PUBLIC

Code Assessment

of the Grove ALM Controller Smart Contracts

December 23, 2025

Produced for

by

CHAINSECURITY

Contents

1 Executive Summary

Dear all,

Thank you for trusting us to help GroveLabs with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Grove ALM Controller according to Scope to support you in forming an opinion on their security risks.

GroveLabs offers Grove ALM Controller, a fork of Spark ALM Controller, that implements a set of on-chain components of the Grove Liquidity Layer designed to manage and control the flow of liquidity between Ethereum mainnet and L2s by leveraging Sky DSS Allocator.

This latest review covers v1.8.0 which introduces support to interact with UniswapV3, the Merkl Distributor and usage of CCTPv2 amongst some other refactoring in existing code.

The most critical subjects covered in our audit are access control, the correct integration with the external protocols. The general subjects covered are gas efficiency, documentation and composability.

During our review, several issues were uncovered in the interaction with UniswapV3. Most importantly, the `maxSlippage` check on operations limiting the relayer was found to be ineffective, see Ineffective maxSlippage check in UniswapV3Lib. Further issues uncovered included Governance Tick Bounds Not Revalidated When Adding Liquidity to Existing Position and Pool/TokenId Mismatch Allows Incorrect Rate Limit Accounting in addLiquidity().

After the intermediate report all reported issues have been resolved.

In summary, we find that the codebase provides a high 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	1
• Code Corrected	1
Medium-Severity Findings	2
• Code Corrected	2
Low-Severity Findings	13
• Code Corrected	8
• Risk Accepted	3
• Acknowledged	2

2 Assessment Overview

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

2.1 Scope

The assessment was performed on the source code files inside the Grove ALM Controller 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	11 Aug 2025	1658e2034e0149ae1f5db0692370822008417903	Initial Version
2	19 Sep 2025	487999f61a28969cb499dfdeff880745c7f4c979	Pendle Redemptions
3	10 Oct 2025	47d268522780590b17d12b6b560a622480de5847	Pendle Redemptions Fixes
4	21 Oct 2025	64e1131538e7d222608eec7e5dd6c2924f767fc6	Pendle Redemptions Release
5	27 Nov 2025	0f7e7979520b74e496a7ead632c840eb8c70889b	v1.8.0
6	19 Dec 2025	b5d47f14299f4d82b0ddeea25f303cbd3a09b6d7	v1.8.0 Fixes
7	23 Dec 2025	2c6e3d4297d5f244894d05f3dbbe47bcada34712	v1.8.0 Final

For the solidity smart contracts, the compiler version `0.8.25` was chosen and `evm_version` is set to `cancun`.

Grove ALM Controller is a fork of Spark ALM Controller. The review scope covered the differences between Grove ALM Controller's initial commit and Spark ALM Controller v1.5.0. This report also includes the risk accepted, acknowledged issues and notes from the upstream v1.5.0 audit report as these remain relevant to the code being reviewed. The following files are in scope:

```
src/
    ALMProxy.sol
    MainnetController.sol
    ForeignController.sol
    RateLimitHelpers.sol
    RateLimits.sol
    interfaces/
        IALMProxy.sol
        IRateLimits.sol
        CCTPInterfaces.sol
        CentrifugeInterfaces.sol
        ILayerZero.sol
    libraries/
        CCTPLib.sol
        CentrifugeLib.sol
```

```
        CurveLib.sol
        PSMLib.sol
deploy/
    ControllerDeploy.sol
    ControllerInstance.sol
    ForeignControllerInit.sol
    MainnetControllerInit.sol
```

In Version 2, the following files were added to scope:

```
src/
    interfaces/
        PendleInterfaces.sol
    libraries/
        PendleLib.sol
```

In Version 5, the following files were added to scope:

```
src/
    interfaces/
        MerklInterfaces.sol
        UniswapV3Interfaces.sol
    libraries/
        MerklLib.sol
        UniswapV3Lib.sol
        common/
            ERC20Lib.sol
            MathLib.sol
```

2.1.1 Excluded from scope

All other files are out of scope. In particular, all 3rd-party protocols Grove ALM Controller integrate with are out of scope and assumed to work honestly and correctly as documented, including: Aave, Ethena, Centrifuge, Morpho, Curve Stableswap-NG Pools Pendle and Uniswap.

For the curve integration, it is assumed only the Stableswap-NG Plain pools will be used. Note that the deployment script is in scope. However, governance should validate the deployment.

In addition, the inherent centralization risks of USDC are out of the scope of this review:

- USDC is deployed behind a proxy, and its implementation can be upgraded by an admin.
- CCTP relies on a set of centralized offchain signers to provide the bridging attestation.

For the Pendle integration, the integration is assumed to be with the pendle router proxy as of October 2025, this cannot account for any future changes or upgrades.

The Uniswap V3 libraries imported in `src/libraries/uniswap-v3/` (`UniV3OracleLib.sol` and `UniV3UtilsLib.sol`) are out of scope and assumed to work correctly.

Note that the deployment script is in scope. However, governance should validate the deployment.

2.2 System Overview

This system overview describes the latest received version (Version 6) of the contracts as defined in the Assessment Overview.

