PUBLIC

Code Assessment

of the Spark ALM Controller Smart Contracts

September 11, 2025

Produced for

Spark

by

CHAINSECURITY

Contents

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. It provides several integrations with DeFi protocols and bridges.

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.

The most recent audit covers the SparkVault integration, removal of the Superstate redemption functionality and bytecode size optimizations for the MainnetController.

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.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	14
• Code Corrected	4
• Specification Changed	2
• Code Partially Corrected	2
• Risk Accepted	5
• Acknowledged	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	7a0535ee07815a2c8604d58469393c56a62d5b81	Initial Version - v1.0.0-beta.0
2	26 Sep 2024	2087250fc5988d3f0117f23e3f959f9423d04909	Second Version
3	03 Oct 2024	c76b422b053dd055aeb2cd555acfe353f05b316e	Fixes And Deployment Scripts - v1.0.0-beta.1
4	07 Oct 2024	342fe537b020ffa8ea7fcedf166b59b7ede21232	Fix Deployment Scripts
5	08 Oct 2024	52deda866ec8abdeaae9ace8574457d3e4209c36	Setting USDS And sUSDS Rate Limits
6	22 Oct 2024	6058f68f79520eb06ea8eded146da13039c47525	Bump Version - v1.0.0
7	29 Nov 2024	f81a7366f339d806a07a992c3aef2afe9a063e13	Aave And Ethena Integrations - v1.1.0-beta.0
8	06 Dec 2024	ad4391c37aa262d3c578a757700c1b6e86a96060	Withdrawals Rate Limit - v1.1.0-beta.1
9	13 Dec 2024	2bb2680893aa3e42210c8f907ec4d5778ace9fe6	Release v1.1.0
10	30 Dec 2024	eb8192199ee7713b583dd1b4920ec156c2333830	Updated Init Scripts - v1.2.0-beta.0
11	30 Jan 2025	6a35adaddc38666ae2f75f0d4792f9f526de0cab	RWAs And Morpho Integrations - v1.3.0-beta.0
12	25 Mar 2025	3de54c3f91d219401406459ac66e4b9b1b9fce44	Release v1.3.0
13	10 Mar 2025	3dd606c2cd4a5f06a726b059397202dc48e47165	v.1.4.0-beta.0
14	03 Apr 2025	b618c3fb1508ce29ac9e92e3bf55229cd1a66d8a	v1.4.0

15	30 May 2025	9f24f17f9dbd79e55e556b8ad29aa8747c4f2297	v1.5.0-beta.0
16	11 July 2025	38da45689cacc6bb402045bb3a577364bd3d8e33	v1.5.0
17	14 Aug 2025	c0292396956dabfdfcf7fce2f21b2438d7a41f3b	v1.6.0-beta.0
18	18 Aug 2025	83defcbd38280e2279c1bbd42409e06ce05ebf6c	Updated Readme
19	27 Aug 2025	cc8b6e30e74ad1d5eef554df17d88bc4d1cb4a00	v1.7.0-beta.1
20	03 Sep 2025	8d06c0dffc0310b6f6e1d243077408d060589635	v1.7.0

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
```

In **Version 15** the following files were added to the scope:

```
src/
    interfaces/ILayerZero.sol
    libraries/
        CCTPLib.sol
        CurveLib.sol
        PSMLib.sol
```

2.1.1 Excluded from scope

All other files are out of scope.

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 integrates with are out of scope and assumed to work honestly and correctly as documented.

For the curve integration, it is assumed only the Stableswap-NG Plain pools will be used.

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

2.2 System Overview

This system overview describes the latest version of the contracts as defined in the Assessment Overview.

At the end of this report section, we have added a changelog section for 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. For simplicity of notation, *admin*, referring to the *DEFAULT_ADMIN_ROLE* on respective contracts, will be used in following sections.

2.2.1 ALMProxy

The ALMProxy provides the following privileged functions that are restricted to addresses with the *CONTROLLER* role (denoted as *controller*):

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.

In (**Version 3**), function `receive()` has been added to the `ALMProxy` to support receiving ETH.

2.2.2 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 *admin* with `setRateLimitData()` (two signatures available) or `setUnlimitedRateLimitData()`.

The following functions are introduced to update the rate limit by the *controller*:

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`.

Note both functions will revert in case the rate limit is not configured (`RateLimitData.maxAmount==0`).

Library RateLimitHelpers is provided to construct keys depending on different inputs:

- `makeAssetKey()`: computes the key based on an asset.
- `makeAssetDestinationKey()`: computes the key based on an asset and a destination address.
- `makeDomainKey()`: computes the key based on a domain.

2.2.3 Controllers

MainnetController and **ForeignController** are implemented that define operations in the context of respective ALMProxy. They share a similar structure but feature some different third-party integrations and operations.

For emergency response, both controllers implement function `removeRelayer()`, enabling the *FREEZER* role (denoted as *freezer*) to directly remove a compromised *relayer*. Each controller is intended to have a main *relayer* and a configured backup. If all *relayers* are removed by *freezer*, the contract becomes effectively frozen. To resolve such a situation, the *admin* can grant the *RELAYER* role (denoted as *relayer*) again.

2.2.3.1 Existing Integration

This section aims to provide an overview of integrations in the system. Note that most functions are restricted to the *relayer* (only exceptions are explicitly mentioned).

Arbitrary Token Transfer: *introduced in* (**Version 11**); integration in **both** controllers (foreign controller since (**Version 19**)).

`transferAsset()`: ERC-20 assets can be transferred out manually using this function respecting the rate limit including the destination address. Note that asset-destination pairs are rate limited. Further, the function is intended to be used for integration with other systems where direct token transfers are needed (e.g. BUIDL minting, Spark Vaults V2).

DSS Allocator: *introduced in* Version 1; only in **mainnet** controller.

`mintUSDS()` / `burnUSDS()` leverage AllocatorVault to mint or burn (`draw()` or `wipe()`) USDS. Minting is rate limited while burning performs cancelling of the rate limit.

Dss LitePSM: *introduced in* Version 1; only in **mainnet** controller.

`swapUSDSToUSDC()` / `swapUSDCToUSDS()` leverage the PSM to swap between USDS and USDC without fees (`buyGemNoFee()` and `sellGemNoFee()`) and are subject to rate limits where swapping back to USDS cancels the rate limit.

Spark PSM: *introduced in* Version 1; only in **foreign** controller.

`depositPSM()` / `withdrawPSM()` enable depositing / withdrawing specific assets to / from Spark PSM with respect to separate rate limits.

Circle CCTP: *introduced in* Version 1; in **both** controllers.

1. `setMintRecipient()`: restricted to the *admin*; sets the token recipient of a destination domain when bridging USDC with CCTP (Circle's Cross-Chain Transfer Protocol).
2. `transferUSDCToCCTP()`: leverages CCTP to bridge USDC to a recipient (expected to be another ALMProxy) on a foreign domain which is subject to rate limits. Additionally, there is a rate limit for all CCTP transfers.

ERC-4626: *introduced in* Version 7; in **both** controllers.

`depositERC4626()` / `withdrawERC4626()` / `redeemERC4626()` enables integrations with ERC-4626 vault. Deposits / withdrawals should respect the respective rate limit on each vault, the shares minted / assets withdrawn will be credited to the ALMProxy. Since Version 17, the deposit rate limit will be replenished during a withdrawal or redemption, which is subject to the cap (`maxAmount`) of the limit and will revert if the limit is not configured (`maxAmount==0`).

Aave: *introduced in* Version 7; in **both** controllers.

`depositAave()` / `withdrawAave()` enables integrations with ERC-4626 vault. Deposits / withdrawals should respect the respective rate limit on each `aToken`, and the newly minted `aToken` / underlying withdrawn will be credited to the ALMProxy. Note that both functions are separately rate limited per aToken. Since Version 17, the Aave deposit rate limit will be replenished during a withdrawal, which is subject to the cap (`maxAmount`) of the limit and will revert if the limit is not configured (`maxAmount==0`).

Ethena: *introduced in* Version 7; only in **mainnet** controller.

1. `setDelegatedSigner()` / `removeDelegatedSigner()`: enables configuring any amount of delegated signers. 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.
2. `prepareUSDeMint()` / `prepareUSDeBurn()`: to mint or burn USDe, an approval can be triggered from the ALMProxy to the Ethena minting contract. Further operations from the delegated signers, Ethena minter and burners are required to complete the action.
3. 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.
4. `cooldownAssetsSUSDe()` / `cooldownSharesSUSDe()` : initiates a withdrawal of sUSDe that requires a cooldown period followed by an `unstake()` call since the instant withdrawal of sUSDe is currently blocked. These functions burn the shares and credit the USDe to the USDeSilo contract.

