certora

Security Assessment Report

GROVE

Grove ALM

December 2025

Prepared for Grove Team

Table of content

Project Summary

Project Scope

Project Name	Repository (link)	Commit Hash	Platform
Grove ALM	https://github.com/grove-labs/grove-alm-controller	0f7e797 (initial) 2c6e3d4 (fix)	Solidity
	https://github.com/grove-labs/xchain-helpers	03390e2 (initial) 53f6ced (fix)	
	https://github.com/grove-labs/grove-gov-relay	1dbe73a (initial) 262e637 (fix)	

Project Overview

This document describes the manual code review findings of **Grove ALM**. The following contract list is included in our scope:

`Grove-ALM-Controller`
`src/MainnetController.sol`
`src/ForeignController.sol`
`src/libraries/*`

`Xchain-helpers`
`src/forwarders/CCTPv2Forwarder.sol`
`src/forwarders/LZForwarder.sol`
`src/receivers/CCTPv2Receiver.sol`
`src/receivers/LZReceiver.sol`
`Various minor refactorings`

`Gov-relay`
`deploy/Deploy.sol`
`script/Deploy.s.sol`

`Various minor refactorings`

The work was undertaken from **November 24, 2025,** to **December 5, 2025**. During this time, Certora's security researchers performed a manual audit of all the Solidity contracts and discovered several bugs in the codebase, which are summarized in the subsequent section.

Protocol Overview

Grove ALM is an Asset Liquidity Management protocol within the Sky ecosystem providing rate-limited, cross-chain liquidity management for RWA and credit operations. The last update added UniswapV3 and CCTPv2 integration and updated LayerZero, Curve and Pendle functionality.

Assessment Methodology

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

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

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

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

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

Threat and Security Overview

System Overview

The fundamental objective of Grove is liquidity provisionment across EVM chains and Defi protocols. To this end, all funds are stored in an immutable ALMProxy on each chain, while all execution logic resides in upgradeable Mainnet and Foreign Controller contracts. All operations are triggered by a permissionless off-chain relayer.

Core Components

- ALMProxy: immutable vault that holds 100% of protocol assets on each chain.
- Controllers (Mainnet + Foreign): upgradeable contracts containing swap, LP, redemption, and bridging logic.
- RateLimit trees: governance-configured leaky-bucket limits per (asset, destination, operation type).
- Relayer: fully permissionless, off-chain actor that builds and submits every transaction.

Fundamental Security Requirement

No actor, regardless of privileges or trust level, may move funds to a non-whitelisted destination or extract value from the system. All asset movements must stay strictly within governance-defined destinations and rate-limited amounts.

Core Threat Model

The relayer is explicitly modeled as untrusted and potentially malicious. As such, a malicious relayer can call any public function on the Controllers with any permitted parameters.
The key questions the threat model asks are:

1. Can the relayer send protocol funds to an arbitrary (non-whitelisted) address?
2. Can the relayer exploit any integration (Uniswap V3, Curve, Pendle, ERC4626 vaults, CCTP, LayerZero, etc.) to make the ALMProxy receive materially fewer assets than governance expects, thereby extracting value within rate-limit windows?

3. These two vectors represent the entire relevant attack surface under the untrusted-relayer model. All security analysis and review scope flow directly from them.

