LIMIT_WEETH_REQUEST_WITHDRAW, weETHModule)`.
3. CODE_NOTES.md: The `makeAddressAddressKey` row in the `RateLimitHelpers` table was not updated when `AaveFacet` (deposit), `ERC4626Facet` (deposit), `ERC7540Facet` (deposit), `CentrifugeFacet` (deposit), `FarmFacet` (deposit), `PendleFacet` (redeem) and `WEETHFacet` (request withdraw) started using it; it still lists only `TransferAssetFacet`, `UniswapV3Facet` and `BasinFacet`.
4. `ILayerZeroFacet.LIMIT_TRANSFER` NatSpec was not updated when `LayerZeroFacet.transfer` started keying by `(token, oftAddress, destinationEndpointId)` via `makeAddressAddressUint32Key`; it still describes the key as only the OFT address and destination endpoint ID.
5. `IPendleFacet.LIMIT_REDEEM` NatSpec was not updated when `PendleFacet._getRedeemRateLimitKey` started keying by `(pt, market)` via `makeAddressAddressKey`; it still describes the key as only the market address.

Additional inconsistencies identified in Version 6:

1. ARCHITECTURE.md: The `AccessControls` description states that the contract "Declares `FREEZER_ROLE` and `ALLOCATOR_ROLE` as constants" and "Provides a `removeAllocator` function gated by `FREEZER_ROLE`". In Version 6 `AccessControls` is generic, declares no role constants, and has no `removeAllocator` function. The freezer-revokes-allocator behavior on the `Controller` path is enforced only after the admin calls `AccessControls.setRoleRevoker(ALLOCATOR_ROLE, FREEZER_ROLE)`.
2. ARCHITECTURE.md and CODE_NOTES.md: still reference `USDEFacet` (in the facet table and the "facets without custom error messages" list); the facet was renamed to `EthenaFacet` in Version 6.
3. THREAT_MODEL.md: presents `FREEZER_ROLE` revoking `ALLOCATOR_ROLE` on the `Controller` as an architectural property. In Version 6 it is enforced only once the admin has configured a role revoker via `AccessControls.setRoleRevoker`; `removeAllocator` is hardcoded only on `ALMProxyFreezable`.

4. RATE_LIMITS.md: the example `keccak256(abi.encode(LIMIT_AAVE_WITHDRAW, aToken))` is incomplete; the Version 6 withdraw key is `makeAddressAddressKey(LIMIT_AAVE_WITHDRAW, pool, aToken)`.
5. RATE_LIMITS.md and CODE_NOTES.md: the gate-check examples still describe `LIMIT_REQUEST_WITHDRAW` for `WSTETHFacet.claimWithdrawal` and `WEETHFacet.claimWithdrawal`. Version 6 introduced s `LIMIT_WSTETH_CLAIM_WITHDRAW` and `LIMIT_WEETH_CLAIM_WITHDRAW` keys; the docs were not updated.
6. WEETH_INTEGRATION.md: the gate-check description for `claimWithdrawal` likewise still references `LIMIT_REQUEST_WITHDRAW` instead of the new `LIMIT_WEETH_CLAIM_WITHDRAW`.
7. CODE_NOTES.md `RateLimitHelpers` table: the `makeAddressUint16Key` row still names `CentrifugeFacet`, but in Version 6 `CentrifugeFacet.transferShares` uses `makeAddressUint16AddressKey` keyed by `(token, centrifugeId, spoke)`; `makeAddressUint16Key` is no longer used by any facet. The table also omits `makeAddressAddressAddressKey` (used by `AaveFacet` deposit and `WEETHFacet` request-withdraw) and `makeAddressUint16AddressKey`.
8. CODE_NOTES.md custom-error table: still missing `ZeroAccessControls`, `ZeroProxy` and `ZeroRateLimits` (carried over from Version 4), and additionally missing the Version 6-introduced `NotRoleAdminOrDefaultAdmin` and `RoleNotGranted` (`IAccessControls`) and `AccessControlUnauthorizedAccount` (`IFacet`).
9. OPERATIONAL_REQUIREMENTS.md Withdrawal Dependencies table uses pre-diamond function names (`withdrawERC4626`, `redeemERC4626`, `withdrawAave`, `claimWithdrawalFromWstETH`, `claimWithdrawalFromWeETH`); the Version 6 equivalents are `ERC4626Facet.withdraw` / `redeem`, `AaveFacet.withdraw`, `WSTETHFacet.claimWithdrawal` and `WEETHFacet.claimWithdrawal`. The Aave entry's prerequisite is also incomplete: Version 6 keys the deposit limit by `(underlying, pool, aToken)`, not by `aToken` alone.
10. UNIV3_UNIV4_COMPARISON.md: the "Rate-limit key structure" row for V3 only describes the per-asset key. Version 6 introduced standard aggregate-and-per-asset rate-limit keys for V3 (for AMMs in general, standardized now); the comparison table no longer reflects what V3 actually uses.