5. `unstakeSUSDe()`: enables the exit of USDe to the ALMProxy once the `cooldownDuration` is reached.

Note that minting, burning and the cooldown are separately rate limited.

Morpho: *introduced in* Version 11; only in **foreign** controller.

`setSupplyQueueMorpho()` / `updateWithdrawQueueMorpho()` / `reallocateMorpho()` allows 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 Minting: *introduced in* Version 11; only in **mainnet** controller.

`subscribeSuperstate()` allows minting USTB with USDC. Function is subject to rate limits and approvals are granted before calling respective functions on Superstate.

Note that redemptions are not supported but USTB can be transferred if necessary.

Maple: *introduced in* Version 11; only in **mainnet** controller.

1. Deposits happen through the existing ERC-4626 deposit function.
2. `requestMapleRedemption()` / `cancelMapleRedemption()`: withdrawals are not ERC-4626 compliant and work based on redemption requests, which can also be cancelled. 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.

Note that rate limits apply to all functions, respectively.

BlackRock BUIDL Minting: *introduced in* Version 11; integration in **mainnet** controller.

Note that this integration is with the arbitrary token transfers (`transferAsset()`). BUIDL tokens are expected to be issued by privileged roles to the ALMProxy.

Centrifuge: only in **mainnet** controller.

1. `requestDepositERC7540()`: *introduced in* Version 11; allows to initiate a call to `requestDeposit()` where assets are transferred to an escrow.
2. `claimDepositERC7540()`: *introduced in* Version 11; allows to finalize the deposit and claim all the available shares once the deposit request is fulfilled.
3. `requestRedeemERC7540()`: *introduced in* Version 11; initiates a withdrawal with shares transferred to escrow.
4. `claimRedeemERC7540()`: *introduced in* Version 11; finalizes the redeem and retrieves the max withdrawable assets.
5. `cancelCentrifugeDepositRequest()` / `cancelCentrifugeRedeemRequest()`: *introduced in* Version 12; allows to cancel an unfinalized deposit / redeem request.
6. `claimCentrifugeCancelDepositRequest()` `claimCentrifugeCancelRedeemRequest()`: *introduced in* Version 12; allows to finalize the cancel deposit / redeem and claim the available assets / shares once the request is fulfilled.

Note that request operations are rate limited individually per token while other operations require that the respective rate limit exists.

Curve StableSwapNG: *introduced in* Version 13; only in **mainnet** controller.

1. `setMaxSlippage()`: the *admin* can configure maximum slippage for the whitelisted pools.
2. `swapCurve()` / `addLiquidityCurve()` / `removeLiquidityCurve()`: enables swaps and liquidity management respecting the defined slippage and rate limits.

DaiUsds: *introduced in* Version 13; only in **mainnet** controller.

`swapDAIToUSDS()` / `swapUSDSToDAI()` support converting DAI to USDS and vice versa using the DAI-USDS converter without rate limits.

LayerZero: *introduced in* Version 15; in **both** controllers.

1. `setLayerZeroRecipient()`: the *admin* can set the `LayerZeroRecipient` of each `destinationEndpointId`.
2. `transferTokenLayerZero()`: triggers a `send()` operation on an OFT to bridge the OFT supported token to a `destinationEndpointId`. Fees are estimated and attached and rate limit per OFT and `destinationEndpointId` is respected.

SPK Farm: *introduced in* Version 17; only in **mainnet** controller.

1. `depositToFarm()`: deposits USDS into the farm, limited by the rate limit configured for this farm and asset.
2. `withdrawFromFarm()`: withdraws USDS which at the same time claims the rewards for the ALMProxy.

Spark Vaults V2: *introduced in* Version 19; in **both** controllers.

`takeFromSparkVault()` enables taking assets from SparkVault. The function is rate limited per SparkVault. Note that funds can be returned with `transferAssets()`.

2.2.3.2 Deprecated Integration

This section aims to provide an overview of deprecated (i.e. removed) integrations in the system. Note that most functions are restricted to the *relayer* role (only exceptions are explicitly mentioned).

SUSDS: *exists from* Version 1 *to* Version 7 *(exclusive)*; only in **mainnet** controller.

`depositToSUSDS()` / `withdrawFromSUSDS()` / `redeemFromSUSDS()` wraps USDS to SUSDS or vice-versa. They are replaced with the ERC-4626 functions.

Morpho integration on **mainnet** controller *exists from* Version 11 *to* Version 13 *(exclusive)*. See the Morpho integration description above for more details.

BlackRock BUIDL Redemptions: integration in **mainnet** controller. *exists from* Version 11 *to* Version 15 *(exclusive)*.

`redeemBUIDLCircleFacility()` enables initiating `buildRedeem.redeem()` from the ALMProxy, expecting to return USDC in exchange for the `buildRedeem.asset`. Note that the function is rate limited.

Superstate redemptions: integration in **mainnet** controller. *exists from* Version 11 *to* Version 19 *(exclusive)*.

`redeemSuperstate()` allows redeeming USTB for USDC. Function is subject to rate limits and approvals are granted before calling respective functions on Superstate.

2.2.4 Deployment Scripts

Deployment scripts have been added that can be used in governance spells.

Since Version 3, 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.

Since Version 10, both MainnetControllerInit and ForeignControllerInit support the functions `initAlmSystem()` and `upgradeController()` for initialization:

- `initAlmSystem()` corresponds to the initialization of the initial deployment where sanity checks are performed, contracts are wired to each other, and roles and CCTP mint recipients are configured. (previously: `subDaoInitFull()` / `init()` with the previous controller being `0x0` in Version 3)
- `upgradeController()` corresponds to a controller upgrade where the previous controller (not being `0x0`) is removed. (previously: `subDaoInitController()` / `init()` with the previous controller not being `0x0` in Version 3).

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

2.2.5 Changelog

In Version 2:

1. `transferUSDCToCCTP` function 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`.

In Version 3:

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.

