Security Assessment Report

Spark ALM Controller

December 2025

Prepared for Spark Foundation

certora

Table of content

certora

Project Summary

Project Scope

Project Name	Repository (link)	Commit Hash	Platform
Spark ALM Controller	https://github.com/sparkdotfi/spark-alm-controller	3e9be4e (initial commit) 3dbc7cb v1.9.0 (fix commit)	Solidity

Project Overview

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

```
src/MainnetController.sol
src/libraries/UniswapV4Lib.sol
src/libraries/ERC4626Lib.sol
src/libraries/CCTPLib.sol
src/libraries/PSMLib.sol
src/libraries/ApproveLib.sol
src/libraries/AaveLib.sol
src/libraries/CurveLib.sol
src/RateLimitHelpers.sol
src/ForeignController.sol
src/RateLimits.sol
src/ALMProxyFreezable.sol
src/interfaces/IALMProxy.sol
src/interfaces/ILayerZero.sol
src/interfaces/IRateLimits.sol
src/interfaces/UniswapV4.sol
src/interfaces/CCTPInterfaces.sol
src/interfaces/Common.sol
src/OTCBuffer.sol
src/ALMProxy.sol
```

The work was undertaken from **December 4, 2025,** to **December 15, 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

Spark 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 (v1.9.0) added a new integration with UniswapV4 and refactored the logic for other integrations into libraries to fit the code size limits.

Assessment Methodology

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

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

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

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

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

certora

Threat and Security Overview

System overview

The Spark ALM has the goal of reallocating funds deposited into the protocol across multiple DeFi and staking integrations on multiple EVM chains including Ethereum Mainnet.

The architecture of the protocol revolves around the `ALMProxy` contract, which is a simple and secure contract that allows other addresses with a `CONTROLLER` role to trigger external and delegate calls.

Controllers act as pluggable logic for the ALM Proxy in a composable way, similar to how a delegatecall-based implementation would.

The ALM Proxy is the contract holding assets under management, and when funds need to be moved, is instructed by the Controllers via the `doCall` and `doCallWithValue` functions.

Controllers (`MainnetController` on Ethereum mainnet, `ForeignController` on other chains) are operated by addresses having either of the following access control roles:

- `RELAYER`: the main operator of the protocol, normally driven by off-chain logic, and responsible for adding, removing, or otherwise converting liquidity across the supported integrations
- `DEFAULT_ADMIN_ROLE`: assigned to the protocol governance and responsible for configuring, the boundaries of the actions that `RELAYER` can do

Controllers enforce these boundaries through configuration in the form of either:

- Static limits saved in storage variables of the Controller contracts themselves
- Dynamic, rate-based limits that are handled through the `RateLimits` accessory contract

Threat model

Much of the security of the protocol revolves around the following key concepts, whose purpose is to limit its exposure to the eventuality of the relayer - being typically a hot wallet - acting maliciously:

- No action on the ALM Proxy should be allowed to unauthorized actors, directly or through the Controller contracts;
- only relayers are allowed to perform operations on the ALM Proxy, and only through the logic enabled by Controller contracts;
- Controller contracts enforce precise boundaries under which relayers can operate:
    - relayers are allowed to directly extract value from the ALM proxy only when explicitly allowed by the governance - by default, all fund flows start and end in with assets held by the ALM proxy
    - operations that otherwise allocate, deallocate or convert assets held by ALM Proxy itself are rate-limited, in order to cap the losses incurred through lossy operations like swaps or otherwise limit the rate at which assets can be concentrated or removed from specific integrations
- there should be no way for the relayer to circumvent any of these boundaries, and only the governance should be allowed to relax or otherwise influence these boundaries
- this applies to boundaries in the form of storage variables in the Controller contracts as well as rate limits configured in the `RateLimits` contract

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	6	6	1
Informational	1	1	-
Total	10	10	3

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	A malicious relayer can steal funds by depositing on arbitrary UniV4 positions	High	Fixed
M-01	Relayer can force losses onto Morpho vault depositors through reallocations	Medium	Fixed
M-02	Incorrect asset pricing in CurveLib	Medium	Acknowledged
L-01	No deadline for uniswap liquidity modifications	Low	Acknowledged
L-02	Rate limiting is inconsistently applied to fees collected from Uniswap	Low	Acknowledged
L-03	Getting rewards from Spark farms requires unstaking	Low	Acknowledged
L-04	Inaccurate rate limiting logic for Maple redemptions	Low	Acknowledged
L-05	Missing entry point for deprecating positions	Low	Fixed
L-06	Insufficient slippage protection in UniswapV4 functions	Low	Acknowledged
I-01	Execution control can be given when decreasing liquidity in Uniswap	Informational	Acknowledged

