

# Code Assessment of the Spark ALM Controller Smart Contracts

March 28, 2025

Produced for



by



# Contents

<b>1</b>	<b>Executive Summary</b>	<b>3</b>
<b>2</b>	<b>Assessment Overview</b>	<b>5</b>
<b>3</b>	<b>Limitations and use of report</b>	<b>14</b>
<b>4</b>	<b>Terminology</b>	<b>15</b>
<b>5</b>	<b>Open Findings</b>	<b>16</b>
<b>6</b>	<b>Resolved Findings</b>	<b>20</b>
<b>7</b>	<b>Informational</b>	<b>23</b>
<b>8</b>	<b>Notes</b>	<b>25</b>

# 1 Executive Summary

Dear all,

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

SparkDAO implements the Spark ALM Controller, a suite of contracts of the Spark Liquidity Layer designed to manage and control the flow of liquidity originating from DSS Allocator. This latest iteration reviewed added support for ERC-7540 (for Centrifuge), SuperState, Morpho, Maple and BUIDL and revised the Freezer role to remove compromised relayers instead of pausing the contract.

The most critical subjects covered in our audit are functional correctness, access control, and the integration with 3rd-party protocols. The general subjects covered are gas efficiency, documentation and composability.

Security regarding all the aforementioned subjects is high.

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.

<b>Critical</b> -Severity Findings	0
<b>High</b> -Severity Findings	0
<b>Medium</b> -Severity Findings	0
<b>Low</b> -Severity Findings	8
• <b>Code Corrected</b>	1
• <b>Specification Partially Changed</b>	1
• <b>Risk Accepted</b>	5
• <b>Acknowledged</b>	1

## 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 Spark 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	15 Sep 2024	<a href="#">7a0535ee07815a2c8604d58469393c56a62d5b81</a>	Initial Version - v1.0.0-beta.0
2	26 Sep 2024	<a href="#">2087250fc5988d3f0117f23e3f959f9423d04909</a>	Second Version
3	03 Oct 2024	<a href="#">c76b422b053dd055aeb2cd555acfe353f05b316e</a>	Fixes And Deployment Scripts - v1.0.0-beta.1
4	07 Oct 2024	<a href="#">342fe537b020ffa8ea7fcedf166b59b7ede21232</a>	Fix Deployment Scripts
5	08 Oct 2024	<a href="#">52deda866ec8abdeaae9ace8574457d3e4209c36</a>	Setting USDS And sUSDS Rate Limits
6	22 Oct 2024	<a href="#">6058f68f79520eb06ea8eded146da13039c47525</a>	Bump Version - v1.0.0
7	29 Nov 2024	<a href="#">f81a7366f339d806a07a992c3aef2afe9a063e13</a>	Aave And Ethena Integrations - v1.1.0-beta.0
8	06 Dec 2024	<a href="#">ad4391c37aa262d3c578a757700c1b6e86a96060</a>	Withdrawals Rate Limit - v1.1.0-beta.1
9	13 Dec 2024	<a href="#">2bb2680893aa3e42210c8f907ec4d5778ace9fe6</a>	Release v1.1.0
10	30 Dec 2024	<a href="#">eb8192199ee7713b583dd1b4920ec156c2333830</a>	Updated Init Scripts - v1.2.0-beta.0
11	30 Jan 2025	<a href="#">6a35adaddc38666ae2f75f0d4792f9f526de0cab</a>	RWAs And Morpho Integrations - v1.3.0-beta.0
12	25 Mar 2025	<a href="#">3de54c3f91d219401406459ac66e4b9b1b9fce44</a>	Release v1.3.0

For the solidity smart contracts, the compiler version 0.8.21 was chosen with the evm version set to shanghai. Since version 10, the compiler version used is 0.8.25. Since version 11, the evm version used is cancun. In addition, this review was based on the integrations with 3rd-party protocols' implementations at the time of respective Spark ALM releases and cannot account for future updates and changes of the 3rd-party protocols.

The files in scope were:

```
src/  
  ALMProxy.sol  
  ForeignController.sol
```

```
MainnetController.sol
RateLimitHelpers.sol
RateLimits.sol
interfaces/
  IALMProxy.sol
  IRateLimits.sol
  CCTPInterfaces.sol
```

In **Version 3** the following files were further added to scope:

```
deploy/
  ControllerDeploy.sol
  ControllerInit.sol
  ControllerInstance.sol
```

In **Version 10** the following files were removed from the scope:

```
deploy/ControllerInit.sol
```

In **Version 10** the following files were added to the scope:

```
deploy/
  ForeignControllerInit.sol
  MainnetControllerInit.sol
```

### 2.1.1 Excluded from scope

All other files are out of scope. It is assumed that USDC and CCTP will work honestly as documented.