Additional inconsistencies identified in Version 7:

1. SECURITY.md: References RELAYER instead of ALLOCATOR.
2. WEETH_INTEGRATION.md: The "Funds Never Stuck in Module" section asserts that the module cannot accumulate stranded funds, but `WEETHModule.claimWithdrawal` only sweeps `ethReceived = address(this).balance` worth of WETH after wrapping; any WETH that was sent to (or accidentally transferred into) the module before the claim remains permanently stuck, since the post-deposit balance is `prevWETH + ethReceived` and only `ethReceived` is transferred to the proxy.
3. OPERATIONAL_REQUIREMENTS.md: The Withdrawal Dependencies table lists "Non-zero deposit rate limit for same vault" as a prerequisite for `withdrawERC4626` and `redeemERC4626`. The implementation calls `_tryIncreaseRateLimit` on the deposit key, which silently no-ops when the deposit key's `maxAmount` is zero.
4. `ILayerZeroFacet.transfer` NatSpec (line 82) states "Excess native fee is refunded to the caller." In `LayerZeroFacet.transfer` the OFT `send` is invoked with `refundAddress = proxy` and any residual ETH in the Controller is swept to the proxy via `_sweepETH()`; excess native fee is therefore returned to the ALMProxy, not to the caller.

Note that the above is best-effort only and that various other mismatches are likely.

Specification partially changed:

The Version 5 points are fully addressed.

Below is a list of unaddressed points from above:

- Cross-cutting#2: `IALMProxyFreezable.AllocatorRemoved` still not documented. But the finding has been mainly addressed.

Below is a list of unaddressed point from above for Version 6:

- #8 Still missing the Version 6-introduced `AccessControlUnauthorizedAccount(IFacet)`.

Below is a list of unaddressed point from above for Version 6:

- #1 SECURITY.md: Mostly resolved, but line 97 still references the removed `RELAYER` role and `removeRelayer`.

7.4 Allowance For Ethena Minter May Not Be Consumed

Informational Version 1 Acknowledged

CS-SKYDPAU-013

The integration with Ethena minter for USDe minting and burning requires external parties' (delegated signers, Ethena minter and redeemer) actions. The *relayer* (renamed to *allocator* in Version 6) can only trigger the `approve()` from the ALMProxy and expect the consecutive actions will be completed by the external parties.

In the following cases the allowance may not be fully consumed:

- The delegated signers sign orders with smaller volume which do not consume all the allowance.
- The Ethena minter or redeemer refuse to submit the order, which blocks the minting or redeeming and does not consume the allowance.
- The expected minting and burning may not be executed successfully due to the restrictions on Ethena minter such as the volume exceeds per block limit.

Consequently, the actual amount used in the interactions may be less than the amount tracked by the rate limit.

Acknowledged:

Sky acknowledged the issue and decided not to change the code.