In Version 4, on the foreign controller's initialization, limits for USDS and sUSDS are set up for PSM deposits and withdrawals.

In Version 7, the integration with any ERC-4626 compliant vault, Aave V3, and Ethena was added. As a consequence of adding the generic ERC-4626 support, the sUSDS-only integration was deprecated.

In Version 8, 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.

In Version 10, the init script library has been refactored:

1. `pauseProxyInit()` introduced in Version 3 is deprecated now; before it is intended 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 has been performed by the SubDAO.
2. 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.

In Version 11:

1. Morpho, Superstate, Maple, BlackRock BUIDL, and Centrifuge (ERC7540) integrations are added.
2. A new *freezer* behavior is introduced in both controllers. The contract no longer includes an active flag that the *freezer* could toggle to pause the contract. More specifically, the *freezer* role could pause all operations (in a controller) with `freeze()` while the *admin* could resume operations with `reactivate()`. This was intended as a safeguard against a compromised *relayer*, allowing the *admin* to replace them through governance which is a slow process. The new implementation

replaces `freeze()`/ `reactivate()` with `removeRelayer()`, 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*, the contract becomes effectively frozen.

3. Library RateLimitHelpers (for deployment scripts) has been extended with custom errors and new functions `makeAssetDestinationKey()`, `unlimitedRateLimit()` (a helper function returning the maximum amount with a slope of 0), and `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.

In (Version 12), Centrifuge deposit / redeem request cancellation support is added.

In (Version 13), Curve-Stableswap-NG-Plain pool integration and DaiUsds conversion are added, and Morpho support has been removed from MainnetController (while it still exists in ForeignController).

In (Version 14), after asking SparkDAO about swap rate limit decreases during adding liquidity, SparkDAO introduced rate limit decrease for swaps on Curve when liquidity is added in an imbalanced way.

In (Version 15):

1. LayerZero integration is added.
2. Internal `_approve()` is modified to achieve similar functionality as `forceApprove()` where the allowance is reset to 0 first.
3. `redeemBUIDLCircleFacility()` is removed.
4. The CCTP, DSS LitePSM, and Curve integration are extracted to dedicated libraries.
5. Helper functions `unlimitedRateLimit()` and `setRateLimitData()` are removed.
6. The initialization script now allows to configure an array of *relayers*.

In (Version 16), function `transferTokenLayerZero()` now supports a wider range of OFTs that require approval prior to `send()`. And it is modified to be payable, hence the *relayers* or the controller itself (if it holds native token) pay the bridging fee. Further, `setLayerZeroRecipient()` is called in the deployment scripts of the Mainnet and Foreign Controllers to set the peer addresses on respective destination issued.

IN (Version 17). SPK Farm integration has been added to the mainnet controller. Further, for both the Mainnet and Foreign controller functions `withdrawERC4626()`, `redeemERC4626()` and `withdrawAave()` have been changed to replenish the deposit rate limit of the asset for the amount withdrawn.

In (Version 19):

1. The integration with Spark Vaults V2 has been added.
2. The `immutable` and `constant` keywords have been removed from MainnetController to reduce bytecode size.
3. The `redeemSuperstate()` function has been removed as it was not used in production.
4. `transferAsset()` has been added to the foreign controller so that the foreign controller can return taken funds to the Spark Vaults V2.

2.3 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 DSS LitePSM.
2. It requires `wards` role on AllocatorVault to `draw()` and `wipe()` USDS.
3. It needs sufficient allowance from AllocatorBuffer to move minted USDS.

MainnetController and ForeignController:

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

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

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

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

- 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.
- Spark ALM Controller is subject to the inherent risks of these protocols (i.e. risks of upgradeability, RWAs, governance ...) and generally the third party protocols receiving funds are assumed to be non-malicious.

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

Aave and Aave-like protocols: 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. Similarly, note that other Aave-like protocols such as Spark are trusted in the same way if support for the respective LP tokens is added.

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)
4. 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.
5. 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.

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.

Curve StableswapNG pools:

1. The tokens in the pool are pegged. Otherwise, swaps (or add imbalanced liquidity) to the depegged tokens are allowed by the controller, hence incur a loss to the ALM Proxy.
2. The *admin* will set up the slippages properly (`setMaxSlippage()`). Otherwise, curve-related functions could be DoSed if it is `>100%`, or significant slippage could be allowed if it is too small.

LayerZero:

LayerZero is out of scope for this review, it is assumed to work as documented, and trusted to behave correctly, and deliver messages to the correct destination:

1. The OFT and recipient of each `destinationEndpointId` configured are fully trusted, otherwise, it can steal the funds being transferred, or upgrade and bridges unexpected tokens. The configuration required for managing the OFT on multiple chains is considered out of scope and assumed to be correct.
2. The LayerZero executor is trusted to always deliver messages with the correct provided options. However in case LayerZero behaves incorrectly, for example due to a compromise, an L2 finality issue, or any other failure the rate limit functionality caps the amount of funds at risk.

SparkVault:

The SparkVault is expected to implement the `take` function and transfer the requested assets to the ALMProxy.

3 Limitations and use of report