High Severity Issues

H-01. A malicious relayer can steal funds by depositing on arbitrary UniV4 positions

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: UniswapV4Lib.sol, MainnetController.sol	Status: Fixed	

Description: Uniswap positions are created with tight and governance-controlled tick ranges (`uniswapV4TickLimits` in MainnetController.sol) to limit losses in case of malicious relayers:

```
JavaScript
File: spark-alm-controller/src/libraries/UniswapV4Lib.sol
40:     function mintPosition(
---
50:     )
51:         external
52:     {
53:         _checkTickLimits(tickLimits[poolId], tickLower, tickUpper);
```

However, a similar validation is missing upon increasing the liquidity of an existing position owned by the ALM proxy.

Therefore, a malicious relayer can mint on their own an UniswapV4 position NFT with arbitrary ranges, transfer it to the ALM proxy, and trigger a deposit (`increaseLiquidityUniswapV4`) when the pool price is manipulated via a sandwich attack, making a profitable exchange at the expense of the ALM proxy and the protocol.

Recommendations: Consider enforcing tick limit checking at liquidity provisioning, similarly to position minting.

Customer's response: Fixed with c948112.

Fix Review: Fix confirmed.

Medium Severity Issues

M-01. 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 bad debt 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 default admin.

Customer's response: Fixed with c1e80c4.

Fix Review: Fix confirmed.

M-02. 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 the pool's stored_rates, which can be an issue in extreme edge cases.

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

```
Rust
File: spark-alm-controller/src/libraries/CurveLib.sol
102:         uint256 minimumMinAmountOut = params.amountIn
103:             * rates[params.inputIndex]
104:             * params.maxSlippage
105:             / rates[params.outputIndex]
106:             / 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 respective 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, under the assumption that 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 (i.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 have decided to accept this risk to maintain lower smart contract complexity. Additionally, there are alternative avenues available for swapping assets if necessary.

Low Severity Issues

L-01. No deadline for uniswap liquidity modifications		
Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: UniswapV4Lib.sol	Status: Acknowledged	

Description: When modifying liquidity, the call to UniswapV4 is encoded with `deadline` hardcoded to `block.timestamp`:

```
JavaScript
File: spark-alm-controller/src/libraries/UniswapV4Lib.sol
497:     function _getModifyLiquiditiesCallData(bytes memory actions, bytes[] memory params)
498:         internal view returns (bytes memory callData)
499:     {
500:         return abi.encodeCall(
501:             IPositionManagerLike.modifyLiquidities,
502:             (abi.encode(actions, params), block.timestamp)
503:         );
504:     }
505:
```

This means that the transactions that trigger the liquidity increase can successfully be included in any block, and as a consequence liquidity could be provided to unintended price ranges if the transaction is delayed to after a meaningful price movement.

Recommendations: Consider allowing the relayer to explicitly provide a deadline.

Customer's response: Acknowledged.

L-02. Rate limiting is inconsistently applied to fees collected from Uniswap

Severity: **Low**	Impact: **Low**	Likelihood: **Medium**
Files: UniswapV4Lib.sol	Status: Acknowledged	

Description: Because Uniswap V4 positions automatically collect fees during operations that modify liquidity, LP fees will be collected when both increasing and decreasing the liquidity provisioned for positions.

When liquidity is decreased, collected fees are counted through the ALM proxy balance deltas:

