PUBLIC

Code Assessment

of the Spark Vaults

Smart Contracts

January 24, 2025

Produced for

SPARK

by

CHAINSECURITY

Contents

1 Executive Summary 3

2 Assessment Overview 5

3 Limitations and use of report 10

4 Terminology 11

5 Open Findings 12

6 Resolved Findings 13

7 Informational 14

8 Notes 15

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 Vaults according to Scope to support you in forming an opinion on their security risks.

SparkDAO implements an ERC-4626 USDC Vault wrapping interactions with the PSM and Savings USDS allowing users to deposit USDC and earn yield from the Sky savings rate.

The most critical subjects covered in our audit are functional correctness, security of the vault's assets, and the proxy/upgradabilitiy pattern. Security regarding all the aforementioned subjects is high.

In summary, we find that the codebase provides a high level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered, and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

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

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 Vaults 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	13 January 2025	49ed49b54e42d68a29a885c2f5fc996c749b4c97	Initial Version
2	17 January 2025	490f4d5d3a99116e8f22dc1b3037b10b6cf6f782	USDCVaultL2
3	23 January 2025	34ff49e1f81e4d5bb75a9905a914dcc69642885e	After intermediate report

For the solidity smart contracts, the compiler version 0.8.21 were chosen.

The following files are in the scope of this review:

```
src/UsdcVault.sol
deploy/UsdcVaultDeploy.sol
deploy/UsdcVaultInstance.sol
```

In Version 2 the following files have been added:

```
src/UsdcVaultL2.sol
deploy/UsdcVaultL2Deploy.sol
```

2.1.1 Excluded from scope

Any file not explicitly listed above as well as third-party libraries are out of the scope of this review.

2.2 System Overview

This system overview describes the latest received version (Version 2) of the contracts as defined in the Assessment Overview.

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 a USDC Vault, an ERC-4626 compliant wrapper for interactions with the PSM and Savings USDS (sUSDS). Users deposit USDC which is converted to sUSDS to earn yield from the Sky Savings Rate (SSR). In return, users receive "Spark USDC Vault" (sUSDC) shares representing their position. Withdrawals reverse this process.

This ERC-4626 Vault issues vault shares that correspond to sUSDS shares at a 1:1 exchange ratio. Spark USDC Vault (sUSDC) shares have all the standard ERC-20 functions (`transfer`, `transferFrom`, `approve`) implemented. The ERC-20 features are extended with ERC-2612 (Permit Extension) and further ERC-1271 (Standard Signature Validation for Contracts). Two entry points for the permit functionality are available allowing to pass either the aggregated signature as bytes or `v`, `r` and `s` separately. Signatures for permits can either be from EOAs or, using ERC-1271, be validated by a contract that allows contracts to act as they "signed" permits.

An exit hatch, function `exit`, is implemented to allow redeeming vault shares for sUSDS directly at a 1:1 exchange rate. While primarily intended for use when the PSM lacks sufficient liquidity to convert to USDC, this function is usable at all times.

Two distinct Vault implementations exist for Ethereum Mainnet and Layer 2 networks due to differences in the PSM used. On Mainnet, the Vault uses the UsdsPsmWrapper to convert USDS to DAI, facilitating exchanges via the fee-incurring DssLitePsm. On Layer 2 networks, the Vault integrates with the fee-free PSM3 (Spark-PSM), which allows swapping USDC to sUSDS directly.

2.2.1 Usdc Vault on L1

This section describes the `UsdcVault` , intended to be deployed on L1.