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.

All other 3rd-party protocols that the ALMProxy interacts with, i.e. Aave and Ethena (added in version 7); Centrifuge, SuperState, Morpho, Maple and BUIDL (added in version 11), are out of scope and assumed to work correctly as documented.

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

## 2.2 System Overview

This system overview describes the initially received version (**Version 1**) of the contracts as defined in the [Assessment Overview](#).

At the end of this report section, we have added subsections for each of the changes according to the versions.

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

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



On each chain, the following contracts are deployed:

1. **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.
2. **Controllers**: Dictate which operations an ALMProxy shall perform. Note that multiple controllers could point to the same ALMProxy.
3. **RateLimits**: Computes limits for liquidity flows.

All of the contracts inherit the standard **AccessControl** and grant the `DEFAULT_ADMIN_ROLE` role to an `admin` which can configure other roles.

## 2.2.1 ALMProxy

The ALMProxy provides the following privileged functions that are restricted to addresses with the `CONTROLLER` role:

1. `doCall()`: triggers a call from ALMProxy to a target contract with `msg.value`.
2. `doCallWithValue()`: triggers a call from ALMProxy to a target contract with a specific `value`.
3. `doDelegateCall()`: triggers a delegatecall from ALMProxy to a target contract.

## 2.2.2 Controllers

**MainnetController** defines operations in the context of the mainnet ALMProxy that are restricted to addresses with the `RELAYER` role:

1. `mintUSDS()` / `burnUSDS()`: leverages **AllocatorVault** to mint or burn (`draw()` or `wipe()`) USDS.
2. `depositToSUSDS()` / `withdrawFromSUSDS()`: wraps USDS to SUSDS or vice-versa.
3. `redeemFromSUSDS()`: this further provides the option to unwrap SUSDS by specifying the shares to burn.
4. `swapUSDSToUSDC()` / `swapUSDCToUSDS()`: leverages the PSM to swap between USDS and USDC without fees (`buyGemNoFee` and `sellGemNoFee`).
5. `transferUSDCToCCTP()`: leverages CCTP (Circle's Cross-Chain Transfer Protocol) to bridge USDC to a recipient (expected to be another ALMProxy) on a foreign domain.

**ForeignController** defines operations in the context of a foreign domain ALMProxy that are restricted to addresses with the `RELAYER` role:

1. `depositPSM()` / `withdrawPSM()`: deposits / withdraws specific assets to / from Spark PSM.
2. `transferUSDCToCCTP()`: leverages CCTP (Circle's Cross-Chain Transfer Protocol) to bridge USDC to a recipient (expected to be another ALMProxy) on a foreign domain.

Both controller types provide the following administrative functions:

1. `freeze()`: restricted to the `FREEZER` role to pause all the operations defined above.
2. `reactivate()`: restricted to the `DEFAULT_ADMIN_ROLE` to reactivate the operations.
3. `setMintRecipient()`: set the token recipient of a destination domain when bridging USDC with CCTP.

In addition, except interaction with SUSDS, all other operations are subject to specific limits defined by [Rate Limits](#).

## 2.2.3 Rate Limits

RateLimits defines a limit on a rate for a given key. The rate limit will linearly grow from `lastAmount` with `slope` over the time elapsed (tracked with `lastUpdated`), and is capped `maxAmount`. The full rate limit data or the current rate limit can be queried from `getRateLimitData()` and `getCurrentRateLimit()`, respectively.

The data can be set by the `DEFAULT_ADMIN_ROLE` with `setRateLimitData()` (two signatures available) or `setUnlimitedRateLimitData()`.

The following functions are introduced to update the rate limit by the `CONTROLLER` role:

1. `triggerRateLimitDecrease()`: It deducts an amount from the current rate limit and sets `lastUpdated` to `block.timestamp`. In case of the rate limit is insufficient, the call will revert.
2. `triggerRateLimitIncrease()`: It adds an amount to the current rate limit (capped by the `maxAmount`) and sets `lastUpdated` to `block.timestamp`.

## 2.2.4 Deployment Scripts

In version 3 (see [Changes in Version 3](#)), deployment scripts have been added that can be used in governance spells.

`MainnetControllerDeploy` and `ForeignControllerDeploy` libraries both offer the functions `deployController` and `deployFull` to deploy on mainnet and on the foreign chain, respectively. While `deployController` solely deploys a controller with a given ALM proxy and rate limit contract, the `deployFull` function deploys the ALM proxy and rate limit contract as well.

