CHAINSECURITY

PUBLIC

Code Assessment
of the Boosted Spark Savings Vault Smart Contracts

August 17, 2026

Produced for

Spark

Contents

1 Executive Summary

Dear SparkDAO Team,

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 Boosted Spark Savings Vault according to **Scope** to support you in forming an opinion on their security risks.

SparkDAO implements `SparkBoostedVault`, a savings vault where interest is distributed based on a manually set Vault Savings Rate (VSR). The system is derived from the previously reviewed `SparkVault` (Spark Vaults V2): it retains its rate accumulator engine (`chi` / `rho` / `vsr`, `_rpow`, `drip()`) essentially verbatim, but replaces the ERC-20/ERC-4626 tokenized share layer with non-transferable, per-deposit positions whose yield (never the principal) vests under a quadratic curve with a cliff. Whitelisted takers, intended to be the Spark ALM Proxy, can draw on the vault's funds to invest.

The most critical subjects covered in our audit are functional correctness, asset solvency, access control, and the precision of arithmetic operations. Security regarding all the aforementioned subjects is high. However, some notes should be considered, for example **Asset Solvency** or **Circular Reinvesting**.

The general subjects covered are documentation, code complexity, upgradeability, and operational considerations. 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

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 Boosted Spark Savings Vault repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

Version	Date	Commit Hash	Note
1	16 July 2026	f29635169a514f1cad0a67bc566d6ae4260b776e	v1.0.0-beta.0
2	14 August 2026	26dcfaf918f309c82d8e8d04e12fc96031b61b0b	v1.0.0

For the Solidity smart contracts, the compiler version `0.8.35` was chosen with EVM version set to `cancun`.

The files in scope were:

```
src/
├── ISparkBoostedVault.sol
└── SparkBoostedVault.sol
```

2.1.1 Excluded from scope

All other files and all dependencies (e.g., the OpenZeppelin libraries) are out of scope. The deployment scripts and the tests are out of scope. The underlying tokens are out of scope.

3 System Overview

This system overview describes the latest received version (**Version 2**) of the contracts as defined in the Assessment Overview. At the end of this report section, we have added a subsection 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 implements `SparkBoostedVault`, a non-tokenized savings vault where interest is distributed based on a manually set Vault Savings Rate (VSR), and where each deposit creates an individual position whose yield vests over time according to a quadratic vesting curve. Whitelisted takers, intended to be the Spark ALM Proxy (controlled by the relayer through its controller), can draw on the vault's funds to invest. `SparkBoostedVault` is deployed through an ERC-1967 proxy using the UUPS pattern.

Note that the design is a per-position vesting variant of `SparkVault` (Spark Vaults V2): the interest accrual mechanics are the same, but the vault implements neither ERC-20 nor ERC-4626. There are no transferable share tokens and no allowance system; positions are bound to the depositing account.

3.1 Interest Accrual

The vault generates interest for the liquidity providers to incentivize them. Namely, the interest generated is defined by the `vsr` (a per-second rate in `RAY` precision) and is reflected in the rate accumulator `chi` (shares to underlying in `RAY`). Thus, `chi` is 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 stored in `rho`, and where `t_i` corresponds to the current timestamp.

The accumulator can be updated by anyone through `drip()`. It is automatically refreshed before deposits, withdrawals, and VSR updates, and its current value is simulated by `nowChi()` in the view functions. The exponentiation is performed with the `_rpow` implementation borrowed from Savings DAI (sDAI).

The VSR can be set by the `SETTER_ROLE` within the bounds `[minVsr, maxVsr]` configured by governance, which are themselves restricted to `[RAY, MAX_VSR]`, where the constant `MAX_VSR` corresponds to circa 100% APY.

3.2 Positions

`deposit(assets)` transfers the assets from the caller into the vault and creates a new position, which records:

- `principal`: the deposited assets (reduced on partial withdrawals),
- `shares`: the rate-based shares, computed as `assets * RAY / chi` (rounded down),
- `depositTime`: the current timestamp, which anchors the vesting schedule.