GroveLabs offers Grove ALM Controller, a fork of Spark ALM Controller, that implements a set of on-chain components of the Grove Liquidity Layer designed to manage and control the flow of liquidity between Ethereum mainnet and L2s by leveraging Sky DSS Allocator.

On each chain, the following contracts are deployed. All of the contracts inherit the standard AccessControl and grant the *DEFAULT_ADMIN_ROLE* role to an *admin* which can configure other roles.

ALMProxy: Entity that holds funds and interacts with external contracts (e.g. DssAllocator, PSM). Thus, it holds the required privileges to interact with other contracts. It exposes `doCall()`, `doCallWithValue()`, and `doDelegateCall()` to the *controller* role for customized executions. `receive()` is also implemented to receive native tokens.

RateLimits: Defines and enforces limits for liquidity flows. Limits can be configured by the *admin*. The rate limit will linearly grow from `lastAmount` with `slope` over the time elapsed (tracked with `lastUpdated`), and is capped `maxAmount`. Limits will be consumed or recharged by the **controller** with `triggerRateLimitDecrease()` or `triggerRateLimitIncrease()`.

Controllers: Dictates which operations an ALMProxy shall perform. Note that multiple controllers could point to the same ALMProxy. **MainnetController** and **ForeignController** are implemented that define operations in the context of respective ALMProxy. They share a similar structure but feature some different third-party integrations and operations. The *admin* can configure the parameters for 3rd-party integrations and set up other roles: *relayer* that triggers ALMProxy executions and *freezer* that can revoke a *relayer* in emergency. The `immutable` and `constant` keywords have been removed from MainnetController to reduce bytecode size, this further leads to the addition of public getters of these variables.

For more details regarding these contracts and the unchanged 3rd-party integrations, please consult the Spark ALM Review. Note the Superstate and Maple integrations have been removed.

2.2.1 Centrifuge V3

Grove ALM Controller further integrates with Centrifuge V3. In general, the deposit and redemption (including the respective cancellation) functionalities stay the same as Centrifuge V2, while the following function is added to both MainnetController and ForeignController:

transferSharesCentrifuge: The controller now allows transferring share class tokens to a cross-chain recipient address. The recipient for each destination chain (`centrifugeId`) must be explicitly configured by the *admin* using `setCentrifugeRecipient()`. The transfer amount is rate-limited based on the token and the destination chain. Once the message is processed on the destination chain (`Spoke.executeTransferShares()`), the shares are minted to the recipient.

2.2.2 Pendle

Since Version 2, Grove ALM Controller integrates Pendle redemption capabilities. The integration handles the redemption of Principal Tokens (PT) after market expiry:

redeemPendlePT: Allows the controller to redeem matured Principal Tokens from expired Pendle markets. The function requires the market to have reached expiry before redemption can occur. The redemption converts PT tokens back to their underlying yield-bearing token through Pendle's router, with the amount being rate-limited per market. The function automatically calculates the minimum expected output based on the Standardized Yield (SY) token's exchange rate, applying a small buffer to account for potential rounding errors. The redeemed assets are returned to the ALMProxy.

Since Version 3, a **relayer** specified slippage protection ensures redemptions execute as expected, assuming an honest relayer.

Since Version 4, rate limit is decreased according to the `totalTokenOutAmount` instead of the `pyAmountIn`.

2.2.3 Uniswap V3

Since Version 5, Grove ALM Controller integrates Uniswap V3 for token swaps and liquidity provision on both MainnetController and ForeignController:

swapUniswapV3: Allows the relayer to execute token swaps through whitelisted Uniswap V3 pools. The swap is rate-limited per input token and pool combination. Price movement and slippage are constrained by the admin (the Governance) through `swapMaxTickDelta` and `maxSlippage`.

addLiquidityUniswapV3: Allows the relayer to mint new Uniswap V3 positions or add liquidity to existing positions. The tick range for new positions must fall within admin configured bounds (`addLiquidityTickBounds`) for this pool. For existing positions, the provided tick range must be within the position's existing tick range. The deposited amounts for both tokens are rate-limited per token and pool.

removeLiquidityUniswapV3: Allows the relayer to decrease liquidity from existing Uniswap V3 positions belonging to the ALMProxy and collect the resulting tokens. The withdrawn amounts are rate-limited per token and pool. The function collects all owed tokens (including any accumulated fees) to the ALMProxy.

The *admin* (the governance) can configure pool parameters using `setUniswapV3PoolMaxTickDelta()`, `setUniswapV3AddLiquidityLowerTickBound()`, and `setUniswapV3AddLiquidityUpperTickBound()`. Such a configuration effectively whitelists the pool for use by the relayer.

Since Version 6, the *admin* can configure the TWAP oracle lookback period for a pool using `setUniswapV3TwapSecondsAgo()`. This parameter determines the time window used for time-weighted average price calculations in the Uniswap V3 oracle integration.

