PUBLIC

Code Assessment

of the Spark Vaults V2 Smart Contracts

October 01, 2025

Produced for

by

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 Vaults V2 according to Scope to support you in forming an opinion on their security risks.

SparkDAO implements SparkVault V2, a tokenized vault (ERC-4626) where interest is distributed based on a manually set Vault Savings Rate (VSR). Whitelisted takers, intended to be the Spark ALM Controller, can draw on the vault's funds to invest.

The most critical subjects covered in our audit are functional correctness, asset security, and implementation integrity. Security regarding all the aforementioned subjects is high, however there are some informational issues and notes to consider in the report, for example Asset Solvency or Circular Reinvesting.

The general subjects covered are operational considerations, code complexity and documentation. ERC-4626 preview functions may revert during low liquidity conditions, which follows standard interpretation but requires integrator awareness, see ERC-4626 Preview Functions Liquidity Check.

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	1
• Code Corrected	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 Vaults V2 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	22 Aug 2025	8f1442ad5094d1ddd48c95d390dee6cd6c164133	Initial Version
2	08 Sep 2025	7f1f11d2406308dab89ce3b285f9e6d9a8f36277	v1.0.0
3	01 Oct 2025	0a686ba2fcf874bc1542171a323779ba73ac2dc5	v1.0.1

For the solidity smart contracts, the compiler version `0.8.29` was chosen with EVM version set to `cancun`.

The files in scope were:

```
src/
    ISparkVault.sol
    SparkVault.sol
```

2.1.1 Excluded from scope

All other files and all dependencies are out of scope. The underlying tokens are out of scope. The usage of the contract is out of scope.

2.2 System Overview

This system overview describes the latest received version (Version 3) 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 implements a tokenized vault (ERC-4626), `SparkVault`, where interest is distributed based on a manually set Vault Savings Rate (VSR). Whitelisted takers, intended to be the Spark ALM Controller, can draw on the vault's funds to invest. `SparkVault` is deployed through an ERC-1967 proxy using the UUPS pattern. Note that the design is similar to Savings USDS (sUSDS).

2.2.1 Spark Vault V2

The `SparkVault` contract implements an ERC-4626 token where the underlying asset corresponds to a suitable ERC-20 token. The vault token uses the same decimals as the underlying asset. So-called takers can then use the balance held by the vault as liquidity.

The vault generates interest for the liquidity providers to incentivize them. Namely, the interest generated is defined by the `vsr` and is reflected in the exchange rate `chi` (shares to underlying in RAY). Thus, `chi` is a rate accumulator defined as $chi_i = chi_{i-1} * vsr^{t_i - t_{i-1}}$ where t_{i-1} corresponds to the latest update timestamp and is defined by `rho`, and where t_i corresponds to the current timestamp.

Note that the Vault Savings Rate (VSR) can be updated by anyone through `drip`. The exchange rate is automatically refreshed or simulated before any relevant ERC-4626 function is executed.

Note that a global deposit limit, `depositCap`, exists that limits the total underlying on deposit actions. That is reflected in both `maxDeposit` and `maxMint`. Similarly, withdrawals can be limited. This is due to the `TAKER_ROLE` being allowed to access the liquidity held by the vault. Accordingly, `maxWithdraw` and `maxRedeem` take the vault's current token balance into account.

Additionally, the following functionality is implemented:

- `deposit` and `mint` can be called with a referral code.
- `assetsOf` returns the assets a user would receive according to their current share balance.
- `assetsOutstanding` returns the assets owed to the contract (i.e. interest payments and taken amount repayments).
- `nowChi` simulates an update of `chi`.
- EIP-2612 is implemented for signed approvals. Additionally, `permit` is implemented to also allow passing `r,s,v` as a packed bytes array.
- `getImplementation` is a public getter to retrieve the implementation address.