The `MainnetControllerInit` library implements functionality to initialize the ALM controller architecture on mainnet. Three functions are offered:

1. `subDaoInitController`: For SubDAO. Sanity checks along with assigning the expected roles and setting rate limits for the expected keys. Additionally, allows to revoke the roles of a previously used controller contract.
2. `subDaoInitFull`: For SubDAO. `subDaoInitController` with additional sanity checks on the ALM proxy and the rate limit contract along with setting the required access control on the allocator infrastructure.
3. `pauseProxyInit`: For Sky Governance. Calls `kiss` on the PSM to allow fee-less swaps on the PSM. Note that no checks are performed. Thus, this should be performed only after the expected spell to setup the ALM controller have been performed by the SubDAO.

Note that the current version of the mainnet controller will be only able to use CCTP to bridge to Base.

The `ForeignControllerInit` library implements functionality to initialize the ALM controller architecture on a foreign chain (at the time of writing: Base). Only `init` is offered. Essentially it performs similar actions to `subDaoInitFull` in the `MainnetControllerInit`. However, customized to the `ForeignController` contract. Note that only USDC will be usable in the integration with the Spark PSM on L2.

Note that L1 addresses are not published to the Chainlog (which is intended).

## 2.2.5 Changes in Version 2

The changes below were introduced in version 2 of the codebase:

1. The `transferUSDCToCCTP` function now has an additional limit on the amount of USDC that can be bridged to a given domain.
2. The `MainnetController`'s `swapUSDCToUSDS` function now uses the PSM's `fill` function to allow filling the PSM if needed. As a consequence, swaps will be repeated as long as the full USDC amount has not been swapped (with filling happening before the swap). In case the USDC amount cannot be swapped, a revert will occur as before.



3. The unused immutables `usds` and `susds` have been removed from the `ForeignController`, hence these two addresses cannot be queried from the controller anymore.

## 2.2.6 Changes in Version 3

The following changes were introduced in version 3 of the codebase:

1. Function `receive()` has been added to the `ALMProxy` to support receiving ETH.
2. Events will be emitted when changing the controller's `active` status and setting the mint recipient on a destination domain.
3. Deployment scripts have been included in scope, see [Deployment Scripts](#).

## 2.2.7 Changes in Version 4

Now, on the foreign controller's initialization, limits for USDS and sUSDS are set up for PSM deposits and withdrawals.

## 2.2.8 Changes in Version 7

In this version, the integration with any ERC-4626 compliant vault, Aave V3, and Ethena was added. These interactions can only be triggered by the relayer when the controller is active.

For the ERC-4626 compliant vault and Aave V3 integration in the `MainnetController` and `ForeignController`:

- The relayer can deposit into any ERC-4626 vault with `depositERC4626()` respecting the respective rate limit on each vault, the minted shares will be credited to the `ALMProxy`. The relayer can also withdraw from any ERC-4626 vault with `withdrawERC4626()` or `redeemERC4626()` without any rate limit. The sUSDS related interactions can be achieved with these ERC-4626 functions, hence they are removed.
- The relayer can also deposit into Aave with `depositAave()` respecting the `aToken` dependent rate, and the newly minted `aToken` will be credited to the `ALMProxy`. The relayer can also withdraw any underlying asset of an `aToken` from Aave with `withdrawAave()` without any rate limit.

For Ethena USDe and sUSDe integration in the `MainnetController`:

- The relayer can configure any amount of delegated signers for the `ALMProxy` with `setDelegatedSigner()` and `removeDelegatedSigner()`. The assigned delegated signers must explicitly accept the delegation with `confirmDelegatedSigner()` on Ethena before they can sign any `Order` on behalf of the `ALMProxy`. An `Order` is an expirable intent to mint or redeem USDe with fixed spending collateral and outcome USDe.
- In addition, it requires a minter and burner role on Ethena minting contract to submit the tx with the `Order` and signature prepared by the delegated signers.
- The relayers can only `prepareUSDeMint()` and `prepareUSDeBurn()` by triggering an approval from the `ALMProxy` to the Ethena minting contract. Further operations from the delegated signers, Ethena minter and burners are required to complete the action.
- Once the `ALMProxy` obtained USDe, it can be deposited into sUSDe to earn an interest. The deposit into sUSDe follows the typical ERC-4626 deposit interface.
- The instant withdrawal of sUSDe is blocked currently, which requires a cooldown period followed by an `unstake()` call. The relayer can trigger the cooldown with `cooldownAssetsUSDe()` or `cooldownSharesUSDe()`, which burned the shares and credited the USDe to the `USDeSilo` contract. Once the `cooldownDuration` is reached, the relayer can trigger the exit of USDe to the `ALMProxy` with `unstake()`.