2.2.4 ERC4626 Vaults

Since Version 6, both MainnetController and ForeignController include exchange rate protection for ERC4626 vault deposits:

The *admin* can set maximum exchange rate thresholds using `setMaxExchangeRate()`, which takes a number of shares and the maximum expected assets for those shares. The exchange rate is stored with 1e36 precision. When depositing into ERC4626 vaults via `depositERC4626()`, the function validates that the actual exchange rate (shares received per asset deposited) does not exceed the configured maximum. This protects against exchange rate manipulation attacks where an attacker could inflate the share price to cause unfavorable deposits.

2.2.5 Merkl

Since Version 5, Grove ALM Controller integrates Merkl for reward distribution:

toggleOperatorMerkl: Allows the relayer to toggle an operator address for the Merkl distributor enabling or disabling that operator's ability to claim rewards on behalf of the ALMProxy. The operator can only trigger claims, funds are always sent to the user, the ALMProxy. On MainnetController, the Merkl distributor address is hardcoded to the Ethereum mainnet address. On ForeignController, the *admin* must configure the distributor address using `setMerklDistributor()`.

2.2.6 CCTP Integration

Grove ALM Controller integrates Circle CCTP for cross-chain USDC transfers. In Version 5, the integration was upgraded to use CCTP v2:

depositForBurn(): Allows the controller or relayer to send USDC to a destination chain via CCTP v2. The recipient for each destination domain must be configured by the *admin* using `setMintRecipient()`. Transfers are subject to per-token and per-destination rate limits.

2.2.7 Curve (ForeignController)

Since Version 3, ForeignController includes Curve StableswapNG integration matching the existing MainnetController functionality:

swapCurve: Allows the relayer to execute token swaps through whitelisted Curve StableswapNG pools. The swap is rate-limited per input token and pool combination. Slippage protection is enforced through the relayer-provided `minAmountOut` and the admin-configured `maxSlippage` for the pool.

addLiquidityCurve: Allows the relayer to deposit tokens into Curve pools to receive LP tokens. Deposit amounts for each token are rate-limited per token and pool combination. The function validates minimum LP tokens received against the `maxSlippage` configuration.

removeLiquidityCurve: Allows the relayer to burn LP tokens and withdraw underlying tokens from Curve pools. The withdrawn amounts for each token are rate-limited per token and pool. The function validates minimum withdrawn amounts against the `maxSlippage` configuration.

The *admin* can configure maximum slippage per pool using `setMaxSlippage()`.

2.2.8 Aave

Since Version 6, both MainnetController and ForeignController's Aave integration includes slippage protection for deposits:

depositAave: The function now requires that a `maxSlippage` value has been configured for the aToken by the *admin*. After supplying assets to the Aave pool, the function verifies that the aTokens received are at least `amount * maxSlippage / 1e18`. This protects against unfavorable exchange rates when supplying assets to Aave.

2.2.9 Asset Transfer

Since Version 3, ForeignController includes:

transferAsset: Allows the controller to transfer arbitrary assets from the ALMProxy to a destination address. The transfer amount is rate-limited based on the asset and destination combination.

2.3 Trust Model

ALMProxy: The *admin* is fully trusted, otherwise, it can set up controllers and trigger any calls with the privilege of ALMProxy. The *controller* is also trusted.

In addition, the ALMProxy requires several roles to operate, which are assumed to be set up properly by governance, for instance:

1. It requires `bud` role to swap without fee on DSS LitePSM.
2. It requires `wards` role on AllocatorVault to `draw()` and `wipe()` USDS.
3. It needs sufficient allowance from AllocatorBuffer to move minted USDS.

MainnetController and ForeignController:

1. The *admin* is fully trusted, otherwise they can DoS the controller, or steal the bridged money on the destination domain by changing the mint recipient.
2. The *relayer* is semi-trusted, and they can only change the liquidity allocation in the worst case. The *freezer* is also semi-trusted which can temporarily DoS the controller in the worst case.

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

RateLimits: The *admin* is fully trusted to configure the limit data and *controller* correctly.

The 3rd-party integration requires an extended trust model:

- It is assumed Grove ALM Controller will not interact with weird ERC-20 (rebasing / low decimals / ...) and ERC-4626 vault (low token decimals / without share inflation protection / ...). Otherwise, for instance, in case an ERC-4626 has low decimals, a *relayer* may amplify the loss due to rounding errors in shares conversion with many calls for ALMProxy on L2s.
- Grove ALM Controller is subject to the inherent risks of these protocols (i.e. risks of upgradeability, RWAs, governance ...) and generally the third party protocols receiving funds are assumed to be non-malicious.

For the shared integration with: Arbitrary ERC-4626, Aave and Aave-like protocols, Ethena, Maple, and Curve StableswapNG pools, please consult the Trust Model of Spark ALM Review. Note that the Morpho integration has been removed in Version 6.

Centrifuge V3: *Centrifuge is assumed to be the only ERC-7540 integrated.* The privileged roles (wards) are fully trusted, in particular:

- The wards of the ERC-7540 vaults are fully trusted, otherwise they can DoS the system by changing the manager or asyncRedeemManager contract.
- The wards of the asyncRedeemManager is fully trusted, otherwise they may 1) stop fulfilling deposits, redemptions, or cancellation requests; 2) fulfilling the requests with bad conversion rate and incur loss to the users.