Each deposit creates a new, independent position with a unique incrementing `positionId`; there is no top-up of existing positions. An account can hold multiple positions, which are enumerable through view functions. An overloaded `deposit(assets, referral)` variant additionally emits a 16-bit referral code for off-chain tracking.

Deposits are limited by a global liability cap: a deposit reverts if the deposited assets exceed `maxDeposit()`, the remaining headroom between the vault's current total liability `maxLiability() = totalShares * nowChi() / RAY` and `maxLiabilityCap`. The headroom is reported as zero whenever it would not suffice to

mint a single share, so that no amount is quoted which could not be deposited. Note that `maxLiabilityCap` is zero after initialization, so deposits are disabled until governance sets the cap.

3.3 Yield Vesting

A position's raw value is `shares * chi / RAY`, and its yield is the excess of the raw value over the remaining principal. While the principal is always fully withdrawable, only the vested portion of the yield can be withdrawn. The vesting multiplier `m` for a position with `elapsed = block.timestamp - depositTime` is:

- `m = 0` if `elapsed < cliff`,
- `m = (elapsed / term)^2` if `cliff <= elapsed < term`,
- `m = 1` if `elapsed >= term`.

Hence, `withdrawableOf(positionId) = principal + yield * m`. The curve is a quadratic ease-in with a discontinuous jump at the cliff, back-loading the yield towards the end of the term and thereby penalizing early exits more than a linear curve would.

`cliff` and `term` are global, mutable parameters (see privileged functions below). Since the multiplier is evaluated dynamically against the current values, changing them immediately affects the vested/unvested split of all open positions, favorably or unfavorably, while the total accrued yield remains unaffected.

3.4 Withdrawals

Only the position owner can withdraw, to an arbitrary recipient:

- `withdraw(positionId, recipient)`: full withdrawal. Pays out `withdrawableOf(positionId)` and deletes the position.
- `withdraw(positionId, assets, recipient)`: partial withdrawal of up to `withdrawableOf(positionId)`. The position's shares and principal are reduced proportionally to the withdrawn fraction (rounding up, in favor of the vault), while `depositTime` remains unchanged so the remainder continues vesting on the original schedule. If either the shares or the principal would be exhausted, the position is closed instead.

Upon withdrawal, the unvested yield corresponding to the removed shares is forfeited and remains in the vault as surplus. Withdrawals revert if the vault's current token balance is insufficient, as takers may have removed liquidity (see below).

Additionally, the following view functionality is implemented:

- `yieldOf`, `vestedYieldOf`, `unvestedYieldOf` and `vestingMultiplierOf` decompose a position's yield according to the vesting curve.
- `withdrawableOf` returns the assets currently withdrawable from a position, while `maxWithdrawOf` additionally caps this at the vault's current token balance.
- `getPosition`, `getPositionsOf` and `getPositionIdsOf` enumerate positions.
- `maxLiability` returns the current total share-valued liability, and `maxDeposit` returns the remaining headroom under the liability cap, or zero if that headroom would not suffice to mint a single share.
- `nowChi` simulates an update of `chi`.

3.5 Privileged Functions

Privileged addresses can call the following functions:

- `TAKER_ROLE`: Can call `take()` to access the liquidity held by the vault, up to the vault's full token balance.
- `SETTER_ROLE`: Can set the `vsr`. However, the value must be within the bounds set by governance. The accumulator is dripped before the rate changes.
- Governance / `DEFAULT_ADMIN_ROLE`: Can set the VSR bounds with `setVsrBounds` (restricted to `[RAY, MAX_VSR]`), set the liability cap with `setMaxLiabilityCap`, and set the vesting parameters with `setCliff` and `setTerm` (enforcing `cliff <= term`). Additionally, the role can assign any role and can upgrade the contract.