## 2.2.9 Changes in Version 8

In this version withdrawal rate limits for ERC-4626 and Aave have been added. Hence the ALMProxy will only interact with authorized ERC-4626 and Aave aTokens with limited rates.

## 2.2.10 Changes in Version 10

In this version, the init script library has been refactored into two files. Both the `MainnetControllerInit` and the `ForeignControllerInit` support the functions `initAlmSystem()` and `upgradeController()`. The former corresponds to the initialization of the initial deployment (previously: `subDaoInitFull / init` with the previous controller being `0x0`) while the latter corresponds to a controller upgrade (previously: `subDaoInitController / init` with the previous controller not being `0x0`).

Note that rate limits are not configured anymore in the initialization scripts. Further, the helper function `setRateLimitData()` calling `IRateLimits.setRateLimitData()` has been moved to the `RateLimitHelpers` library which governance spells could use if necessary. Additionally, a getter `unlimitedRateLimit()` for generating the parameters used for setting an unlimited rate limit is provided.

## 2.2.11 Changes in Version 11

- New freezer behavior: The contract no longer includes an active flag that the freezer role could toggle to pause the contract. Previously, this was intended as a safeguard against a compromised relayer, allowing the admin role to replace them through governance which is a slow process. The new implementation replaces `freeze()/reactivate()` with a `removeRelayer` function, enabling the freezer to directly remove a compromised relayer. The contract is intended to have a main relayer and a configured backup. If all relayers are removed by freezer-role accounts, the contract becomes effectively frozen. This change applies to both the Mainnet and Foreign controllers.
- Library `RateLimitHelpers` (for deployment scripts) has been extended with custom errors and new functions:
  - `makeAssetDestinationKey()`: Generates a key based on key, asset, and destination for the new `transferAsset` function, enabling destination-based limits.
  - `unlimitedRateLimit()`: Helper function returning the maximum amount with a slope of 0.
  - `setRateLimitData()`: Performs sanity checks before calling `setRateLimitData()` on the `RateLimit` contract. The library now ensures that rate limit parameters are within reasonable bounds before setting them in the contract.
    - Max amount bounds: Must fall within a meaningful range (e.g., 1 unit and 1 trillion unit of tokens).
    - Slope validity: Replenishing rate must not be excessively fast or slow. Specifically, the slowest replenishment can refill an equivalent amount of lower bound in one hour, and the fastest replenishment can refill an equivalent amount of upper bound in one hour.
- Morpho support in `MainnetController` and `ForeignController`: New functions for the relayer, `setSupplyQueue()`, `updateWithdrawQueue()` and `reallocate()` have been added allowing to call the respective function on Morpho. To successfully execute these calls on vaults, the ALM proxy must have the allocator role in the `MorphoVault`. Deposits and withdrawals to the vault are handled via the existing ERC-4626 functions which enforce rate limits and hence restrict interactions to explicitly whitelisted vaults.

- Superstate support in MainnetController: `subscribeSuperstate()` enables the relayer to call Superstate's subscription function to mint USTB by transferring USDC to Superstate. The required USDC transfer approval is granted. Similarly, `redeemSuperstate()` allows the relayer to call Superstate's redemption function to redeem USTB for USDC, requiring USTB approval. Both functions are subject to rate limits defined by `LIMIT_SUPERSTATE_SUBSCRIBE` and `LIMIT_SUPERSTATE_REDEEM`, respectively.
- Maple support in MainnetController: While deposits happen through the existing ERC-4626 deposit function, withdrawals are not ERC-4626 compliant and work based on redemption requests. Functions for the relayer to initiate calls by the ALMProxy to `requestRedeem()` (`requestMapleRedemption()`) and `removeShares()` (`cancelMapleRedemption()`) have been added. It is assumed that manual withdrawal is disabled for ALMProxy on Maple, hence tokens will be pushed to ALMProxy once the redemption is processed. Should ALMProxy be configured to manual withdrawal on Maple, relayers have to trigger an explicit ERC-4626 `redeem()` on ALMProxy to retrieve the tokens, which further requires a withdrawal limit being configured for the maple pool.
- BlackRock BUIDL support in MainnetController: Assets are transferred out manually using the new `transferAssets` function facilitating ERC-20 transfers. This function is subject to a rate limit including the destination address. Function `redeemBUIDLCircleFacility()` enables the relayer to initiate `buildRedeem.redeem()` from the ALMProxy, expecting to return USDC in exchange for the `buildRedeem.asset`.
- Centrifuge (ERC7540) support in MainnetController: Generic support for ERC-7540 (Asynchronous ERC-4626) has been implemented to enable interaction with Centrifuge. Rate limit is configured to ensure interactions with whitelisted vault only. For deposits, `requestDepositERC7540()` allows the relayer to initiate a call to `requestDeposit()` where assets are transferred to an escrow. Once the deposit request is fulfilled, the relayer can finalize the deposit and claim all the available shares using `claimDepositERC7540()`. Similarly, for redemptions, `requestRedeemERC7540()` initiates a withdrawal with shares transferred to escrow, while `claimRedeemERC7540()` finalizes it and retrieves the max withdrawable assets.