Pendle: Pendle contracts (Router, markets, and SY, PT, YT tokens) are trusted to function correctly. The owner of the router is fully trusted, while PT and YT contracts have no owner. The SY contract implementation and any privileged roles on it are fully trusted. For more details, see Risk of Pendle Integration. The controller handles redemptions only; PT tokens arrive at the ALM proxy separately (e.g., transferred after having been purchased through a brokerage service). This may be done with funds of the ALMProxy and a deposit using `transferAsset()`.

Further, shares transfer related infrastructures (Gateway, Spoke, Hub...) and the privileged roles are expected to work correctly and honestly. Otherwise, the shares in flight may be stuck due to censorship.

Uniswap V3: Uniswap V3 pool contracts and the position manager are trusted to function correctly. The controller relies on pool state (current tick, token addresses, fee tier) for swap execution and liquidity operations. The Relayer can only execute operations on pools that have been configured by the admin (the governance). We expect that only legitimate pool addresses are whitelisted. Since Version 6, the TWAP oracle functionality relies on the pool's observation cardinality being sufficient for the configured `twapSecondsAgo` period.

Merkl: The Merkl distributor contract is trusted. On ForeignController, the *admin* must correctly configure the distributor address. Operators toggled via `toggleOperatorMerkl` gain the ability to claim rewards on behalf of the ALMProxy.

CCTP v2: Circle's burn and mint system and its attestation service are trusted to process transfers correctly.

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	0
Low-Severity Findings	5

5.1 Centrifuge Conversion Rate May Change Between Request Submission and Execution

Correctness Low Version 1 Risk Accepted

CS-GRVALM-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 conversion rate.

However, the conversion rate 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.

5.2 Centrifuge Deposit / Redemption Can Be DoSed by Cancellation

Security Low Version 1 Risk Accepted

CS-GRVALM-003

The *relayers* can request to cancel pending deposits / redemptions on Centrifuge and claim them later once being 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:

GroveLabs is aware of the risk.

5.3 LayerZero Approvals

Correctness | Low | Version 1 | Acknowledged

CS-GRVALM-004

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.

Acknowledged:

GroveLabs has acknowledged that dangling approvals may still exist.

5.4 Relayer Can DoS SUSDE Unstaking

Security | Low | Version 1 | Risk Accepted

CS-GRVALM-005

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) reset the `cooldownEnd` to be `cooldownDuration` from current `block.timestamp`. Note the 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 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: GroveLabs was aware of this issue. In addition, in case a malicious *relayer* DoSed the sUSDe `unstake()`, the *freezer* will revoke the *RELAYER* role from the malicious *relayer*.

5.5 Revoking Unused Approval

Design | Low | Version 1 | Acknowledged

CS-GRVALM-006

The *freezer* can remove *relayers* 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:

GroveLabs 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	1
Medium-Severity Findings	2
Low-Severity Findings	8
Informational Findings	3

- Ineffective maxSlippage Check in UniswapV3Lib Code Corrected

- Governance Tick Bounds Not Revalidated When Adding Liquidity to Existing Position Code Corrected
- Pool/TokenId Mismatch Allows Incorrect Rate Limit Accounting in addLiquidity() Code Corrected

- Relayer Can Bypass Min Amount Validation When Adding to Existing Position Code Corrected
- Misleading Relayer Tick Parameter Validation When Adding to Existing Position Code Corrected
- Missing UniswapV3 Tick Spacing Check Code Corrected
- Missing maxSlippage Validation in Setters Code Corrected
- UniswapV3Lib: Swap May Not Consume All Input Tokens Code Corrected
- Disabled Slippage Protection for Pendle Redemptions Code Corrected
- Maple Redemption Can Be DoSed Code Corrected
- Over-reduced Limit in Maple Redemption Code Corrected

- Incorrect Comment Above UniswapV3Lib.swap() Code Corrected
- Unused IERC20Metadata Import Code Corrected
- Maple Manual Withdraw May Be Enabled Code Corrected

6.1 Ineffective maxSlippage Check in UniswapV3Lib

Design High Version 5 Code Corrected

CS-GRVALM-027

This issue was reported by Client independently at the start of the review.

The admin role (Governance) can set a `maxSlippage` parameter per pool. This is intended to protect against a compromised relayer submitting inappropriate minimum parameters when interacting with a pool in a manipulated state where the assets are not at market price.

Uniswap V3 library functions attempt to mitigate this by bounding the relayer provided minimum with the admin set `maxSlippage`.

For example in the `swap()` function the check is:

```
require(params.minAmountOut >= (endingBalance - startingBalance) * params.maxSlippage / 1e18 ,
"UniswapV3Lib/min-amount-not-met");
```

However, this check only ensures that the minimum token amount out is within a percentage of the actual swap result, not that the value received is within the tolerated slippage of the value sent. Hence this protection is ineffective / the relayer can set `minAmountOut` as required.

Similarly this issue applies in `addLiquidity()` and `removeLiquidity()`. Here a slippage check for both tokens, and their amount of change, is implemented. In case this is replaced by a check against the change in "value", this would have to be done over the sum of both token changes, instead of per token.