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

3.6 Upgradeability and Initialization

`SparkBoostedVault` uses the UUPS upgradeability pattern with ERC-7201 namespaced storage. The implementation contract disables initializers in its constructor. `initialize(asset, admin, term, cliff)` initializes the inherited `AccessControlEnumerableUpgradeable` module, sets `chi`, `vsr`, `minVsr` and `maxVsr` to `RAY`, sets `rho` to the current timestamp, stores the vesting parameters (enforcing `cliff <= term`), and grants the `DEFAULT_ADMIN_ROLE` to the given admin. The initial values are emitted through the corresponding events. Upgrades are restricted to the `DEFAULT_ADMIN_ROLE`.

3.7 Changes in Version 2

In **Version 1**, deposits were limited against the liability cap directly: a deposit reverted with `MaxLiabilityCapExceeded` if `maxLiability() + assets` exceeded `maxLiabilityCap`. Moreover, `maxDeposit()` returned the remaining headroom without accounting for the minimum amount required to mint a share, so that it could quote amounts which could not be deposited. In **Version 2**, the deposited assets are checked against `maxDeposit()` and the error is named `MaxDepositExceeded`.

4 Trust Model

Below, the relevant roles for `SparkBoostedVault` are defined.

4.1 Roles

Governance / `DEFAULT_ADMIN_ROLE`

- Trust level: Fully trusted.
- Permissions: Authorize UUPS upgrades. Grant and revoke all roles (including `DEFAULT_ADMIN_ROLE`, `SETTER_ROLE` and `TAKER_ROLE`). Set the VSR bounds (`setVsrBounds`), the liability cap (`setMaxLiabilityCap`), and the vesting parameters (`setCliff`, `setTerm`).
- Worst-case impact: Replace the implementation with arbitrary code to steal all deposited funds, or grant itself the `TAKER_ROLE` and drain the vault's balance. Even without upgrading, increasing `cliff` / `term` retroactively moves the accrued yield of all open positions back into the unvested state (up to indefinitely), so that exiting users forfeit their yield; the deposited principal, however, always remains withdrawable (given sufficient liquidity).

`TAKER_ROLE`

- Trust level: Fully trusted. Intended to be the Spark ALM Proxy.
- Permissions: Can call `take()` to withdraw any amount of the underlying asset, up to the vault's full token balance.
- Assumptions: Deploys the taken funds productively and returns liquidity (taken amounts and accrued interest) to the vault as needed to satisfy user withdrawals. Does not reinvest taken funds back into the vault.
- Worst-case impact: Can fully drain the contract's balance with `take()`, leaving users unable to withdraw.

`SETTER_ROLE`

- Trust level: Partially trusted.
- Permissions: Can set the `vsr`. Expected to set meaningful VSR values; however, restricted to the bounds `[minVsr, maxVsr]` set by governance.
- Worst-case impact: Within the bounds, can maximize the vault's interest obligations (up to `maxVsr`, at most circa 100% APY) or minimize the yield paid to depositors (down to `minVsr`), including timing rate changes strategically around deposits and withdrawals.

Users

- Trust level: Untrusted.
- Permissions: Can deposit to create positions, withdraw from their own positions (to any recipient), and call the permissionless `drip()`.

4.2 External Dependencies

- Underlying token: Only valid ERC-20 tokens that are non-rebasing, non-reentrant, have no transfer fees and exhibit no other special behaviors are supported. A misbehaving token can break the vault's accounting (e.g., fee-on-transfer tokens make positions undercollateralized) or, if reentrant, enable unexpected interactions.

- Proxy: The system is expected to be deployed correctly behind an ERC-1967 proxy with the vault as its UUPS implementation, and to be initialized atomically with deployment.
- Solvency: The vault performs no accounting of amounts owed by takers. It is assumed that takers and governance return sufficient funds to the vault such that all withdrawals (principal and vested yield) can eventually be served.