7.5 Can Only Collect Uniswap V3 Position Fees When Removing Liquidity

Informational Version 1 Acknowledged

CS-SKYDPAU-014

The `removeLiquidity` function in `UniswapV3Lib.sol` will internally call the `NonfungiblePositionManager's collect` function to collect fees from the position. The `removeLiquidity` function does not allow being called when removing zero liquidity from the position. There exists no separate `collect()` function in `UniswapV3Lib.sol` that allows a relayer (renamed to allocator in Version 6) to only collect the fees of a given position. As a result, the only way to collect the

fees from the position is to remove a non-zero amount of liquidity by calling the `removeLiquidity()` function.

Acknowledged:

Sky acknowledged the issue and decided not to implement a fix.

7.6 CurveLib: Rounding Error Amplification

Informational | Version 1 | Acknowledged

CS-SKYDPAU-010

The CurveLib now executes the following equivalent computation to get the `MinimumAmountOut` given the configured `maxSlippage`:

```
// Makes the assumption that value in should equal value out.
uint256 valueIn             = amountIn * rates[inputIndex] / 1e18;
uint256 equivalentAmountOut = valueIn * 1e18 / rates[outputIndex];
uint256 MinimumAmountOut    = equivalentAmountOut * maxSlippages[pool] / 1e18;

require(
    minAmountOut >= MinimumAmountOut,
    "CurveLib/min-amount-not-met"
);
```

It performs three divisions, and the rounding error of the first two division results is amplified by the consecutive multiplications. Consequently, the `MinimumAmountOut` could be underestimated.

As of Version 7, the computation of the "value" of tokens is also less accurate:

```
for (uint256 i = 0; i < inputAmounts.length; ++i) {
    inputValue += _toNormalizedAmount(inputAmounts[i], rates[i]);
}
```

Now, each iteration has a rounding error. The rounding errors are accumulated. Previously, the normalization occurred only once all values were summed up.

Acknowledged:

Sky responded:

```
We opted to make the code easier to understand here,
acknowledge small rounding error got introduced.
```

7.7 Dust eETH Shares When Wrapping Into weETH

Informational | Version 1 | Acknowledged

CS-SKYDPAU-015

In `WEETHLib`, when wrapping eETH into weETH, the following steps are performed:

1. Convert eETH shares to an eETH amount using `liquidityPool.amountForShare()`, which performs the following share to underlying computation that rounds down.

```
eETHAmount = (_share * getTotalPooledEther()) / totalShares;
```

1. Call `WEETH.wrap()` with eETHAmount from the previous step, which computes the shares (rounding down) to be transferred as follows:

```
shares = (_amount * eETH.totalShares()) / totalPooledEther;
```

Consequently, not all the eETH shares may be wrapped due to rounding errors in the conversion. Consider the following examples:

```
# Example 1
Pool State:
    totalShares:        1000000000000000007
    totalPooledEther:   1000000000000000003
    ExchangeRate:       ~1.0
User State:
    userShares:         100000000000000000
    userBalanceEther:   99999999999999999
    sharesToTransfer:   99999999999999999
    dust:               1

# Example 2
Pool State:
    totalShares:      816116506552259432609369 8
    totalPooledEther: 446965807297444879060172 8
    ExchangeRate:     0.547674
User State:
    userShares:       765401968638904196136 96
    userBalanceEther: 419190772485416859646 72
    sharesToTransfer: 765401968638904196136 94
    dust:             2
```

7.8 LayerZero Lack of Global Rate Limits

Informational **Version 1** **Acknowledged**

CS-SKYDPAU-003

The LayerZero integration implements rate limits per OFT and destination pair. In contrast, the CCTP integration implements a limit per destination and a global CCTP limit (always in USDC). The LayerZero integration however lacks a notion of global limits per token and, hence, an inconsistency between the bridging rate limits for CCTP and LayerZero exists.