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

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

4 Terminology

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

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

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

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

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

5 Open Findings

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

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

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

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	8

- Lack of Rate Limits Code Partially Corrected Acknowledged
- LayerZero Approvals Code Partially Corrected Acknowledged
- Centrifuge Deposit / Redemption Can Be DoSed by Cancellation Risk Accepted
- Centrifuge Tranche Token Price May Change Between Request Submission and Execution Risk Accepted
- Maple Redemption Can Be DoSed Risk Accepted
- Over-reduced Limit in Maple Redemption Risk Accepted
- Relayer Can DoS SUSDE Unstaking Risk Accepted
- Revoking Unused Approval Acknowledged

5.1 Lack of Rate Limits

Design Low Version 15 Code Partially Corrected Acknowledged

CS-SPRKALM-021

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

Further, the LayerZero integration might pay fees in native tokens. However, the outbound amount of native tokens is not rate limited. Thus, a *relayer* colluding with an endpoint operator could drain the native token balance.

Code partially corrected and acknowledged:

In Version 16, function `transferTokenLayerZero()` has been marked as payable and will attach exactly `fee.nativeFee` when calling the ALMProxy. Hence the *relayers* or the controller itself (if holds any native tokens) will pay the bridging fee. SparkDAO has acknowledged the inconsistency between the bridging rate limits for CCTP and LayerZero.

5.2 LayerZero Approvals

Correctness Low Version 15 Code Partially Corrected Acknowledged

CS-SPRKALM-022

The `MainnetController.transferTokenLayerZero()` always approves the OFT while `ForeignController.transferTokenLayerZero()` never does. While that implementation supports the USDT implementation of OFTs, it may not support other implementations. Namely, not all L1 OFTs require approvals and some L2 OFTs require approvals. As a consequence, unnecessary approvals could be given while necessary approvals are not given. Note that the OFT standard defines `IOFT.approvalRequired()` which defines whether an approval should be given.

Ultimately, the controllers could consider `IOFT.approvalRequired()` to support a wider range of OFTs.

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

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

To summarize, a wider range of OFTs could be supported and dangling approvals could nonetheless exist.

Code partially corrected and acknowledged:

In Version 16 the getter `approvalRequired()` is queried to determine either an approval should be granted prior to `send()`, hence a wider range of OFTs are now supported. SparkDAO has acknowledged the dangling approvals could still exist.

5.3 Centrifuge Deposit / Redemption Can Be DoSed by Cancellation

Security Low Version 12 Risk Accepted

CS-SPRKALM-010

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.

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

Correctness | Low | Version 11 | Risk Accepted

CS-SPRKALM-011

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

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

5.5 Maple Redemption Can Be DoSed

Security | Low | Version 11 | Risk Accepted

CS-SPRKALM-012

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

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

Risk accepted:

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

5.6 Over-reduced Limit in Maple Redemption

Correctness | Low | Version 11 | Risk Accepted

CS-SPRKALM-014

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

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

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

5.7 Relayer Can DoS SUSDE Unstaking

Security Low Version 7 Risk Accepted

CS-SPRKALM-016

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 relayer can keep triggering new cooldowns with as little as 1 wei asset to block previous exits from sUSDe to USDe, hence DoS the sUSDe to USDe conversion.

Note: 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*.

5.8 Revoking Unused Approval

Design Low Version 7 Acknowledged

CS-SPRKALM-015

The *freezer* can remove *relayers* from the MainnetController which prevents *relayers* from triggering any more interactions or funds transfers from the ALMProxy.

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

Note that prior to Version 11, the issue was reported in regards to the freezing with `freeze()`.

Acknowledged:

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

6 Resolved Findings

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

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

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	6

- Withdrawal May Be Blocked if Deposit Limit Is Reset **Specification Changed**
- Incorrect Approval Return Decoding **Code Corrected**
- Incorrect Slippage Protection for Curve **Code Corrected**
- Rounding Error in Slippage Calculation **Code Corrected**
- Outdated README **Specification Changed**
- Loosely Restricted Specific Calls to Arbitrary Address **Code Corrected**

Informational Findings	7

- Mint Recipient Is Not Initialized **Code Corrected**
- Sanity Checks In Deployment Scripts **Code Corrected**
- Unused Parameters **Code Corrected**
- ALM Proxy Cannot Receive **Code Corrected**
- Constructor Parameters **Code Corrected**
- Events **Code Corrected**
- Gas Optimizations **Code Corrected**

6.1 Withdrawal May Be Blocked if Deposit Limit Is Reset

Design **Low** **Version 17** **Specification Changed**

CS-SPRKALM-025

In this version, for both the Mainnet and Foreign controllers functions `withdrawERC4626()`, `redeemERC4626()` and `withdrawAave()` have been changed to replenish the deposit rate limit of the asset for the amount withdrawn.

However, this also implies withdrawals can only be made if there is an active deposit limit (`maxAmount!=0`), otherwise the call to increase deposit rate limit will revert due to the following check in contract RateLimits:

```
require(maxAmount > 0, "RateLimits/zero-maxAmount");
```

Consequently, in case deposit is blocked by resetting its rate limit (namely `maxAmount==0`), withdrawal will also be accidentally blocked.

Specification Changed:

The README has been updated to clarify the following requirement:

> Withdrawals using *withdrawERC4626/redeemERC4626/withdrawAave* must always have a non-zero deposit rate limit set for their corresponding deposit functions in order to succeed.

6.2 Incorrect Approval Return Decoding

Correctness **Low** **Version 15** **Code Corrected**

CS-SPRKALM-023

SparkDAO self-reported an issue about `_approve` function in the controllers and the curve lib.

Namely, the following code intended to retrieve the boolean return value from the low-level call:

```
( bool success, bytes memory data )
    = address(proxy).call(abi.encodeCall(IALMProxy.doCall, (token, approveData)));

// Decode the first 32 bytes of the data, ALMProxy returns 96 bytes
bytes32 result;
assembly { result := mload(add(data, 32)) }
```

However, the low-level call's `data` to be a pair of length of the data and the boolean returned by the call of the proxy. However, note that the proxy returns `bytes`. Thus, the `data` wraps around the returned `bytes` so that the boolean has a 64 byte offset in `data`.

Code corrected:

SparkDAO has corrected the problem accordingly.

6.3 Incorrect Slippage Protection for Curve

Correctness **Low** **Version 13** **Code Corrected**

CS-SPRKALM-008

When adding/removing liquidity the amount minted and burned in form of LP tokens are slippage protected. The maximum slippage check when adding/removing liquidity will assume that LP tokens have a value of 1 for all times. However, note that this assumption may be incorrect.

Note that while the below will focus on adding liquidity, the idea is similarly applicable to the removal of liquidity.

Within `addLiquidityCurve()`, the following condition on `minLpAmount` is required:

```
minLpAmount >= valueDeposited * maxSlippage / 1e18
```

The comparison implemented

- either compares an LP amount with "value" (e.g. USD) which is generally unsuitable,

- or assumes a one-to-one redeemability of LP tokens with "value" which is an incorrect assumption. Note that the Curve Documentation specifies the following for `get_virtual_price()`:

  ```
  Getter for the current virtual price of the LP token, which represents a price
  relative to the underlying.
  ```

As a consequence,

- the minimum `minLpAmount` must be set larger than necessary which might lead to reverts,
- or the `maxSlippage` configuration will be required to account for the share price which would, however, affect the slippage for swaps.

Note that the documentation's definition is in accordance with the implementation of `add_liquidity()`. The function mints as follows:

```
mint_amount = unsafe_div(total_supply * (D1 - D0), D0)
```

where `D0` and `D1` correspond to the prior and new stableswap invariant, respectively.

Note that `D` value corresponds to the "total effective value". Hence, the amount minted corresponds the pro rata shares according to the value added. However, also note that `total_supply/D0` corresponds to `1/get_virtual_price()`. Hence, that this is in accordance with the documentation.

Ultimately, the comparisons implemented for adding and removing liquidity compare two values with different meanings.

Code corrected:

`get_virtual_price()` is now being used to correctly compute the values required for the slippage computation.

6.4 Rounding Error in Slippage Calculation

Correctness Low Version 13 Code Corrected

CS-SPRKALM-009

Support for Curve swaps and liquidity management were added in Version 13. A `maxSlippage` can be configured to prevent bad operations on a pool. Whereas the slippage checks are subject to rounding errors in the following cases.

swapCurve(): The `minimumMinAmountOut` is rounded down as follows:

```
uint256 minimumMinAmountOut = (amountIn * rates[inputIndex] /
    rates[outputIndex]) * maxSlippage / 1e18;
```

- The rounding error in the first division will be amplified in the next multiplication. In addition, Assuming `amountIn * rates[inputIndex] * maxSlippage` will result in a value with 54 decimals, `uint256` should have sufficient precision to store the result without overflow in practice.
- If the `amountIn` is small enough, the `minimumMinAmountOut` can be rounded down to zero, especially when `tokenIn` has more decimals than `tokenOut`.

addLiquidityCurve(): The following slippage computation is rounded down (possibly to zero) similarly as `swapCurve()`, which is subject to the same error amplification.