Functions Overview

UniswapV3

- swapUniswapV3
  - **Destination Protection:** the recipient is hardcoded as address(proxy) in the swap params, which means funds cannot be redirected.
  - **Value Protection:**
    - Rate limit on input asset
    - Slippage check: require(params.minAmountOut >= (endingBalance - startingBalance) * params.maxSlippage / 1e18)
    - Tick delta check: require(params.tickDelta <= params.poolParams.swapMaxTickDelta)
  - **Issue:**
    - As described in C-01, the slippage and tick check can be bypassed due to the use of spot prices, which can be easily manipulated.
- addLiquidityUniswapV3
  - **Destination Protection:** the recipient is hardcoded as address(proxy) in mint params, which means that the NFT position is minted only to the proxy.
  - **Value Protection:**
    - Rate limit on both tokens deposited
    - Dual slippage: require(params.min.amount0 >= balanceDiff0 * params.maxSlippage / 1e18)
    - Tick bounds: require(params.tick.lower >= params.tickBounds.lower)
  - **Issue:**
    - The same issue described in swapUniswapV3 applies here.
- removeLiquidityUniswapV3
  - **Destination Protection:** the recipient is hardcoded in the collect call, which means that the collected tokens can only go to the proxy.
  - **Value Protection:**
    - Ownership check: require(params.positionManager.ownerOf(params.tokenId) == address(proxy))

        - Dual slippage: require(params.min.amount0 >= collected0 * params.maxSlippage / 1e18)
        - Rate limit on withdrawn tokens
    - **Issue:**
        - The same issue described in swapUniswapV3 applies here.

Curve

- swapCurve
    - **Destination Protection:**
        - The receiver is hardcoded as address(proxy) in the exchange call, which means the output cannot be redirected
    - **Value Protection:**
        - Rate limit on USD value: params.amountIn * rates[inputIndex] / 1e18
        - Slippage: require(params.minAmountOut >= amountIn * rates[in] * maxSlippage / rates[out] / 1e18)
    - **Issue:**
        - As described in M-01, the stored_rates() are not live prices, which means an exploitation vector opens if the 1:1 exchange rate is broken. This is unlikely due to the use of Stableswap-NG pools with a very high amplitude, but not impossible.
- addLiquidityCurve
    - **Destination Protection:**
        - The receiver is hardcoded as address(proxy) in add_liquidity, which means the LP tokens are minted to only the proxy.
    - **Value Protection:**
        - Rate limit on total USD value deposited
        - Slippage: require(minLpAmount >= valueDeposited * maxSlippage / virtualPrice)
        - Swap accounting for implicit swaps during add
    - **Issue:**
        - The same issue described in swapCurve applies here.
- removeLiquidityCurve
    - **Destination Protection:**
        - The receiver is hardcoded as address(proxy) in remove_liquidity, which means the withdraw tokens go to only the proxy.

- **Value Protection:**
    - Slippage: require(valueMinWithdrawn >= lpBurn * virtualPrice * maxSlippage / 1e36)
    - Rate limit on actual withdrawn USD value
- **Issue:**
    - The same issue described in swapCurve applies here.

Pendle

- redeemPendlePT
    - **Destination Protection:**
        - The receiver is hardcoded as address(proxy) in redeemPyToToken, which means redeemed assets go to proxy only
    - **Value Protection:**
        - Maturity check: require(params.pendleMarket.isExpired())
        - Double Balance check: minTokenOut & minAmountOut
        - Rate limit on actual output
    - **Issue:**
        - As described in L-02, the logic assumes that SY and Asset token always have a 1:1 exchange rate. This is not always the case.

ERC4626

- depositERC4626 / withdrawERC4626 / redeemERC4626
    - **Destination Protection:**
        - The receiver/owner is hardcoded as address(proxy) in all vault calls, which means the shares or assets cannot be redirected.
    - **Value Protection:**
        - Rate limit on asset amounts
    - **Issue:**
        - As described in a previous report which will be merged with this one in the final version, there is the assumption that the vault is impervious to inflation attacks. When this does not hold (new vault or drained vault) there is a clear attack vector for a malicious relayer to extract value.

Aave

- depositAave / withdrawAave
    - **Destination Protection:**

        - onBehalfOf/to is hardcoded as address(proxy), which means the assets can only go to the proxy
    - **Value Protection:**
        - Rate limit on underlying amounts.
    - **Assumption:**
        - The integrating partner Aave is trusted to maintain a correct design and ensure the fairness of the aToken:underlying exchange rate.