Acknowledged:

Sky has acknowledged the inconsistency between the bridging rate limits for CCTP and LayerZero.

7.9 Maple Manual Withdraw May Be Enabled

Informational | Version 1 | Risk Accepted

CS-SKYDPAU-016

A Maple redemption requires two steps:

1. The user submits a redemption request.
2. The privileged redeemer processes the request.

In a typical path, no more user interactions are required after step 1, and the underlying tokens will be automatically sent to the user in step 2.

However, in case manual withdrawal is enabled for the user, another call to `MaplePool.redeem()` must be initiated to fulfill the withdrawal and trigger the underlying token transfer.

Note that manual withdrawals can only be enabled by the privileged roles (pool delegator and protocol admins) of MapleWithdrawalManager with `setManualWithdrawal()`. In this case, the ALM Proxy has to explicitly call redeem (ALM Controller's `redeemERC4626()`) to finalize the redemption. And this requires a `LIMIT_4626_WITHDRAW` configured on the ALM Controller for this MaplePool. In addition, the manually withdrawable shares will be internally accounted in the MapleWithdrawalManager, hence the share balance of ALMProxy (`balanceOf()`) will not contain this.

7.10 Missing Validation of Pendle Market V3

Informational | Version 1 | Acknowledged

CS-SKYDPAU-017

Pendle is designed to be permissionless. Anyone can create an SY token and deploy PT, YT and a market for the tokens using the factory. In `PendleLib`, the rate limit checks validate that the Pendle market is a contract whitelisted by the *default admin*, but do not verify that it is a legitimate contract deployed by the PendleMarketFactoryV3 using `isValidMarket()`. Note that market validation can also be performed when configuring the rate limits.

Acknowledged:

Sky states:

```
The check can be done in spell tests that are onboarding a particular market.
On top of that onboarding of new assets is very thoroughly reviewed by multiple
parties anyway.
```

7.11 Per Pool `maxSlippage` Instead of per Operation

Informational | Version 1 | Acknowledged

CS-SKYDPAU-018

A single `maxSlippage` value is defined per pool (by the admin/governance) and is used for all operations of a given pool. This could be considered to be a design issue, since it might be necessary to have a different max slippage tolerance for a `removeLiquidity()` operation (e.g. due to impermanent loss) than for a `swap()` or `addLiquidity()` operation. Being able to define a different `maxSlippage` value per operation would allow for more flexibility and control over the slippage tolerance on a pool.

Acknowledged:

Sky acknowledged the issue and decided not to implement a fix since they will mostly be operating in stablecoin-stablecoin pools, and therefore the slippage considerations should be fairly similar.

7.12 Withdraw From Aave Can Be Blocked By LTV=0 Asset

Informational | Version 1 | Acknowledged

CS-SKYDPAU-020

When an asset is deposited under a user for the first time, the asset will be automatically configured as collateral if its LTV is non-zero and it is not in isolation mode.

In case a user has an asset enabled as collateral which has `LTV==0`, the user will not be able to withdraw any other assets that have `LTV>0`.

As a consequence, the following theoretical attack is possible:

- An attacker observed an asset that has `LTV>0` is going to be configured to `LTV==0` on Aave.
- It can supply on behalf of the ALMProxy (or send directly) a dust amount of this aToken.
- After the parameter change on Aave, the asset has `LTV==0`. The attacker successfully DoS the ALMProxy, which will not be able to withdraw the desired aToken (i.e. aUSDS, aUSDC...) from Aave.

Note that `AaveFacet.withdraw()` enforces a per `aToken` withdraw rate limit (`LIMIT_AAVE_WITHDRAW`). Provided this rate limit has been configured for the affected `aToken`, the *relayer* (renamed to *allocator* in Version 6) can withdraw the full balance of the asset with `LTV==0`, which resets the `usingAsCollateral` flag to `false` and recovers the ALMProxy from the DoS.