Code corrected:

In Version 5, TWAP oracle-based price validation was introduced to address the ineffective slippage protection:

For swap: The swap function now uses a TWAP oracle to determine the price limit tick, replacing the use of the current pool tick. The admin configures a `twapSecondsAgo` parameter per pool, and the swap's price limit is calculated based on the TWAP tick ± the admin-configured `maxTickDelta`. This provides protection against price manipulation.

For addLiquidity: A validation function `_validateAddLiquidityMinAmounts()` was added that validates the relayer provided minimum amounts against expected amounts calculated from the TWAP price and the `maxSlippage` parameter. This ensures minimums are reasonable given the fair market price.

For removeLiquidity: No TWAP validation was added. GroveLabs is aware that this conflicts with the stated trust model that the relayer shouldn't be able to extract value from the system. However, GroveLabs chose to accept this risk, prioritizing the ability to quickly exit pools in adverse conditions over protection against a malicious relayer manipulating prices during removal. GroveLabs assessed that the likelihood of needing emergency exits is higher than the risk of malicious relayer abuse through `removeLiquidity()`. This pattern of fewer guardrails on removal is consistent with other Grove ALM Controller components such as Curve.

The existing slippage checks remained in the code, but since they were ineffective these were removed in Version 6. Except for the slippage check in `removeLiquidity()`, which still exists.

6.2 Governance Tick Bounds Not Revalidated When Adding Liquidity to Existing Position

Correctness | Medium | Version 5 | Code Corrected

CS-GRVALM-024

The admin (Governance) can set tick bounds per pool to restrict the price ranges for liquidity positions. When a relayer calls `addLiquidity()` to add liquidity to an already existing position, the `_addLiquidityToExistingPosition()` function does not revalidate that the position's ticks are within the currently set governance tick bounds.

The position's ticks may be outside the current governance bounds because:

1. The governance tick bounds were updated after the position was created
2. The position was transferred from an external source (bypassing initial validation)

The only tick related validation performed is checking that the relayer-provided tick parameters are within the position's existing tick range:

This allows liquidity to be added to positions that may be outside the currently set tick bounds.

Code corrected:

The `_addLiquidityToExistingPosition()` function now validates that the position's ticks are within the currently set governance tick bounds.

6.3 Pool/TokenId Mismatch Allows Incorrect Rate Limit Accounting in addLiquidity()

Correctness | Medium | Version 5 | Code Corrected

CS-GRVALM-019

When adding liquidity to an existing Uniswap V3 position, the `addLiquidity` function of UniswapV3Lib.sol does not validate that the provided `tokenId` belongs to the specified `pool` address. This allows a malicious or compromised relayer to add liquidity to a `tokenId` of a different pool than the one used for rate limit accounting by providing mismatched parameters.

The `addLiquidity()` function performs rate limit accounting based on the `context.pool` parameter:

```
IUniswapV3PoolLike pool = IUniswapV3PoolLike(context.pool);

address token0 = pool.token0();
address token1 = pool.token1();

// ... approvals and liquidity addition ...

// Rate limits updated for the provided pool
context.rateLimits.triggerRateLimitDecrease(
    RateLimitHelpers.makeAssetDestinationKey(context.rateLimitId, token0, address(pool)),
    amount0
);
```

However, `_addLiquidityToExistingPosition()` only validates ownership of the `tokenId`, not that the `pool` parameter matches the pool of the `tokenId`. Liquidity is added to the pool of the `tokenId` regardless of the `context.pool`.

A malicious or compromised relayer can:

1. Provide a whitelisted `context.pool` (e.g., USDC/WETH)
2. Provide a `tokenId` from a different pool (e.g., USDC/WBTC position)
3. Supply liquidity in the shared token only (USDC)
4. Rate limits are decremented for the USDC/WETH pool
5. Actual liquidity is added to the USDC/WBTC position

UniswapV3 has multiple pools per token pair (one per fee tier), so matching tokens alone does not guarantee the same pool.

Note that `removeLiquidity()` will not work for this `tokenId` because it validates that the `tokenId` belongs to the specified pool (`_validateRemoveLiquidityParams()`).

Code corrected:

The `_addLiquidityToExistingPosition()` function now validates that the `token0`, `token1` and `fee` of the provided `context.pool` match the values of the position with `tokenId`.

6.4 Relayer Can Bypass Min Amount Validation When Adding to Existing Position

Security Low Version 6 Code Corrected

CS-GRVALM-025

The `addLiquidity()` function validates `params.min.amount0` and `params.min.amount1` against the current TWAP by calling `_validateAddLiquidityMinAmounts()`, which uses the provided `params.tick.lower` and `params.tick.upper` to calculate expected amounts.

When adding liquidity to an existing position, the position's own `tickLower` and `tickUpper` are used for the actual liquidity addition, while the relayer provided `params.tick.lower` and `params.tick.upper` are used in the min amount validation. There is no connection to the position's ticks which are validated, and hence the relayer may freely choose them to manipulate the min amount validation.