## 2.2.12 Changes in Version 12

- Centrifuge support in MainnetController: Access to functions for canceling deposit and redemption requests has been added, along with functions to claim the funds after being fulfilled: `cancelDepositRequest()`, `cancelRedeemRequest()`, `claimCancelDepositRequest()` and `claimCancelRedeemRequest()`.

## 2.2.13 Roles & Trust Model

ALMProxy: The admin is fully trusted, otherwise, it can setup 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 setup properly by governance, for instance:

1. It requires `bud` role to swap without fee on `DssPSMLite`.
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: 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. 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.

As of version 3, deployments scripts are in scope. Before initializing the contracts, 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.

As of version 7, the integration with arbitrary ERC-4626, Aave, and Ethena requires an extended trust model. Generally the third party protocols receiving funds are assumed to be non-malicious.

**Arbitrary ERC-4626:** The admin should only configure a deposit rate limit for trusted external ERC-4626 vaults.

**Aave:** Aave governance can adjust the parameters for the reserves such as setting it inactive, paused or frozen or upgrading the pool / aToken implementations. Spark ALM Controller is subject to the potential risks of Aave, and Aave governance is fully trusted to not misbehave.

**Ethena:** The funds deposited to Ethena are subject to the risks of depegging and minting / burning limits of Ethena. The minter and burner of USDe are fully trusted, otherwise, they can DoS the USDe minting and burning. The delegated signers assigned by relayers are semi-trusted:

1. There could be a race condition if multiple delegated signers exist, who can sign orders with different volume and quotes.
2. The signers may not sign any order or not use all the allowance prepared by the ALMProxy.
3. The signers can sign with bad quotes regardless of the ones Ethena returned. It is assumed **the fully trusted Ethena minter / redeemer will never submit these malicious orders.** (see [A Compromised Ethena Minter Or Redeemer May Execute A Bad Order](#))

Semi-trusted roles are expected to operate honestly but may be compromised. They must not be able to gain control over funds or exit funds from the system. While they may continue actions until replaced, all funds must remain under the ALMProxy's control.

Holding sUSDe further implies the risks of blacklisting (for deposit / withdraw / transfer) and confiscating, the admins and other privileged roles of sUSDe are trusted to not misbehave.

The relayers are semi trusted, a compromised [Relayer Can DoS SUSDE Unstaking](#), which requires admin privilege to remove the malicious relayer. In addition, as a relayer can assign any delegated signer, a compromised relayer can incur all the aforementioned risks of a malicious delegated signer.

In addition, it is assumed Spark 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.

As of version 11, the integration with ERC-7540 (Centrifuge), Morpho, SuperState USTB, Maple, and BlackRock BUIDL further requires an extended trust model. Generally Spark is subject to the inherent risks of these protocols (i.e. risks of upgradeability, RWAs, governance ...) though they are assumed to be non-malicious.

**SuperState USTB:** The owner of USTB is fully trusted, otherwise it can for instance 1) block the deposit / withdrawals by pausing USTB and its redemption contract; 2) manipulate the conversion price by changing the oracle. It is assumed the owner of AllowList is trusted and will whitelist the Spark ALMProxy. Instant redemption is not possible if there is insufficient USDC on the RedemptionIdle contract. In addition, USTB, AllowList and RedemptionIdle contracts are deployed behind upgradeable proxies, hence the proxy admins are trusted to not upgrade to malicious implementations.

**Maple:** Maple governance is fully trusted which oversees the protocol, who can upgrade the implementations of the manager contracts which feature important logic and accounting. The redeemer of MapleWithdrawalManager is trusted to process the redemptions in time. The governor and the pool delegator is trusted since they can trigger the update of unrealized loss accounting on the LoanManager contract, which will influence the redemption conversion rate. The protocol admin and pool delegator are also trusted, otherwise they may interfere with the automatic redemption by setting the manual withdrawal for the ALMProxy.