Centrifuge

- transferSharesCentrifuge
    - **Destination Protection:**
        - The recipient from centrifugeRecipients[destinationCentrifugeId] mapping is set by Sky governance, so the relayer cannot change it.
    - **Value Protection:**
        - Rate limit: keccak256(abi.encode(LIMIT_CENTRIFUGE_TRANSFER, token, destinationCentrifugeId))
        - Validation: require(recipient != 0)
    - **Assumption:**
        - The integrating partner Centrifuge spoke contract is trusted to behave correctly.

The main attack surfaces of the threat model are covered to a high degree throughout the integrating functions. We could not identify any case where the destination protection was insufficient. In regards to the value protection, we identified one Critical issue in the Uniswap integration and a few potentially impactful issues based upon incomplete assumptions.

On a general note, we would like to draw your attention to the long-term risks of assumptions which are almost always correct, but can never be completely guaranteed. Examples of this are:

- *We only use Pendle Markets with standard SY tokens*
- *An ERC4626 Vault will always be resilient to inflation attacks*
- *Curve Pools have sufficient amplitude to maintain 1:1 rate*
- *Integrating protocol X works in a fixed fashion*
- *Etc..*

Since these assumptions will be true in 99.999% of cases, they are often internalized as always being true. This is problematic as over time the extremely rare case will eventually occur and without protections in place, the protocol could suffer massive damages.

It is our recommendation that defensive programming is set into place whenever an assumption cannot be fully guaranteed.

Findings Summary

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

Severity	Discovered	Confirmed	Fixed
Critical	-	-	-
High	1	1	1
Medium	2	2	1
Low	4	4	1
Informational	2	2	1
Total	9	9	4

Severity Matrix

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

Detailed Findings

ID	Title	Severity	Status
H-01	UniswapV3 slippage checks can be circumvented	High	Fixed
M-01	Incorrect asset pricing in CurveLib	Medium	Acknowledged
M-02	Relayer can force losses onto Morpho vault depositors through reallocations	Medium	Fixed
L-01	Hardcoded maxFee = 0 for CCTP Standard transfers	Low	Acknowledged
L-02	minTokenOut calculation in redeemPendlePT incorrectly calculates SY token amount	Low	Acknowledged
L-03	UniswapV3 liquidity can be deposited on donated or outdated positions	Low	Fixed
L-04	Missing reentrancy protection for non-CEI implementations	Low	Acknowledged
I-01	Unnecessary double call in LZForwarder.configureSender	Informational	Fixed
I-02	LZForwarder.sendMessage does not propagate result	Informational	Acknowledged

High Severity Issues

H-01. UniswapV3 slippage checks can be circumvented		
Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: UniswapV3Lib.sol	Status: Fixed	

Description:

The protection against pool manipulation attacks, through maxSlippage, is insufficient since it is based on a price that is sourced from slot0. The same issue applies to swapMaxTickDelta.

A malicious relayer can:

- manipulate the pool price with a front-running swap;
- send a swap with a very low minAmountOut
- At this point, the check below that is intended to protect from this scenario becomes ineffective. It will still pass, because if endingBalance almost equals startingBalance or so, the product of their difference with maxSlippage does still allow for an arbitrarily low minAmountOut even when maxSlippage is around .99e18.

```Rust
File: src/libraries/UniswapV3Lib.sol
103:                  require(params.minAmountOut  >=  (endingBalance  -  startingBalance)  *
params.maxSlippage / 1e18 , "UniswapV3Lib/min-amount-not-met");
```

The potential damage, if we use the test rateLimits as an indication, is $1M per asset/pool

Recommendations: Either use a TWAP or a manipulation-resistant oracle as a source of pricing

Customer's response: Fixed with 8899713.

Fix Review: Fix confirmed.