Privileged addresses can call the following functions:

- `TAKER_ROLE`: Can call `take` to access the liquidity provided by LPs as outlined above.
- `SETTER_ROLE`: Can set the `vsr`. However, the value must be within the bounds set by governance.
- Governance / `DEFAULT_ROLE_ADMIN`: Can set the VSR bounds with `setVsrBounds`, assign any role and can upgrade the contract. Additionally, the role can set the deposit cap with `setDepositCap`.

Note that it is expected that no losses are made and that the interest as well as the taken amounts are eventually sent to the vault.

2.2.2 Changelog

In **Version 2**, the following changes were introduced compared to prior versions:

- The deposit cap has been introduced.
- The possibilities of circular reinvesting have been partially limited, see Circular Reinvesting.

Prior to **Version 3**, SparkVault was hardcoded to 18 decimals.

2.3 Trust Model

Below, the relevant roles for `SparkVault` are defined:

- `TAKER_ROLE`: Fully trusted. Can fully drain the contract with `take`.
- `SETTER_ROLE`: Partially trusted. Expected to set meaningful VSR values. However, restricted to the bounds set by governance.
- Governance / `DEFAULT_ROLE_ADMIN`: Fully trusted. Can upgrade the contract arbitrarily and can perform any action of any other role.
- Users: Untrusted.

- Tokens: The underlying token is expected to be a valid token that is non-rebasing, has no fees and is non-reentrant.

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:

- Design: Architectural shortcomings and design inefficiencies

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	1

- Unbounded Interest Obligations Code Corrected

6.1 Unbounded Interest Obligations

Design Low Version 1 Code Corrected

CS-SPK-VLTV2-001

The vault owes VSR interest on all deposited funds, including idle funds that haven't been taken by the taker. With no deposit limits, unexpectedly large deposits can create significant interest obligations for the Spark ecosystem.

Consider the scenario where a whale deposits 100M USDS, immediately earning 10% VSR (10M USDS annual interest obligation). The Spark ecosystem (e.g. ALM controller, lending) might not be able to generate sufficient yield for such large obligations.

Ultimately, the design assumes the broader Spark ecosystem can generate sufficient yield to cover all vault obligations. However, there's no direct link between vault deposits and yield generation. Without deposit limits or a direct link between vault deposits and yield generation, the vault can accumulate interest obligations that the ecosystem must cover regardless of actual yield performance. This issue compounds over time if the taker cannot deploy funds or the rate setter doesn't adjust the VSR.

Code corrected:

The code has been adjusted to introduce a deposit limit. Now, the combination of VSR and the deposit cap leads to predictable maximum amounts of interest obligations.

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 Code Consistency

Informational **Version 1**

CS-SPK-VLTV2-002

The codebase aims to not duplicate logic. However, `maxRedeem` unnecessarily duplicates logic where `convertToShares` could be reused.

Namely, the function implements

```
uint256 maxShares  = IERC20(asset).balanceOf(address(this)) * RAY / nowChi();
```

which would we be equivalent to

```
uint256 maxShares = convertToShares(IERC20(asset).balanceOf(address(this)))
```

Note that the similar function, `maxWithdraw` accordingly reuses the `assetsOf` function.

To summarize, functionality could be reused to keep the code style more consistent.

7.2 Lack of Events

Informational **Version 1**

CS-SPK-VLTV2-004

The `initialize` function sets state variables but does not emit the respective events. Below is a list of such occurrences where an initial event could be emitted:

1. `chi` and `rho` are set to initial values which can be interpreted as the initial `Drip`.
2. `minVsr` and `maxVsr` are set to `RAY` but `VsrBoundsSet` is not emitted.
3. `vsr` is set to `RAY` but `VsrSet` is not emitted.

7.3 Library Initializers Unused

Informational **Version 1**

CS-SPK-VLTV2-005

The internal initializers for `AccessControlEnumerableUpgradeable` and `UUPSUpgradeable` are not used in `initializer` (e.g. `__AccessControlEnumerable_init_unchained`). While both are no-op functions, it is generally recommended to call the functions in case the library is updated and includes relevant code.