Acknowledged:

Sky has acknowledged this issue and stated a rate limit for the `LTV=0` asset will be added to withdraw this asset in case this attack happens.

7.13 `setOTCRechargeRate` May Flip OTC Swap Status

Informational | Version 1 | Acknowledged

CS-SKYDPAU-022

`setOTCRechargeRate()` changes the recharge rate without checking if there is an ongoing OTC swap. When called during an active swap, the new rate is applied retroactively to calculate the swap's readiness status, which can cause `isOtcSwapReady()` to flip immediately.

This occurs because `getOtcClaimWithRecharge()` uses the current `rechargeRate18` for the entire elapsed time since `sentTimestamp`:

```
otc.claimed18 + (block.timestamp - otc.sentTimestamp) * otc.rechargeRate18;
```

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 A Compromised Ethena Minter Or Redeemer May Execute A Bad Order

Note Version 1

Ethena's minter and redeemer are two crucial roles that can submit the signed orders to the Ethena minting contract. Since the delegated signers are only semi-trusted and can be malicious, the minter and redeemer are fully trusted to never submit bad orders signed by the malicious delegated signers (also described in Trust Model).

In the worst case, if Ethena's minter or redeemer are compromised, they may collude with a malicious delegated signer to execute an order with a bad quote that drains the approved USDC or USDe from ALMProxy.

8.2 ABI Changes

Note Version 1

Offchain programs / services should be aware of the abi changes (compared to the initial ALM controller):

- Mainnet Controller now has functionalities and events of new integrations.
- Following events are removed from Mainnet Controller's abi: `LayerZeroRecipientSet`, `MaxExchangeRateSet`, `MintRecipientSet`, `UniswapV4TickLimitsSet`, and OTC related events.
- Nonce field has been removed from event `CCTPTransferInitiated`.
- Events `OTCBufferSet` and `OTCRechargeRateSet` do not contain the old values any more.

As of Version 4, the second stage of the Diamond PAU v1.13 migration, is deviating from all previous versions. All relevant parties should be aware.

8.3 Aave Interprets Uint256 Max Withdrawal as Full Withdrawal

Note Version 1

The *relayers* (renamed to *allocators* in Version 6) should be aware that Aave will interpret a withdrawal with `type(uint256).max` amount as a full withdrawal of the user's balance. The *relayers* should be careful of this special behavior if they are dependent on the input amount.

8.4 Asynchronous Operations May Be Interfered With

Note Version 1

The execution of some asynchronous operations may be interfered with and unable to finalize due to another operation. For instance, the operation of `prepareUSDeMint()` will be initiated to mint USDe, which simply grants allowance of tokens to be deposited. Before the 3rd-party operation to consume the allowance, another operation, i.e. `swapCurve()`, may use up the tokens. This may cause the 3rd-party operation to fail due to insufficient token balances.

The *relayers* (renamed to *allocators* in Version 6) should be careful of the asynchronous operations and avoid the interference of different operations.

8.5 Curve Withdrawal Slippage

Note Version 1

When removing liquidity from Curve with `removeLiquidityCurve()` a balanced withdrawal is performed. Note that the performed slippage protection is not strictly necessary. Namely, assuming tokens are pegged, that is due to no negative slippage in terms of "underlying value" being possible.

As a consequence, note that the *relayer* (renamed to *allocator* in Version 6) will typically be forced to simulate the transaction to be able to provide rough values suitable to pass the check.

8.6 Maple Deposit Ignores Unrealized Losses

Note Version 1

When depositing into the MaplePool, shares are minted assuming there are no unrealized losses from the underlying loan managers, hence this deposit will bear part of the impairment immediately. In addition, withdrawals from MaplePool will bear existing unrealized loss and forfeit the potential recovery of the impairment. The *relayers* (renamed to *allocators* in Version 6) should monitor Maple's loan and unrealized loss status before deposits and withdrawals to avoid loss to the ALMProxy.