Medium Severity Issues

M-01. Incorrect asset pricing in CurveLib		
Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: CurveLib.sol	Status: Acknowledged	

Description:

In CurveLib, the price estimation in the below code is based on curve stored_rates, which can be an issue in extreme edge cases.

The below formula calculates the minimum amountOutMin that a relayer is allowed to pass when swapping assets on Curve:

```
Rust
File: grove-alm-controller/src/libraries/CurveLib.sol
uint256 minimumMinAmountOut = params.amountIn
    * rates[params.inputIndex]
    * params.maxSlippage
    / rates[params.outputIndex]
    / 1e18;
```

This calculation works under the assumption that rates[i] (from pool.stored_rates()) is a surrogate for pricing. stored_rates instead represents the scaling of "staked" assets relative to their underlying, and does not represent a state of the market, and even less, an indication of fair value.

If this were a normal pool, a malicious relayer could use a manipulation attack to easily extract value. However, since the protocol is only using Stableswap-NG pools with an extremely high amplitude, the 1:1 exchange is nearly always maintained. It would require a real-life black swan event (f.e. Luna UST) to push the pool to such extremes that even with a high amplitude the 1:1 ratio would break.

Recommendations:
In order to accurately estimate the fair price to enforce maxSlippage in swap, addLiquidity and removeLiquidity operations, we recommend using a TWAP or off-chain oracle.

While we don't recommend using spot prices, using stored_rates in conjunction with price_oracle readings like done in Curve oracles would be the textbook way to do so.

Customer's response: Acknowledged. We will only be using Stable NG pools with current functional support.

M-02. Relayer can force losses onto Morpho vault depositors through reallocations		
Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: ForeignController.sol	Status: Fixed	

Description:

The `ALMProxy` serves as a Morpho vault allocator based on the `ForeignController` Morpho logic. One of the functions a relayer can call is `reallocateMorpho()` which allows assets to be reallocated between different markets.

It is therefore possible that a malicious relayer calls this function to perform a reallocation towards markets with pending liquidations, potentially increasing the protocol's - as well as other depositor's - exposure to bad debt socialization.

Recommendations: Consider making reallocations available only to a completely trusted role like the admin.

Customer's response: Fixed with [b5d47f1](b5d47f1).

Fix Review: Fix confirmed.

Low Severity Issues

L-01. Hardcoded maxFee = 0 for CCTP Standard transfers		
Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: CCTPLib.sol	Status: Acknowledged	

Description:

In CCTPLib, the argument passed as maxFee for depositForBurn calls is hardcoded to zero:

Rust
```
cctp.depositForBurn,
    (
        usdcAmount,
        destinationDomain,
        mintRecipient,
        address(usdc),
        bytes32(0), // destinationCaller = 0 means anyone can relay
        0,          // maxFee = 0 for standard burns (no fast burn fee)
        2_000       // minFinalityThreshold = 2000 for standard (finalized) messages
    )
```

While this is correct at the time of writing, it is worth noting that the CCTP documentation is explicit on the possibility that Standard transfers have non-zero fees:

• *If **maxFee** is less than the minimum Standard Transfer fee, the burn reverts onchain.*

Recommendations: Make maxFee configurable (one per source chain).

Customer's response: Acknowledged, will not fix. Circle explicitly mentions which deployments have the fee switch and for now it's only Sei. On other deployments turning on fee switch is not even possible because of that.

L-02. minTokenOut in redeemPendlePT incorrectly calculates SY token amount

Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: librairies/PendleLib.sol	Status: Acknowledged	

Description: In redeemPendlePT, the minTokenOut is calculated as the intermediary SY token. This is forwarded to the pendleRouter.redeemPyToToken function in the TokenOutput struct. Within Pendle the internal function _redeemSyToToken is called where we find on line 90:

```Rust
contracts/router/base/ActionBase.sol#90
if (netTokenOut < out.minTokenOut) revert("Slippage: INSUFFICIENT_TOKEN_OUT");
```

The net amount of asset tokens are compared with an amount of SY tokens, which is incorrect. While for the vast majority of markets, the exchange rate between the two is indeed 1:1, Pendle documentation explains that this does hold for all SY (Standard SYs)

If the market is a non-standard one where the rate of exchange is not 1:1, there are 2 outcomes:

- SY minTokenOut is smaller than intended => slippage check is too loose
- Sy minTokenOut is greater than intended => DoS, the function will always revert on the slippage check.