7.4 VSR Out-of-Bounds

Informational Version 1

CS-SPK-VLTV2-006

The VSR can in theory be out-of-bounds. Namely, that is the case when the bounds are adjusted. Consider the following scenario:

1. `setVsrBounds(RAY, MAX_VSR)`
2. `setVsr(MAX_VSR)`
3. `setVsrBounds(RAY, RAY)`

After step 3, `vsr` remains at `MAX_VSR` even though the new bounds only allow `RAY`, leaving the VSR out-of-bounds until the next `setVsr()` call with a value respecting the new limits.

7.5 Incomplete Interface Definition

Informational Version 1 Code Partially Corrected

CS-SPK-VLTV2-003

The `ISparkVault` interface defines most of the functions implemented by `SparkVault`. However, some remain undeclared within the interface, and one function's documentation doesn't match the implementation. Below is a list of such functions:

1. Functions inherited through contracts where the respective interface is not implemented by the `ISparkVault` interface (e.g., `IAccessControlEnumerable`).
2. Automatically generated getters for constants: `MAX_VSR`, `PERMIT_TYPEHASH`, `RAY`, `SETTER_ROLE`, `TAKER_ROLE`.
3. Automatically generated getters for storage variables: `maxVsr`, `minVsr`.
4. Governance setter functions: `setVsrBounds`.
5. Convenience functions defined within the contract: `assetsOf`, `assetsOutstanding`, `getImplementation`, `nowChi`, `permit` (with signature as `bytes`).
6. Initializer: `initialize`.

Additionally, the interface documentation for `chi()` is misleading. The documentation describes it as returning "the current rate accumulator", but the implementation actually returns the storage variable `chi`, which represents the rate accumulator at the last drip time (`rho`). The current rate accumulator is calculated by the `nowChi()` function, which applies the time-based growth since the last drip. Note that the interface does not necessarily need to provide all functions. However, a more complete interface could simplify scripts and integrations.

Code partially corrected:

1. Corrected.
2. None added to the interface.
3. None added to the interface.
4. Corrected.
5. Partially corrected. Only `nowChi` was added.
6. Not added.

To summarize, the functionality not added consists of automatically generated getters and convenience functionality.

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 Asset Solvency

Note Version 1

`SparkVault` might hold insufficient funds to allow for complete share redemptions. Namely, that is due to:

- Funds being taken but not returned by takers (e.g. losses, taker becoming malicious).
- Interest payments not being provided to the vault.

As a consequence, the vault might not be solvent to return funds to all users. Thus, the following might occur:

- Bank runs
- Depegging of the vault tokens

Ultimately, takers and governance are trusted to return funds to the users as promised. To prevent problems, they should aim to keep a ratio of funds within the vault to ensure that withdrawals are typically possible.

8.2 Circular Reinvesting

Note Version 1

Funds withdrawn via `take()` are intended for deployment into external yield strategies, not for systems that would eventually reinvest back into this vault. The vault assumes the `TAKER_ROLE` is trusted not to create circular reinvestment loops.

If a taker were to withdraw and redeposit funds (directly or indirectly through another system), it would create artificial deposit growth without new external capital. The vault treats all deposits equally and accrues yield based on total supply, potentially creating unsustainable yield obligations.

This design relies on the assumption that:

- Takers are trusted entities that will not create circular flows
- Withdrawn funds flow to genuine external yield opportunities
- Any system receiving funds via `take()` does not reinvest back into this vault

Governance should ensure takers understand these constraints and monitor for any unexpected circular patterns.

Note that as of Version 2, the taker cannot be provider of funds or the receiver of shares. Note that the check does not prevent circular reinvesting since it can be always bypassed. However, the check can prevent human error in well-defined contracts (e.g. ALM controller where the recipient is the ALM proxy and also the taker) but might limit the potential integration possibilities of takers.