The state-changing functions compliant with EIP-4626 are:

- `deposit(uint256 assets, address receiver)`: Converts the specified amount of USDC (assets) into USDS using the PSM, subject to a fee (tin). The resulting USDS is deposited into Savings USDS, earning sUSDS shares for the vault. Spark USDC Vault shares (sUSDC) equivalent to the amount of sUSDS are minted and sent to the receiver.
- `mint(uint256 shares, address receiver)`: Determines the amount of USDC (assets) required to mint the specified number of Spark USDC Vault shares (sUSDC). Transfers the calculated amount of USDC from the caller, converts it into USDS via the PSM (incurring a fee tin), and deposits the USDS into Savings USDS to earn sUSDS shares. Mints the specified amount of sUSDC and assigns it to the receiver.
- `withdraw(uint256 assets, address receiver, address owner)`: Withdraws the specified amount of USDC (assets) to the receiver. Calculates the amount of USDS needed to exchange for the desired amount of USDC via the PSM, including the fee (tout). Withdraws this amount of USDS from Savings USDS, burns the corresponding number of sUSDC shares from the owner, and completes the transfer of USDC. The owner can be the caller or an authorized account with sufficient allowance.
- `redeem(uint256 shares, address receiver, address owner)`: Redeems the specified number of Spark USDC Vault shares (sUSDC) into USDC. Converts the corresponding number of sUSDS shares into USDS, then exchanges the USDS for USDC using the PSM (subject to a fee tout). Burns the redeemed sUSDC shares and transfers the resulting USDC to the receiver. The owner can be the caller or an authorized account with sufficient allowance.

Both `deposit` and `mint` functions support a secondary entry point (non-ERC-4626 specific) with two additional parameters:

- a referral parameter for marking deposits originating from specific frontends, emitting a `Referral( uint16 indexed referral, address indexed owner, uint256 assets, uint256 shares)` event.
- a slippage protection parameter: `minShares` for deposit and `maxAssets` for mint.

View functions related to EIP 4626:

- `asset()`: returns the address of USDC.
- `totalAssets()`: returns the total assets held by the contract (sUSDS) converted to USDC, ignoring the fees charged by the PSM.

- `convertToShares(uint256 assets)`: calculates the number of shares the contract would exchange for the given amount of assets (USDC).
- `convertToAssets(uint256 shares)`: calculates the amount of assets (USDC) the contract would exchange for the given number of shares.
- `maxDeposit(address)`: returns the maximum amount that can be deposited, limited by the DAI balance of the PSM. On L1 USDC must first be converted into DAI via the PSM and then into USDS before being deposited into Savings USDS. Conversion between DAI and USDS is unrestricted.
- `previewDeposit(uint256 assets)`: Returns the number of shares that would be minted for the given amount of assets (USDC) at the current block.
- `maxMint(address)`: returns the maximum amount of shares that can be minted, limited by the DAI balance of the PSM as USDC must first be converted into DAI via the PSM and then into USDS before being deposited into Savings USDS. Conversion between DAI and USDS is unrestricted.
- `previewMint(uint256 shares)`: calculates the amount of assets (USDC) needed to mint the specified number of shares at the current block.
- `maxWithdraw(address owner)`: Returns the maximum amount of assets (USDC) the specified address can withdraw based on its balance. Withdrawals are blocked if the PSM is halted. Additionally, the available USDC liquidity in the PSM may limit withdrawals.
- `previewWithdraw(uint256 assets)`: returns the exact amount of shares that would be burned if the caller withdraws this amount of assets (USDC) in this block.
- `maxRedeem(address owner)`: returns `balanceOf[owner]`, the maximum number of shares the specified address can redeem (its current balance). Redemptions are blocked if the PSM is halted. Additionally, the available USDC liquidity in the PSM may limit redemptions.
- `previewRedeem(uint256 shares)`: returns the amount of assets (USDC) the caller would receive if he redeems this amount of shares in this block.

Further getters for sUSDS and PSM getters are exposed:

- `chi():` returns the rate accumulator of sUSDS
- `rho():` returns the timestamp of the last update of the rate accumulator.
- `ssr():` returns the sky savings rate.
- `tin():` returns the fee when exchanging USDC for USDS in the PSM.
- `tout():` returns the fee when exchanging USDS for USDC in the PSM.

2.2.2 L2

This section highlights the differences with the L1 version.