```
uint256 valueDeposited;
for (uint256 i = 0; i < depositAmounts.length; i++) {
    _approve(curvePool.coins(i), pool, depositAmounts[i]);
    valueDeposited += depositAmounts[i] * rates[i] / 1e18;
}

require(
    minLpAmount >= valueDeposited * maxSlippage / 1e18,
    "MainnetController/min-amount-not-met"
);
```

removeLiquidityCurve(): Similarly the following slippage computation is rounded down (possibly to zero).

```
require(
    valueMinWithdrawn >= lpBurnAmount * maxSlippage / 1e18,
    "MainnetController/min-amount-not-met"
);
```

General:

Further, in all three operations (`swapCurve()`, `addLiquidityCurve()` and `removeLiquidityCurve()`) the rate limit decreases are slightly under-estimated due to the decreases being rounded down.

Code corrected:

While still some rounding errors could occur, the code has been improved.

6.5 Outdated README

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

CS-SPRKALM-013

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* permissions has not been updated and still reflects the previous behavior. Technically by removing all *relayers*, the *freezer* can still "freeze all actions".

Specification 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* description has been updated.
- The `TransferAsset` function is now described in the README.

6.6 Loosely Restricted Specific Calls to Arbitrary Address

Security | Low | Version 7 | Code Corrected

CS-SPRKALM-020

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.

6.7 Mint Recipient Is Not Initialized

Informational | Version 3 | Code Corrected

CS-SPRKALM-007

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.

6.8 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.

6.9 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.

6.10 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.

6.11 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`.

6.12 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.

6.13 Gas Optimizations

Informational **Version 1** **Code 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.

Code corrected:

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

7 Informational

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

7.1 `msg.value` Validation in `transferTokenLayerZero`

Informational **Version 16** **Acknowledged**

CS-SPRKALM-024

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

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

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

7.2 Maple Manual Withdraw May Be Enabled

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

CS-SPRKALM-017

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.

7.3 Allowance For Ethena Minter May Not Be Consumed

Informational | Version 7 | Acknowledged

CS-SPRKALM-018

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

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

- The delegated signers sign orders with smaller volume which do not consume all the allowance.
- The Ethena minter or redeemer 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.

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

Informational | Version 7 | Acknowledged

CS-SPRKALM-019

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

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

As a consequence, the following theoretical attack is possible:

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

Note that there is no rate limit and token restriction on function `withdrawAave()`. The *relayer* can withdraw the full balance of the asset with `LTV==0`, which resets the `usingAsCollateral` flag to `false` and recovers the ALMProxy from the DoS.

Acknowledged:

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.

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 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**

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

8.3 Asynchronous Operations May Be Interfered

Note **Version 13**

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

The *relayers* should be careful of the asynchronous operations and avoid the interference of different operations.

8.4 Avoid Morpho Deposit Into Market With Bad Debt

Note **Version 11**

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

8.5 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. The *relayers* should be careful of the potential cap changes in the future.

8.6 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.

As of **Version 15**, the redemption feature has been removed. However, the note is left is for completeness reasons for earlier deployments.

8.7 Curve Withdrawal Slippage

Note **Version 13**

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

As a consequence, note that the *relayer* will typically be forced to simulate the transaction to be able to provide rough values suitable to pass the check.

8.8 Inconsistent Swap Rate Limit Decrease for Curve

Note **Version 14**

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

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

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

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

8.10 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.

8.11 OFT Considerations

Note **Version 15**

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

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

8.12 OFTs With Mandatory Fee in lzToken Are Not Supported

Note Version 15

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

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

8.13 Special Cases Handling

Note Version 1

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

1. Assume it is desired that for an L2, all funds are bridged back to mainnet. However, in case the PSM3 never holds sufficient USDC to bridge back to L1, funds will remain on L2. As a result, another controller could be whitelisted that initiates redeeming the PSM shares against the other two assets to then bridge them back to mainnet through the respective bridges.
2. The mint recipient for CCTP could be blacklisted. That effectively could DoS the USDC bridging. In that case, a new controller could be added that allows calling CCTP's `replaceDepositForBurn` to resolve the issue.

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