A malicious/compromised relayer can exploit this by providing arbitrary `params.tick.lower` and `params.tick.upper` values to bypass the min amount checks while the actual liquidity is added at the position's existing tick range.

Code corrected:

The code now enforces that the relayer passes the position's current ticks. Since the position's ticks are validated against the governance bounds, the ticks used for the minAmount validation are now also properly validated.

6.5 Misleading Relayer Tick Parameter Validation When Adding to Existing Position

Correctness Low Version 5 Code Corrected

CS-GRVALM-028

When a relayer adds liquidity to an existing position, the `_addLiquidityToExistingPosition()` function validates the relayer provided tick parameters against the position's current ticks:

```
(, , , int24 tickLower, int24 tickUpper, ) = _fetchPositionData(params.tokenId, params.positionManager);
require(params.tick.lower >= tickLower, "UniswapV3Lib/invalid-tick-lower");
require(params.tick.upper <= tickUpper, "UniswapV3Lib/invalid-tick-upper");
```

However, these relayer provided `params.tick.lower` and `params.tick.upper` values are not actually used when adding liquidity. The `increaseLiquidity` call always adds liquidity at the position's existing ticks (`tickLower` and `tickUpper`).

The relayer tick parameters are only meaningful in the `_mintLiquidity()` case where they define the new position's tick range. This validation is misleading as it suggests the relayer can control where liquidity is added, when liquidity is always added at the position's current tick range.

Code corrected:

In Version 6, the `params.tick.lower` and `params.tick.upper` are enforced to equal the position's corresponding ticks (`tickLower` and `tickUpper`).

6.6 Missing UniswapV3 Tick Spacing Check

Design | Low | Version 5 | Code Corrected

CS-GRVALM-021

The `_mintLiquidity()` function in the `UniswapV3Lib.sol` file does not check that both the lower and upper tick are valid ticks when taking into account the tick spacing of the specific Uniswap V3 pool.

When trying to mint a Uniswap V3 position with a tick that is not a multiple of the tick spacing the Uniswap V3 code will revert **without** returning any error message:

In case a transaction reverts due to the above check, it will be difficult to figure out the reason for the transaction failure. By adding an explicit check inside `_mintLiquidity()` that checks for valid lower and upper tick values this error can be caught early and with a descriptive error message.

Code corrected:

The `_mintLiquidity()` function now validates that both the lower and upper tick are valid ticks when taking into account the tick spacing of the specific Uniswap V3 pool.

6.7 Missing `maxSlippage` Validation in Setters

Design | Low | Version 5 | Code Corrected

CS-GRVALM-020

The `setMaxSlippage()` function that was added in the `ForeignController` (and already existed in the `MainnetController`) does not check that the provided `maxSlippage` value is at most 100%. The `maxSlippage` value has `1e18` precision, meaning `1e18` stands for 100%.

Unlike the above function, the other setter functions in both controllers (that set min and max values for UniswapV3 interactions), do contain validation to ensure the provided value is within sensible bounds.

Code corrected:

The `setMaxSlippage()` function in both the `ForeignController` and `MainnetController` now validates that the provided `maxSlippage` value is at most `1e18` (100%).

6.8 UniswapV3Lib: Swap May Not Consume All Input Tokens

Design | Low | Version 5 | Code Corrected

CS-GRVALM-026

The `swap()` function may not consume all input tokens, but the rate limiter is decreased for the full amount and approval is given for the full amount, resulting in potentially dangling approval.

The swap function uses Uniswap V3's `exactInputSingle` with a non-zero `sqrtPriceLimitX96` parameter. According to Uniswap's documentation, this means the swap may consume less than the specified `amountIn`:

> WARNING: Passing in a non-zero `sqrtPriceLimitX96` can mean that less tokens than the amount specified by `amountIn` are swapped. Any contract that uses a non-zero `sqrtPriceLimitX96` parameter will need to refund any unswapped tokens.

The implementation decreases the rate limiter by the full `amountIn` before executing the swap, then approves the full amount to the router. If the swap stops early due to the price limit being reached, unswapped tokens remain in the proxy contract with dangling approval to the router, while the rate limiter has been decreased by more than the actual amount swapped.

Code corrected:

The `swap()` function now checks the actual amount swapped (difference between before and after balance of `tokenIn`) and decreases the rate limiter by this amount instead of the full `params.amountIn`. Additionally, the router's allowance is now explicitly reset to zero after the swap to clear any remaining dust.

6.9 Disabled Slippage Protection for Pendle Redemptions

Design | Low | Version 2 | Code Corrected

CS-GRVALM-001

The call to the Pendle router includes a `TokenOutput` parameter with a `minTokenOut` amount for slippage protection. However, the `PendleLib` implementation calculates this value using the same exchange rate that will be used during the redemption itself:

```
// expected to receive full amount, but the buffer is subtracted
// to avoid reverts due to potential rounding errors
uint256 minTokenOut = params.pyAmountIn * 1e18 / ISY(sy).exchangeRate() - 5;
```

This approach effectively disables slippage protection. The `minTokenOut` is derived from the SY token's current exchange rate, which is the same rate that the Pendle redemption process will use to calculate the actual output. Therefore, this value cannot protect against unexpected price movements or manipulations.