5 System Considerations

We leverage this section to highlight findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require a modification inside the project. Instead, they should raise awareness in order to improve the overall understanding for users and developers.

SC1 Asset Solvency

System Consideration	Version 1

`SparkBoostedVault` might hold insufficient funds to serve all withdrawals. 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. Since withdrawals revert whenever the vault's token balance is insufficient, liquidity is effectively served on a first-come, first-served basis and bank runs might occur.

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.

SC2 Circular Reinvesting

System Consideration	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 the total shares, 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, in contrast to `SparkVault` (Spark Vaults V2), no check prevents an account holding the `TAKER_ROLE` from opening positions in the vault.

SC3 Exceeding the Liability Cap

System Consideration | Version 1

Note that the liability cap can be exceeded in multiple ways:

- `setMaxLiabilityCap(x)` for `x < maxLiability()`.
- Interest accrual increases the total liability (`totalShares * nowChi() / RAY`) beyond the cap over time.

In both cases, new deposits are blocked (`maxDeposit()` returns 0) while the existing liability continues to accrue interest. Nonetheless, the maximum interest obligation remains predictable given the cap and the VSR bounds.

SC4 Extreme Values

System Consideration | Version 1

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 `totalShares` affects critical operations including `drip()` (and hence deposits, withdrawals and VSR updates, which drip first) as well as `maxLiability()`, which gates deposits. Below are some considerations:

- Too high `chi`: At the maximum VSR (circa 100% APY), `chi` roughly doubles per year, reaching `~1024 * RAY` after 10 years and `~1.05e6 * RAY` after 20 years. This would still allow for share supplies that, for example, fit into `uint128` (see below for more details). Note that `chi` is stored as `uint192`; the unchecked cast in `drip()` is safe since `nowChi()` is bounded by `type(uint256).max / RAY` (the multiplication in `nowChi()` would revert beforehand), which is below `type(uint192).max`.
- Too high total shares: Overflow occurs when `totalShares * nowChi() > type(uint256).max`. With `chi = 1024 * RAY` (10 years), the maximum safe `totalShares` is approximately `1e47`. Given that shares are credited at a rate of at most 1:1 relative to the deposited assets (as `chi >= RAY`), 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 `yieldOf()` or `_deposit()`), the `drip()` and liability computations are the most critical as they affect core vault functionality and could trap existing positions.

Further, note that the `maxLiabilityCap` can be set so that, even with tokens with large supplies, reverts cannot occur due to excessive share supplies (the exception being `chi` becoming too large).

SC5 Repayments

System Consideration | Version 1

Repayments are simply transfers of the underlying token to this contract. The vault does not track the amounts owed by takers; any tokens held by the vault count as liquidity that is available both for user withdrawals and for `take()`. Anyone can repay, there is no restriction that only the taker can repay. Any donations or accidental transfers are indistinguishable from repayments. Governance and the VSR setter should take this into account.

SC6 Retroactive Vesting Parameter Changes

System Consideration | Version 1

`vestingMultiplierOf` reads `cliff` and `term` live from global storage `setCliff` and `setTerm`, both gated by `DEFAULT_ADMIN_ROLE`, overwrite these globals, so changing either re-scales the vesting of all existing positions retroactively: raising `term` or `cliff` withholds yield already counted as vested, lowering them releases more immediately. Principal is unaffected; only the vested-yield component depends on the multiplier.

This differs from a VSR change, where `setVsr` calls `drip()` first to checkpoint `chi`, so the new rate applies only going forward. `setCliff` and `setTerm` have no such checkpoint, so their effect is fully retroactive, and repeated changes can re-scale vested amounts repeatedly. Depositors cannot rely on the vesting schedule in effect when they deposited.

SC7 Incompatible with Permissionless Withdrawals

System Consideration | Version 1

Spark Vaults V2 supports Permissionless Withdrawals, a backup path that lets users redeem their shares even when the vault holds no idle liquidity, by sourcing the shortfall from whitelisted venues through the Spark ALM MainnetController and the ALMProxy.