Recommendations: While this has the precondition of actively choosing a non-standard market, there is no functional reason to calculate the SY amount for minTokenOut. We recommend:

```Rust
uint256 minTokenOut = params.pyAmountIn - 5;
```

Customer's response: Acknowledged, will not fix. We will only be using standard SYs.

L-03. UniswapV3 liquidity can be deposited on donated or outdated positions

Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: UniswapV3Lib.sol	Status: Fixed	

Description: The UniswapV3Lib library used by MainnetController and ForeignController implements a tickBounds check to limit the impermanent loss that relayers can cause by depositing on positions outside of the trading range for the pool.

This check is enforced at position creation, and later when liquidity is increased, is applied indirectly by checking that the owner of the position is the ALM proxy.

The indirect check is however insufficient for two reasons:

- The position could have been created by someone else, with an arbitrary tick range, then donated to the pool
- The position could have been created by the controller, however when the configured tickBounds were different, for example due to evolving market conditions

In these two situations, relayers can deposit liquidity on UniswapV3 positions that don't match what is allowed by the admin configuration.

Recommendations: We recommend changing the key calculation for UniswapV3 liquidity rate limiting - having liquidity operations rate limited by position instead of pool would cover both situations:

- In the donation case, the donated positions would by default come with zero rate limit, therefore disallowing relayers to deposit
- In the market evolution case, the governance can gracefully deprecate old positions by disallowing deposits and allowing withdrawals, while still allowing deposits on the newer positions

Customer's response: Fixed with abdd2ca and 2c6e3d4.

Fix Review: Fix confirmed.

L-04. Missing reentrancy protection for non-CEI implementations		
Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: MainnetController.sol, ForeignController.sol	Status: Acknowledged	

Description: Many of the operations that MainnetController and ForeignController drive the ALM Proxy for are not implemented with a checks-effects-interactions design. This is mostly prominent for rate limits that are applied later, but more importantly to slippage protections in UniswapV3 operations.

These checks could be biased and in the worst case circumvented in case an external call gives control of execution back to the relayer.

Recommendations: We did not identify any concrete scenario with the current implementation where this could be exploited, however we recommend the good practice of defending in depth by guarding the MainnetController and ForeignController entry points against reentrancy since some of its logic does not follow the checks-effects-interactions pattern.

Customer's response: Acknowledged; will address at a later date.

Informational Issues

I-01. Unnecessary double call in LZForwarder.configureSender

Description:

The LZForwarder.configureSender function is a helper for propagating sender side configuration to the LayerZero endpoint.

It propagates two types of configuration: UlnConfig and ExecutorConfig .

Because the setConfig function of the endpoint accepts an array of configurations, the function could be optimized to send a single call with both configurations.

Customer's response: Fixed with 70480b9.

Fix Review: Fix confirmed.

I-02. LZForwarder.sendMessage does not propagate result

Description:

The sendMessage function of LZForwarder is a helper method for sending cross-chain messages through LayerZero. The LayerZero endpoint it calls returns a MessagingReceipt structure that contains information that is useful for tracking the cross-chain delivery.

This information could be useful in an integrated setup where the LZForwarder library is called by a smart contract through a UI, but is however discarded.

Recommendations: We recommend having the sendMessage function return the MessagingReceipt instead of dropping this information.

Customer's response: Acknowledged, will not fix. No forwarder returns values and we won't use them in the Spell flow.

Disclaimer

About Certora

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

Certora also provides services such as auditing, formal verification projects, and incident response.