8.7 OFT Considerations

Note Version 1

Governance should be aware that certain OFTs are not supported. Below is a list of considerations to make when adding support for an OFT:

- OFTs that try to pull more than was specified are not supported due to lack of sufficient approval.
- OFTs that could try to burn (without approval) more than was specified are generally not supported. If more would be burned than specified, the rate limit accounting could be incorrect. Thus, such OFTs should not be added.
- OFTs with inherent rate limits could lead to unsuccessful operations. More specifically, the executions could revert due to OFT rate limits.
- OFTs should be ensured to follow the OFT standard correctly. Additionally, the underlying token should not be allowed to change or similar as this could lead to rate limit violations.
- The gas cost for configured `destinationEndpointId` should be carefully monitored to ensure that the hardcoded value of `200_000` is sufficient.

- Function `transferTokenLayerZero()` queries the messaging fee with `quoteSend(..., false)` prior to `send()` to prepare the fee payment by attaching required native tokens. Even though it requests the quote with `payInLzToken = false`, this does not guarantee the fee contains no lzToken. In summary, OFTs that always require part of the fee in lzToken are not supported by Diamond PAU v1.13 hence should not be used.

8.8 Permissionless Spell Execution and Rate-Limit Changes

Note **Version 7**

`RateLimits.setRateLimitData` is gated by `DEFAULT_ADMIN_ROLE`. That role is expected to be held by a governance pause/spell mechanism whose execution step is permissionless.

Both lowering and raising the rate limit create a double-spend window around the spell's execution. That is similar to the ERC-20 race-condition.

Governance should be aware that lowering/increasing rate limits requires careful consideration and should assume an allocator may engineer such double-spends and time spell execution accordingly.

8.9 Recipient Mappings Cannot Be Cleared

Note **Version 6**

The bridging / cross-chain facets store a recipient per destination identifier and reject `bytes32(0)` in the setter, with no remove function. Once a destination is offboarded (i.e. by resetting its rate limit), the configured recipient stays in storage and the public getter keeps returning the previously set value:

1. `CCTPFacet.setMintRecipient()` requires `recipient != bytes32(0)`; `getMintRecipient()` returns whatever was last set.
2. `LayerZeroFacet.setRecipient()` requires `recipient != bytes32(0)`; `getRecipient()` returns whatever was last set.
3. `CentrifugeFacet.setRecipient()` requires `recipient != bytes32(0)`; `getRecipient()` returns whatever was last set.

Off-chain consumers querying these getters for an offboarded destination receive an outdated address rather than a clear "unconfigured" signal, and the *admin* cannot disable a previously configured destination by zeroing the recipient.

8.10 Risk of Pendle Integration

Note **Version 1**

The following risks are introduced by the Pendle integration:

1. The Pendle router proxy contract forwards calls to different implementations (facets). It is controlled by its owner and upgradeable. Consequently, all funds allocated to the Pendle integration could be lost if the owner was malicious.
2. The SY token contract can have various implementations. Consequently, if it was malicious, the funds for redemption could be lost.

3. The redemption relies on the `yt.pyIndexCurrent()` computation and an honest relayer (renamed to allocator in Version 6) submitting a minAmountOut. There are no admin-defined slippages.

Markets and tokens should be properly inspected before being added to the rate limits.

8.11 Special Cases Handling

Note Version 1

The PAU can be extended by adding support for new facets. Some currently unresolvable scenarios could be resolved in the future if needed.

For example, the mint recipient for CCTP could be blacklisted. That effectively could DoS the USDC bridging. In that case, a new controller could be added that allows calling CCTP's `replaceDepositForBurn` to resolve the issue.

Ultimately, some unlikely (and intentionally unhandled) issues may arise with the existing controllers. To resolve such issues, new controllers can be added.