8.3 ERC-4626 Deposit Blacklist

Note Version 2

EIP-4626 states the following about the `maxDeposit` and `maxMint` functions:

```
MUST factor in both global and user-specific limits
```

Note that `_mint` implements a blacklist:

1. `address(0)` as the receiver.
2. `address(this)` as the receiver.
3. `TAKER_ROLE` as the recipient.
4. `TAKER_ROLE` as the `msg.sender`.

Note that blacklists can be interpreted as a deposit limit of 0. However, common practice (e.g. OZ ERC4626.sol) does not consider blacklists as deposit limits.

To summarize, integrators should be aware that it the `maxMint` and the `maxDeposit` function do not reflect that special addresses cannot cannot deposit or mint.

8.4 ERC-4626 Preview Functions Liquidity Check

Note Version 1

The `previewRedeem` and `previewWithdraw` functions revert when the vault lacks sufficient liquidity. The ERC-4626 standard states these functions "MUST NOT revert due to vault specific user/global limits" but "MAY revert due to other conditions."

The current implementation treats liquidity as an "other condition" rather than a vault limit. This interpretation is reasonable since:

- Insufficient liquidity would cause the actual withdraw/redeem to revert
- The check prevents preview functions from returning misleading values
- Liquidity is a transient state, not a configured limit

This follows common practice, but integrators should be aware these preview functions can revert on low liquidity.

8.5 Exceeding Deposit Limits

Note Version 2

Note that deposit limits can be exceeded in multiple ways:

- `setDepositCap(x)` for `x < totalAssets()`.
- Interest generated leads to exceeding the deposit cap.

Nonetheless, in both cases the maximum interest obligation is predictable.

8.6 Extreme Supply Values

Note Version 2

Governance should be aware that under extreme conditions (that are considered unlikely / are not expected to occur when the vault is configured with a legitimate asset) computations can revert and can lead to DoS scenarios within the contract.

The multiplication of `chi` and `totalSupply` (shares) affects critical operations including `drip()` and hence share redemption. Below are some considerations:

- Too high `chi`: `chi` is at most `~1024*RAY` after 10 years and `~1.05e6*RAY` after 20 years at maximum VSR. This would still allow for share supplies that, for example, fit into `uint128` (see below for more details).
- Too high total supply of shares: The vault's `totalSupply` (shares) depends on deposits of the underlying asset. Overflow occurs when `totalSupply * chi > type(uint256).max`. With `chi = 1024*RAY` (10 years), max safe totalSupply ≈ `10**47` share tokens. Given that shares are minted 1:1 or less relative to deposited assets (as chi grows), this far exceeds any legitimate underlying token supply (e.g. USDC, USDS).

While other computations involving `shares * chi` or `assets * RAY` could also overflow (e.g., in `convertToAssets`, `mint`, `previewMint`), the `drip()` and redemption operations are the most critical as they affect core vault functionality and could trap existing positions.

Further, note that as part of the limitations related to the total supply, the following considerations regarding the `depositCap` can be made:

- The `depositCap` could be set so that even with tokens with large supplies reverts cannot occur due to the deposit cap (exception is due to `chi` becoming too large).
- Note that the deposit cap, the total supply and chi considerations could hypothetically lead to unexpected behaviour related to `maxMint` and `maxDeposit`. For example, `depositCap = uint256.max/RAY+1` could hold so that `maxMint` returns the magic value `uint256.max`. Assuming that the supply is non-zero there is technically a limit. However, given the considerations above related to the supply, such scenarios seem unlikely.

8.7 Repayments

Note **Version 1**

Repayments are simply transfers of the underlying token to this contract. Anyone can repay, there is no restriction that only the taker can repay. Any donations or accidental transfers are treated as repayments and reduce the outstanding debt. Governance/VSR setter should take this into account.