On Layer 2, since the SSR is not directly accessible, the rate is queried from a trusted RateProvider. The PSM is intended to be the PSM3 implementation. This PSM incurs no fee, allows direct swaps between sUSDS and USDC and provides preview functions for swaps.

The following list describes key differences only:

- `deposit(uint256 assets, address receiver)`: Converts the specified amount of USDC (assets), transferred from the caller, into sUSDS. Spark USDC Vault shares (sUSDC) equivalent to the amount of sUSDS received from the PSM swap are minted to the receiver.
- `mint(uint256 shares, address receiver)`: Determines the amount of USDC (assets) required to mint the specified number of Spark USDC Vault shares (sUSDC) using the preview function of the PSM. Transfers this amount of USDC from the caller, converts it into sUSDS via the PSM. Mints the specified amount of sUSDC and assigns it to the receiver.

- `withdraw(uint256 assets, address receiver, address owner)`: Withdraws the specified amount of USDC (assets) to the receiver. Queries the PSM to get the amount of sUSDS needed to exchange for the desired amount of USDC via the PSM. Burns the corresponding number of sUSDC shares from the owner. The owner can be the caller or an authorized account with sufficient allowance.
- `redeem(uint256 shares, address receiver, address owner)`: Redeems the specified number of Spark USDC Vault shares (sUSDC) into USDC. Converts the corresponding number of sUSDS shares into USDC using the PSM, the USDC assets are directly transferred to the receiver. Burns the redeemed sUSDC shares. The owner can be the caller or an authorized account with sufficient allowance.
- `maxDeposit(address)`: returns the maximum amount that can be deposited, limited by the sUSDS balance of the PSM.
- `maxMint(address)`: returns the maximum amount of shares that can be minted, limited by the sUSDS balance of the PSM.
- `maxWithdraw(address owner)`: Returns the maximum amount of assets (USDC) the specified address can withdraw. This is limited by the balance of the owner and the available USDC liquidity in the PSM.
- `maxRedeem(address owner)`: returns `balanceOf[owner]`, the maximum number of shares the specified address can redeem (its current balance). Additionally, the available USDC liquidity in the PSM may limit redemptions.

2.2.3 Upgradeability

UsdcVault inherits from Openzeppelin's UUPSUpgradeable which provides all functionality for UUPS Proxies implementation contracts to facilitate upgradeability. The implementation overrides `_authorizeUpgrade()` adding access control to restrict implementation upgrades by `wards` (assumed to be the Governance Pause proxy exclusively) only. The following permissioned functions, callable by addresses bearing the `ward` role, are available to update the wards mapping:

- `rely(address usr)`: Adds an address to the `wards` mapping. Emits the `Rely` event.
- `deny(address usr)`: Removes an address from the `wards` mapping. Emits the `Deny` event.

Furthermore, `getImplementation()` has been added returning the address of the current implementation which is retrieved from the defined storage slot.

For the UsdcVault Proxy the widely used OpenZeppelin implementation of ERC1967Proxy is used. All calls are executed as delegatecall to the implementation contract, the address of the implementation contract is stored at slot calculated as `bytes32(uint256(keccak256('eip1967.proxy.implementation')) - 1)`.

2.2.4 Deployment

Some EOA deploys the contracts (UsdcVault and a ERC1967Proxy) and switches the owner (adding the new owner as ward and removing oneself as ward) using the UsdcVaultDeploy/UsdcVaultL2Deploy scripts. On L1 the addresses of the UsdsPsmWrapper and SUSDS are retrieved from the hardcoded Chainlog address, on L2 the PSM address is provided as a parameter.

2.2.5 Roles & Trust Model

UsdcVault is expected to be deployed on Ethereum Mainnet with the PSM expected to be an UsdsPsmWrapper. UsdcVaultL2 is to be used with the PSM3 (Spark-PSM) on Layer 2 networks.

Wards: Privileged roles. Fully trusted. Expected to be the Spark Governance PauseProxy only. Addresses holding the ward role can update the implementation of the Proxy and hence change the behavior and state of the contract. Further wards can add/remove more wards.