8.12 `ALMProxy` Exposes No `doStaticCall` Helper

Note Version 6

`ALMProxy` exposes `doCall()`, `doCallWithValue()`, and `doDelegateCall()` but no `doStaticCall()` equivalent. Facets that need to read state from an external contract through the proxy (in case the result is caller-dependent) must therefore route those reads via `doCall()`, which executes a regular `CALL` and does not constrain the callee at the EVM level.

`LayerZeroFacet.transfer()` is the clearest example. The minimum amount received and the messaging fee are both derived from queries against the allocator-supplied `oft`:

```
IALMProxy(proxy).doCall(oft, abi.encodeCall(ILayerZeroLike.quoteOFT, (sendParams)))
IALMProxy(proxy).doCall(oft, abi.encodeCall(ILayerZeroLike.quoteSend, (sendParams, false)))
```

The facet treats both calls as view-only on the OFT side. Because `doCall()` performs a regular `CALL`, a misbehaving or compromised `oft` is free to mutate its own state, trigger further external calls, or reenter the `ALMProxy` during what the facet treats as a quoting call. `doStaticCall()` could be a defense-in-depth mechanism. A `STATICCALL`-shaped entry point on the proxy would eliminate this surface at the EVM level for any facet that uses the proxy purely to read external state.

As of Version 8, the above has been handled. While no `doStaticCall()` has been introduced, the resolution is to perform `STATICCALL` to `doCall()` so that no state changes can occur in `doCall()` without reverting. Developers and reviewers should ensure any other facet that uses `doCall()` purely to read external state likewise wraps it in a `STATICCALL`.

8.13 `Controller` and `Beacon`: Governance Should Avoid Conflicting Wirings

Note Version 4

Both `Controller._setConfigAndDispatches` and `Beacon._setConfigAndDispatches` require that every `callSelector` in a new integration config is currently unwired, otherwise they revert with `CallSelectorAlreadyWired`. Several governance scenarios can produce conflicting wirings:

- **Selector migration within a single `updateIntegrations` call.** `Controller.updateIntegrations` processes the `ids` array sequentially, deleting the existing config for each id before applying the new one. If two integrations in the same call swap ownership of a selector, the order of the array determines whether the call succeeds. For example, if integration `A` previously owned selector `s` and the new config for integration `B` (listed earlier in the array) claims `s`, processing `B` reverts because `A` still owns `s`. The same update succeeds if the caller orders the array so that the integration releasing the selector is processed first.
- **Selector migration across separate governance actions.** The same conflict arises when the integrations are updated in separate transactions. If integration `A` currently owns selector `s` and governance later applies a new config for integration `B` claiming `s`, the update for `B` reverts until `A` is first removed or updated to release `s`.
- **`Controller` and `Beacon` desynchronization.** The `Beacon` is the source of truth but each `Controller` holds its own `dispatches` mapping and only updates when `updateIntegrations(ids)` is called for specific ids. An integration added to the `Beacon` is not routable on a `Controller` until that `Controller` is explicitly synced. An integration removed from the `Beacon` remains wired on the `Controller` until `removeIntegrations` is called, and a subsequent `updateIntegrations` for that stale id reverts with `IntegrationNotFound` because `getConfigs` returns a zero-facet config, so the stale wiring cannot be corrected by an update. A mutated config in the `Beacon` is likewise not reflected until the id is included in the `Controller`'s next `updateIntegrations` call.

In practice selectors are expected to be distinct across integrations and such conflicts should not arise. Governance should nonetheless be careful about execution ordering and about keeping `Controller` state synchronized with the `Beacon` to avoid reverting transactions or unexpected scenarios.

8.14 `msg.value` Refund to ALMProxy

Note Version 1

For some operations, the *relayer* (renamed to *allocator* in Version 6) has to attach `msg.value`.

That is mainly relevant for bridging with Centrifuge and LayerZero to pay for the crosschain transfer. For both (LayerZero: since Version 7), if `msg.value` is excessive, the excess value will be refunded to the ALMProxy instead of the relayer.