```
JavaScript
File: spark-alm-controller/src/libraries/UniswapV4Lib.sol
358:         uint256 rateLimitDecrease =
359:             _getNormalizedBalance(token0, endingBalance0 - startingBalance0) +
360:             _getNormalizedBalance(token1, endingBalance1 - startingBalance1);
361:
362:         // Perform rate limit decrease.
363:         IRateLimits(rateLimits).triggerRateLimitDecrease(
364:             RateLimitHelpers.makeBytes32Key(LIMIT_WITHDRAW, poolId),
365:             rateLimitDecrease
366:         );
```

However, when liquidity is increased, fees are discounted from rate limits, and for effect of the `_clampedSub` floor at zero, can also be collected without any change to rate limits by triggering a liquidity increase with 0 amounts.

Recommendations: Due to the low impact and absence of an easy fix for the issue, our intention in this case is to raise awareness of the issue without recommending an immediate fix.

Customer's response: Acknowledged. We agree the finding is valid; however, the impact is minimal. We have decided against implementing a fix to prioritize keeping contract complexity low.

L-03. Getting rewards from Spark farms requires unstaking

Severity: **Low**	Impact: **Low**	Likelihood: **Medium**
Files: MainnetController.sol	Status: Acknowledged	

Description: The withdrawFromFarm function allows simultaneous unstaking (`withdraw`) and reward collection (`getReward`) from Spark farms. Because this is the only entry point for calling `getReward` and farms prevent withdrawal requests with zero amounts, collecting rewards unnecessarily forces the caller to also unstake value.

Recommendations: Consider splitting the `getReward` and `withdraw` calls into distinct entry points in MainnetController.

Customer's response: Acknowledged. We accept this as a known operational limitation of the current design.

L-04. Inaccurate rate limiting logic for Maple redemptions

Severity: **Low**	Impact: **Low**	Likelihood: **Medium**
Files: MainnetController.sol	Status: Acknowledged	

Description: The `requestMapleRedemption` function in MainnetController requests redemption of `shares`, but rate limiting is applied on `assets`; to achieve this, rate limiting is applied on the conversion via the `convertToAssets` function:

```
JavaScript
File: spark-alm-controller/src/MainnetController.sol
808:     function requestMapleRedemption(address mapleToken, uint256 shares) external
nonReentrant {
809:         _checkRole(RELAYER);
810:         _rateLimitedAddress(
811:             LIMIT_MAPLE_REDEEM,
812:             mapleToken,
813:             IMapleTokenLike(mapleToken).convertToAssets(shares)
814:         );
```

In the Maple case, this conversion is inaccurate because it does not count the unrealized losses.

Recommendations: Consider using the `convertToExitAssets` function that better reflects the assets withdrawn.

Customer's response: Acknowledged. We utilize convertToAssets specifically because it is the more conservative approach.

L-05. Missing entry point for deprecating positions

Severity: **Low**	Impact: **Low**	Likelihood: **Medium**
Files: UniswapV4Lib.sol, MainnetController.sol	Status: Fixed	

Description: The protocol enforces `tickLimits` to ensure newly-created positions don't create a path for relayers to cause impermanent loss by concentrating liquidity outside the typical trading range for a pool.

This check is however applied only at position creation, so if the exchange price evolves over time, there is no way for the governance to deprecate positions with outdated tick limits.

Recommendations: Consider applying liquidity rate limits per position instead of pool.

Customer's response: Fixed with c948112.

Fix Review: Fix confirmed.

L-06. Insufficient slippage protection in UniswapV4 functions

Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: UniswapV4Lib.sol	Status: Acknowledged	

Description: The `maxSlippage` setting alone is not an accurate protection against pool price swings because it clamps `amountOutMin` to a value relative to a hardcoded 1:1 assumption for the pool's price which may not be accurate in case of a depeg or if non-stable pools are supported in the future.

Recommendations: We recommend checking `amountOutMin` against a `maxSlippage` scaling on the TWAP reading of the pool price instead.

Alternatively, we recommend:

- hardening the pool review process to make sure that no non-stablecoin pool is ever approved with the current implementation

- preparing for the possibility of swaps failing in case one of the two assets in a stable pool depegs.

Customer's response: Acknowledged. The system operates under the assumption of a 1:1 peg for these assets. In the event of a depeg, protocol administrators will intervene to adjust rate limits and manage the situation manually.

Informational Issues

I-01. Execution control can be given when decreasing liquidity in Uniswap

Description: When decreasing liquidity in UniswapV4 positions, the NFT owner check is not performed like in other liquidity operations. This allows for the situation where the contract allows liquidity removal from a position NFT owned by a malicious relayer, who also registered themselves as `subscriber`.

In this situation, Uniswap gives execution control to the NFT `subscriber` via a callback, and the called address can execute potentially dangerous code.

Recommendations: While we marked this finding as informational as we did not find any way to exploit this scenario with the current implementation, we recommend implementing an NFT ownership check also to liquidity removal flow.

Customer's response: Acknowledged. We have determined that the code functions correctly and securely regardless of whether a subscriber triggers a callback function based on position events.

Disclaimer

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

About Certora

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