**Morpho:** It is assumed the Spark ALMProxy has allocator role on the specified Morpho Vault. Other privileged roles of Morpho Vault: owner (assumed to be the Spark sub-proxy), curator, and guardian are generally trusted to facilitate the operations.

**BUIDL:** Since the minting process is centralized, the issuer and master of BUIDL token is fully trusted to issue the correct amount of tokens to Spark ALMProxy after the underlying assets transfer, otherwise, the full deposit may be lost. BUIDL token is deployed behind an upgradeable proxy, hence the proxy owner is trusted to not upgrade to a malicious implementation. The owner of the Redemption contract is also trusted, otherwise it can DoS the redemption by pausing the contract. In addition, it is assumed there is sufficient liquidity in the holder of the LiquiditySource contract to facilitate the redemptions.

**Centrifuge:** *Centrifuge is assumed to be the only ERC-7540 integrated.* The wards of the ERC-7540 vaults are fully trusted, otherwise they can DoS the system by changing the `manager` contract. The wards of the InvestmentManager 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. It is further assumed the tranche token being used supports `authTransferFrom()` without prior approval, otherwise, function `requestRedeemERC7540()` will always revert due to insufficient allowance.

### 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.

<b>Critical</b> -Severity Findings	0
<b>High</b> -Severity Findings	0
<b>Medium</b> -Severity Findings	0
<b>Low</b> -Severity Findings	7

- [Centrifuge Deposit / Redemption Can Be DoSed by Cancellation](#) **Risk Accepted**
- [Centrifuge Tranche Token Price May Change Between Request Submission and Execution](#) **Risk Accepted**
- [Maple Redemption Can Be DoSed](#) **Risk Accepted**
- [Outdated README](#) **Specification Partially Changed**
- [Over-reduced Limit in Maple Redemption](#) **Risk Accepted**
- [Freeze Cannot Revoke the Unused Approval](#) **Acknowledged**
- [Relayer Can DoS SUSDE Unstaking](#) **Risk Accepted**

### 5.1 Centrifuge Deposit / Redemption Can Be DoSed by Cancellation

**Security** **Low** **Version 12** **Risk Accepted**

CS-SPRKALM-008

In Version 12, 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:

SparkDAO is aware of the risk.



## 5.2 Centrifuge Tranche Token Price May Change Between Request Submission and Execution

**Correctness** **Low** **Version 11** **Risk Accepted**

CS-SPRKALM-009

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

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

## 5.3 Maple Redemption Can Be DoSed

**Security** **Low** **Version 11** **Risk Accepted**

CS-SPRKALM-010

Maple redemption can be DoSed by a compromised relay in two ways:

1. Each user can have at most 1 redemption request in `MapleWithdrawalManager`. Hence a compromised relay can keep triggering dust redemptions and block the legitimate redemptions from honest relays. In this case, the honest relays 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 relay, other relays will not be able to trigger future redemptions.

---

### Risk accepted:

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

## 5.4 Outdated README

**Correctness** **Low** **Version 11** **Specification Partially Changed**

CS-SPRKALM-011

The README has not been updated to reflect the changes introduced in Version 11. The new external system integrations are not described.

A detailed description of `transferAssets()`, which allows value to exit the ALM system, is missing. While this behavior is intentional and the destination address is restricted by the rate limit, it may conflict with the statement in the trust assumptions section: "The logic in the smart contracts must prevent the movement of value anywhere outside of the ALM system of contracts." If only system addresses are whitelisted, this may align with the intended assumptions, but clarification might be good.

The description of the freezer role's permissions has not been updated and still reflects the previous behavior. Technically by removing all relays, the freezer can still "freeze all actions".

---

### Specification partially changed:

The README has been updated:

- The controller functionality no longer lists individual protocols and supported actions but now refers to external protocols.
- The freezer role description has been updated.
- The `TransferAsset` function is not described in the current README.

## 5.5 Over-reduced Limit in Maple Redemption

**Correctness** **Low** **Version 11** **Risk Accepted**

CS-SPRKALM-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.

## 5.6 Freeze Cannot Revoke the Unused Approval

**Design** **Low** **Version 7** **Acknowledged**

CS-SPRKALM-013

The privileged role `FREEZER` can pause the `MainnetController` with `freeze()`, which prevents the 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 `MainnetController` is frozen.