Slippage protection is intended to allow the caller to specify a minimum acceptable output amount. Other similar functions in the ALMController, such as the Curve operations, accept the minimum amount as a caller provided parameter.

While the trust model states that "the *relayer* is semi-trusted and they can only change the liquidity allocation in the worst case," making the relayer responsible for passing slippage protection parameters is not ideal. Nevertheless, slippage protection is currently disabled, whereas other functions (e.g. the functions to interact with Curve) already require the relayer to specify a min token amount.

Code corrected:

The `PendleLib` implementation now enforces its own slippage protection based on pre/post token balances and the `minAmountOut` provided by the caller (`Relayer`).

6.10 Maple Redemption Can Be DoSed

Security | Low | Version 1 | Code Corrected

CS-GRVALM-014

Maple redemption can be DoSed by a compromised *relayer* 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.

Code corrected:

In Version 2 maple redemption functionality has been removed.

6.11 Over-reduced Limit in Maple Redemption

Correctness | Low | Version 1 | Code Corrected

CS-GRVALM-012

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.

Code corrected:

In Version 2 maple redemption functionality has been removed.

6.12 Incorrect Comment Above `UniswapV3Lib.swap()`

Informational | Version 5 | Code Corrected

CS-GRVALM-022

The comment above the `swap()` function in the `UniswapV3Lib.sol` file is incorrect as it should state that the rate limit is decreased by `tokenIn` instead of `token1`.

Code corrected:

The comment above `swap()` in the `UniswapV3Lib.sol` file now correctly states that the rate limit is decreased by `tokenIn` instead of `token1`.

6.13 Unused `IERC20Metadata` Import

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

CS-GRVALM-023

The UniswapV3Lib.sol file imports `IERC20Metadata` but never uses it.

Code corrected:

The unused `IERC20Metadata` import has been removed from `UniswapV3Lib.sol`.

6.14 Maple Manual Withdraw May Be Enabled

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

CS-GRVALM-013

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.

Code corrected:

In **Version 2** maple redemption functionality has been removed.

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 Can Only Collect Uniswap V3 Position Fees When Removing Liquidity

Informational **Version 5** **Acknowledged**

CS-GRVALM-015

The `removeLiquidity` function in `UniswapV3.Lib.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 `UniswapV3.Lib.sol` that allows a relayer 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:

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

7.2 Curve Pool Interface Compatibility With Optional Parameter

Informational **Version 5** **Acknowledged**

CS-GRVALM-016

The code of the CurveLib was updated to pass an additional boolean parameter (`false`) when calling `remove_liquidity()` on Curve pools to prevent claiming admin fees during liquidity removal:

```
withdrawnTokens = abi.decode(
    params.proxy.doCall(
        params.pool,
        abi.encodeCall(
            curvePool.remove_liquidity,
            (params.lpBurnAmount, params.minWithdrawAmounts, address(params.proxy), false)
        )
    ),
    (uint256[])
);
```

For stableswap pools, not claiming admin fees is safer as this omits any potential issues that may arise in the admin fee claiming hook.

However, not all Curve pools support this optional parameter. Curve pools are implemented in Vyper, which supports optional function parameters. For external functions with default arguments like `def my_function(x: uint256, b: uint256 = 1)`, the Vyper compiler generates `N+1` overloaded function selectors based on `N` default arguments. Consequently, a function may have multiple ABI signatures depending on whether optional parameters are included.

While the currently intended stableswap pools support both the 3-parameter and 4-parameter versions of `remove_liquidity`, other Curve pool types (such as 2crypto pools) may not support the 4-parameter interface. For such pools, the call will revert when attempting to use the additional boolean flag.

Before interacting with a Curve pool using the 4-parameter `remove_liquidity` interface, it's essential to verify that the specific pool implementation supports this optional parameter. Otherwise, the transaction will fail.

Acknowledged:

GroveLabs acknowledged the issue and decided not to implement a fix as they do not plan to use other pool types that do not support this optional parameter.

7.3 Per Pool `maxSlippage` Instead of per Operation

Informational **Version 5** **Acknowledged**

CS-GRVALM-017

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:

GroveLabs 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.

7.4 Use UniversalRouter Instead of SwapRouter02

Informational **Version 5** **Acknowledged**

CS-GRVALM-018

GroveLabs indicated they are using Uniswap's `SwapRouter02` instead of `UniversalRouter` since the latter does not return the `amountOut`.

```
uint256 startingBalance = IERC20(cache.tokenOut).balanceOf(address(context.proxy));
uint256 amountOut = _callSwap(context, params, cache);
uint256 endingBalance = IERC20(cache.tokenOut).balanceOf(address(context.proxy));
```

In the swap function the returned `amountOut` is used to update the rate limit. However, the before and after balance delta of the `tokenOut` is calculated to check the validity of the relayer-provided `minAmountOut` value. This delta indicates the actual `amountOut` and therefore using this value instead of the `SwapRouter02` returned value is more accurate. This would allow using the modern, and Uniswap-recommended, `UniversalRouter` instead of the much older `SwapRouter02`.