Users: Users interacting with the public functions of the system. Untrusted.

Deployer: Executes the deployment scripts deploying the contracts. Untrusted; deployment must be verified before users interact / the frontend is activated.

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:

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	0

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	0
Informational Findings	2

- Unused Function Code Corrected
- No Fixed Solidity Version Code Corrected

6.1 Unused Function

Informational Version 2 Code Corrected

CS-SPVA-004

`UsdcVaultL2._divup` is unused and can be removed.

Code corrected:

The unused function has been removed.

6.2 No Fixed Solidity Version

Informational Version 1 Code Corrected

CS-SPVA-005

SparkDAO uses a floating pragma solidity ^0.8.21. Further, the `foundry.toml` file does not fix a compiler version. Contracts should be deployed with the same compiler version and flags that have been used during testing and audit. Fixing the compiler version helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.

Code corrected:

The compiler version has been fixed to 0.8.21.

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 Gas Optimizations

Informational **Version 1**

CS-SPVA-001

Gas savings can be made. Namely, `psm.tin()` and `psm.tout()` could be replaced by querying the `daiPsm` contract directly. Ultimately, the number of staticcalls could be reduced. Note that this issue non-comprehensive and other potential optimizations could exist.

7.2 Leftover Tokens

Informational **Version 1**

CS-SPVA-002

As documented in the internal helper functions, on L1, small amounts of USDS tokens may remain in the contract as "dust" due to rounding errors during the execution of `_doMint()` and `_doRedeem()`.

7.3 README

Informational **Version 2** **Specification Partially Changed**

CS-SPVA-003

The README.md currently describes the behavior of the UsdcVault for Ethereum Mainnet only, without any mention or details about UsdcVaultL2, which follows a different logic and utilizes a different PSM.

The original description for L1 is slightly imprecise: "by converting USDC to USDS using dss-lite-psm." While a minor detail, the Vault actually uses an intermediary USDSPSMWrapper to facilitate conversion between USDS and DAI, as dss-lite-psm only supports DAI. This operational distinction is worth noting, as the current phrasing could confuse users who might expect the Vault's PSM to be dss-lite-psm, rather than a wrapper, based on the README.

Regarding the section on sanity checks: it may be prudent to include a sentence emphasizing the importance of validating the deployed bytecode itself.

Specification partially changed:

While the README still outlines the L1 version, a high-level description is provided in a separate section.

Additionally, USDSPSMWrapper is now mentioned additionally.

Last, the emphasis on the deployed code is left out.

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 Deployment Verification

Note Version 1

Since deployment of the contracts is not performed by the governance directly, special care has to be taken that all contracts have been deployed correctly.

We therefore assume that the initcode, bytecode, traces and storage (e.g. mappings) are checked for unintended entries, calls or similar. This is especially crucial for any value stored in a mapping array or similar (e.g. could break access control which could lead to unexpected contract implementation upgrades and hence result in stealing of funds).

Additionally, it is of utmost importance that no allowance is given to unexpected addresses (e.g. SUSDS approval to arbitrary addresses could have been given in the constructor of the proxy).

8.2 Griefing Users

Note Version 1

Given the design of the PSM, a limited amount of funds is available for swapping. Hence, some transactions could fail if not sufficient funds are available. As a consequence, user transactions could be griefed by frontrunning them with PSM swaps to make the transactions revert. However, there is no incentive for a malicious party to create such a DoS due to fees. Nevertheless, users can always exit safely with `exit()`.

8.3 PSM Halting Is Less Strict

Note Version 1

Note that a `tin == WAD` (100% fee) is valid in the Lite PSM implementation. However, in the USDCVault implementation, this state is treated as halted. This is necessary in the vault to prevent computations like `_psmUsdsOutToGemInRoundingDown()` from resulting in a division by zero. Note that a 100% fee is unrealistic and thus not a relevant consideration.