In summary, after the `MainnetController` is frozen, any unused approval for the Ethena minter contract can still be used which modifies the balance of `ALMProxy`.

---

### Acknowledged:

SparkDAO acknowledged the issue and decide not to change the code since Ethena is fully trusted.

## 5.7 Relay Can DoS `SUSDe` Unstaking

**Security** **Low** **Version 7** **Risk Accepted**

CS-SPRKALM-014

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

- A cooldown must be initiated first, which (1) burns the shares and credit 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 relay can keep triggering new cooldowns with as less as 1 wei asset to block previous exits from sUSDe to USDe, hence DoS the sUSDe to USDe conversion.

**Note:** SparkDAO was aware of this issue and had reported to us before the audit. In addition, in case a malicious relayer DoSed the sUSDe `unstake()`, SparkDAO will freeze the controller, remove the malicious relayer, and reactivate the controller again.

In version 11, since function `freeze()` has been removed, instead of freezing the whole controller, the freezer should revoke the relayer role from the malicious relayer.

## 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.

<b>Critical</b> -Severity Findings	0
<b>High</b> -Severity Findings	0
<b>Medium</b> -Severity Findings	0
<b>Low</b> -Severity Findings	1
<ul style="list-style-type: none"><li>• <a href="#">Loosely Restricted Specific Calls to Arbitrary Address</a> <b>Code Corrected</b></li></ul>	
Informational Findings	6
<ul style="list-style-type: none"><li>• <a href="#">Mint Recipient Is Not Initialized</a> <b>Code Corrected</b></li><li>• <a href="#">Sanity Checks In Deployment Scripts</a> <b>Code Corrected</b></li><li>• <a href="#">Unused Parameters</a> <b>Code Corrected</b></li><li>• <a href="#">ALM Proxy Cannot Receive</a> <b>Code Corrected</b></li><li>• <a href="#">Constructor Parameters</a> <b>Code Corrected</b></li><li>• <a href="#">Events</a> <b>Code Corrected</b></li></ul>	

### 6.1 Loosely Restricted Specific Calls to Arbitrary Address

**Security** **Low** **Version 7** **Code Corrected**

CS-SPRKALM-018

**ERC4626 Integration:** The integrator can freely trigger calls (respecting the ERC-4626 withdraw and redeem interfaces) from the ALMProxy to arbitrary addresses.

**Aave Integration:** The integrator can freely call `withdrawAave()` for any aToken and amount without any rate limit. In addition, since the Aave pool address is fetched from the aToken instead of being hardcoded, the ALMProxy may eventually call an arbitrary contract with the Aave pool withdraw interface.

These calls are loosely restricted and could be unexpected if there is a function selector collision on contracts where ALMProxy has privileges.

---

#### Code corrected:

Withdrawal rate limits have been added to the withdraw / redeem logic of ERC-4626 and Aave's withdraw logic.

### 6.2 Mint Recipient Is Not Initialized

**Informational** **Version 3** **Code Corrected**



In the controller initialization library, the `RateLimit` of bridging tokens with CCTP has been configured, however, the mint recipients are not. As a result, USDC cannot be bridged after initialization and another spell is required to set the mint recipients.

---

#### Code corrected:

Mint recipients are now configured in the initialization library.

## 6.3 Sanity Checks In Deployment Scripts

Informational Version 3 Code Corrected

CS-SPRKALM-005

1. The initialization code doesn't check if the Foreign Controller is active.
  2. The status of the Spark PSM is not validated in the Foreign Controller initialization library. The Spark ALM would be subjected to Share Inflation Attack if the deployer of Spark PSM does not make the proper first deposit.
  3. In the initialization library for both the Mainnet and Foreign controllers, there is no validation to ensure that the new controller address (`controllerInst.controller`) is different from the old controller address (`params.oldController`). If both addresses are the same, the script will first grant the necessary permissions to the controller and then immediately revoke them. As a result, the controller address will not obtain the `CONTROLLER` role.
- 

#### Code corrected:

Code has been corrected to perform the respective checks.

## 6.4 Unused Parameters

Informational Version 3 Code Corrected

CS-SPRKALM-006

The initialization function `init()` of the Foreign Controller takes as parameters `params.usds` and `params.susds`. However, these parameters are not used in the function.

---

#### Code corrected:

Code has been corrected by removing `usds` and `susds` from the `AddressParams` struct.

## 6.5 ALM Proxy Cannot Receive

Informational Version 1 Code Corrected

CS-SPRKALM-001

The ALM proxy contract is intended to be used for use-cases beyond the implementations of the current controllers. In the future, scenarios might exist where a controller requires that the proxy can receive ETH

(e.g. by withdrawing from WETH). However, such use cases are not possible to implement due to the lack of a `receive` function.