Acknowledged:

GroveLabs acknowledged this and decided not to change the code.

7.5 Missing Validation of Pendle Market V3

Informational | Version 2 | Acknowledged

CS-GRVALM-007

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:

GroveLabs 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.
```

7.6 Allowance For Ethena Minter May Not Be Consumed

Informational | Version 1 | Acknowledged

CS-GRVALM-008

The integration with Ethena minter for USDe minting and burning requires external parties' (delegated signers, Ethena minter and redeemer) actions. The *relayer* 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 refuses 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:

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

7.7 Inconsistent Bridging Rate Limits

Informational | Version 1 | Acknowledged

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:

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

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

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

CS-GRVALM-010

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 has `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 there is no rate limit and token restriction on function `withdrawAave()`. The *relayer* 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:

GroveLabs 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.

7.9 `msg.value` Validation in `transferTokenLayerZero`

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

CS-GRVALM-011

The relayer 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.

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.

In the worst case if Ethena' 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 from ALMProxy.

8.2 Aave Interprets Uint256 Max Withdrawal as Full Withdrawal

Note Version 1

The *relayers* 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.

8.3 Asynchronous Operations May Be Interfered

Note Version 1

Some asynchronous operations may be interrupted by another operation and fail to finalize. 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* should be careful of the asynchronous operations and avoid the interference of different operations.

8.4 Avoid Morpho Deposit Into Market With Bad Debt

Note Version 1

Upon a deposit into MetaMorpho vault, shares will be calculated based on the aggregated expected balance over all the markets in the `withdrawQueue`. In case there is unrealized bad debt in any of the underlying markets, the new deposits will bear this impairment. The *relayers* should monitor the markets conditions and not deposit or reallocate into markets with unrealized bad debt.

8.5 Centrifuge Shares Transfer Refund to ALMProxy

Note Version 1

When transferring Centrifuge shares to another chain with `transferSharesCentrifuge()`, the *relayer* has to attach `msg.value` to pay for the transfer fuel. Note that in case the fuel is excessive, it will be refunded to the ALMProxy instead of the *relayer*.

8.6 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* will typically be forced to simulate the transaction to be able to provide rough values suitable to pass the check.

8.7 Inconsistent Swap Rate Limit Decrease for Curve

Note Version 1

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 GroveLabs.

8.8 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* should monitor the Maple's loan and unrealized loss status before deposits and withdrawals to avoid loss to the ALMProxy.

Note this no longer applies in Version 2, as Maple redemption functionality was removed.

8.9 MorphoAllocations updateWithdrawQueue Subject to Front Running

Note Version 1

In Morpho vaults, any user can supply on behalf of the vault. Since `updateWithdrawQueue()` requires the market to be empty, malicious actors can front-run this call, blocking its intended execution. This is a known issue documented in Morpho's documentation. The recommended workaround is for the allocator to bundle a reallocation that withdraws the maximum from the affected market alongside the updateWithdrawQueue call.

The Grove ALM Controller provides separate `updateWithdrawQueue()` and `reallocate()` functions. Although there's no bundled variant that combines them atomically, the expectation is that including both operations within a single transaction will mitigate the frontrunning risk.

8.10 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 it was specified are not supported due to lack of sufficient approval.
- OFTs could try to burn (without approval) more than it 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.

8.11 OFTs With Mandatory Fee in lzToken Are Not Supported

Note Version 1

Function `transferTokenLayerZero()` queries the fees with `quoteOFT()` prior to `send()` to prepare the fee payment by attaching required native tokens. Even though it quotes OFT with flag `_payInLzToken=false`, it does not guarantee the fee contains 0 lzToken. Hence, in case part of the fee must be paid in lzToken, `send()` will fail due to insufficient approval of lzToken.

In summary, OFTs that always require part of the fee in lzToken are not supported by Grove ALM Controller hence should not be used.

8.12 Risk of Pendle Integration

Note Version 2

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 fully relies on the `exchangeRate()` computation in the SY token, which may not be accurate:
   - The YT may cache and use the same index per block. If the index could grow during a block, the cached value for computation would be stale.
   - In the YT token, the index can only increase. If the actual `exchangeRate()` decreases, it will use the last stale index for computation.

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

Since Version 3, a relayer specified slippage protection has been added to ensure redemptions execute as expected, assuming an honest relayer.

Since Version 4, the `minTokenOut` is computed with `yt.pyIndexCurrent()` instead of `sy.exchangeRate()`, aligned with the computation in the redemption flow. Note that this affects the value passed in the call to Pendle. The controller's own slippage check introduced in Version 3 is unaffected.

8.13 Special Cases Handling

Note Version 1

The ALM's functionality can be extended by allowing new controllers. Some currently unresolvable scenarios, could be resolved in the future if needed. For example:

1. Assume it is desired that for an L2, all funds are bridged back to mainnet. However, in case the PSM3 never holds sufficient USDC to bridge back to L1, funds will remain on L2. As a result, another controller could be whitelisted that initiates redeeming the PSM shares against the other two assets to then bridge them back to mainnet through the respective bridges.
2. 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.