---

#### Code corrected:

Function `receive()` has been added to support receiving ETH.

## 6.6 Constructor Parameters

Informational Version 1 Code Corrected

CS-SPRKALM-002

The constructor of the mainnet controller receives `buffer` as an input. However, the `buffer` could be retrieved from the `vault`. Ultimately, retrieving the `buffer` on-chain could make the code more consistent (e.g. `dai` is retrieved from `daiUsds`).

---

#### Code corrected:

Code has been corrected to retrieve `buffer` from the `vault`.

## 6.7 Events

Informational Version 1 Code Corrected

CS-SPRKALM-003

The controller contracts lack events on important state changes. More specifically no event is emitted on

1. `setMintRecipient()`
2. `freeze()`
3. `reactivate()`

which involve important state changes.

For other functions, such as `MainnetController::mintUSDS` no event is emitted. Note that the relevant events can be retrieved from the external contracts. However, that is also true for CCTP which emits `DepositForBurn` making `CCTPTransferInitiated` redundant. Nevertheless, emitting an event on every action could also be reasonable to easily allow distinguishing which controller (of the potentially many) initiated a certain sequence of calls.

---

#### Code corrected:

The following events have been added to the privileged functions in both mainnet and foreign controllers:

1. event Frozen.
2. event MintRecipientSet.
3. event Reactivated.

# 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 Maple Manual Withdraw May Be Enabled

**Informational** **Version 11** **Risk Accepted**

CS-SPRKALM-015

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 send to the user in step 2.

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

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

## 7.2 Allowance For Ethena Minter May Not Be Consumed

**Informational** **Version 7** **Acknowledged**

CS-SPRKALM-016

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

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

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

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

---

**Acknowledged:**

SparkDAO acknowledged the issue and decide not to change the code.

## 7.3 Withdraw From Aave Can Be Blocked By LTV=0 Asset

Informational Version 7 Acknowledged

CS-SPRKALM-017

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 fully balance of the asset with  $LTV==0$ , which resets the `usingAsCollateral` flag to `false` and recovers the ALMProxy from the DoS.

---

### Acknowledged:

SparkDAO 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.

## 7.4 Gas Optimizations

Informational Version 1 Code Partially Corrected

CS-SPRKALM-004

In the MainnetController's `swapUSDSToUSDC` and `swapUSDCToUSDS` functions, `to18ConversionFactor` is always queried. However, the factor is expected to be a constant and could be made an immutable.

In **Version 7** of the MainnetController:

- The `susds` immutable is no longer needed as the SUSDS related logic are replaced by the general ERC-4626 integration logic.
- The Aave V3 `POOL` is queried every time from the input aToken. Since all the aTokens share the same pool, it can be set as an immutable in the constructor.

---

### Code partially corrected:

The conversion factor has been set as an immutable in the constructor. In **Version 8** the `susds` immutable has been removed.



## 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 7

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

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

### 8.2 Aave Interprets Uint256 Max Withdrawal as Full Withdrawal

**Note** Version 7

Relayers should be aware that Aave will interpret a withdrawal with `type(uint256).max` amount as a full withdrawal with user's balance. Relayers should be careful of this special behavior if they are dependent on the input amount.

### 8.3 Avoid Morpho Deposit Into Market With Bad Debt

**Note** Version 11

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. Relayers should monitor the markets conditions and not deposit or reallocate into markets with unrealized bad debt.

### 8.4 BUIDL Deposit Cap

**Note** Version 11

ALMProxy should only deposit into BUIDL if there is any space before the BUIDL deposit cap is reached in `issueTokens()`. Otherwise, less BUIDL tokens can be minted than the USDC deposited. Though there is no deposit cap (`cap==0`) on BUIDL by the time of version 11. Relayers should be careful of the potential cap changes in the future.

## 8.5 BUIDL Redemption Will Be Blocked by USDC Transfer Fees

**Note** Version 11

Upon a redemption (`redeem()`), the BUIDL Redemption contract will check the pre- and post-transfer USDC balance difference of the user matches the input redemption amount, otherwise it will revert. This requires no transfer fee or loss on USDC, otherwise the redemption will be blocked.

## 8.6 Maple Deposit Ignores Unrealized Losses

**Note** Version 11

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

## 8.7 MorphoAllocations updateWithdrawQueue Subject to Front Running

**Note** Version 11

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 Spark-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.

## 8.8 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.