Such a mechanism could not be used with `SparkBoostedVault`:

1. The vault is not tokenized: there are no ERC-20 shares and no ERC-4626 redeem functionality to operate on.
2. Withdrawals are bound to the position owner: `_withdraw` requires `msg.sender` to own the position, and no redeem-on-behalf functionality exists.

Consequently, when the vault holds insufficient liquidity, withdrawals revert until the taker returns funds; no permissionless backup path can source the shortfall.

> **Acknowledged**
>
> Compatibility with Permissionless Withdrawals is not a requirement. SparkDAO states that the mechanism is a backstop that is not expected to be needed, and that the additional complexity was deliberately not taken on.

6 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

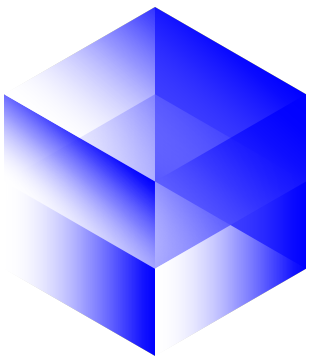
We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

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.

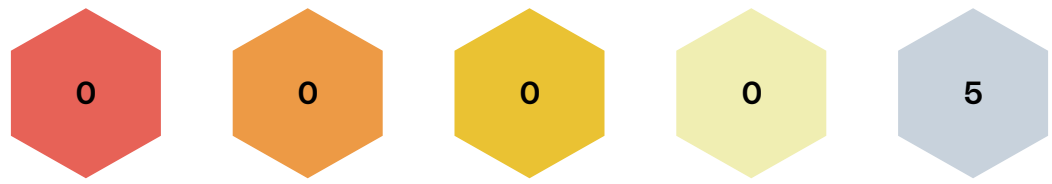
7 Findings

In this section, we describe the findings identified during the course of the engagement. The findings are categorized by severity as explained in the Terminology section. Below we provide an overview of the total number of findings per severity, as well as the number of open issues.



No Open Issues

Total number of findings



Critical | High | Medium | Low | Informational

7.1 Overview

The following table lists all identified findings along with their severity and current status. A finding is considered resolved once it has been fully corrected or the specification has been changed accordingly. Non-informational findings with any other status, including partial fixes, acknowledgements, and accepted risks, remain open.

Info	#001	Incomplete Interface Definition	Acknowledged
Info	#002	Lack of Events	Code Part. Corrected / Acknowledged
Info	#003	Liability Cap Can Be Exceeded by Rounding in Deposit Share Conversion	Specification Changed
Info	#004	Library Initializers Unused	Code Corrected
Info	#005	VSR Out-of-Bounds	Acknowledged

7.2 Descriptions

#001 Incomplete Interface Definition

Informational	Version 1	Acknowledged

The `ISparkBoostedVault` interface declares most of the functions implemented by `SparkBoostedVault`. However, some functionality remains unexposed:

1. Functions inherited through `UUPSUpgradeable` (e.g., `upgradeToAndCall`, `proxiableUUID`, `UPGRADE_INTERFACE_VERSION`) are not declared in the interface.
2. The storage variable `positionCount` is not exposed through a getter (and is consequently also not part of the interface). Off-chain integrators can infer position identifiers from Deposit events, while on-chain callers receive the identifier as the return value of `deposit()`."

Compared to `SparkVault`, `drip()` does not return the current `chi` value.

Note that the interface does not necessarily need to provide all functions. However, a more complete interface could simplify scripts and integrations.

Acknowledged

SparkDAO states:

> There is not trivial way to define the interfaces inherited from OZ because OZ does not write libraries that allow for this. Also, we don't want `positionCount` to be relied upon or understood to be monotonically increasing. It is in the black box. Since this vault already has several breaking interface changes from the V2 vault, we shed all unnecessary constraints.

#002 Lack of Events

Informational	Version 1	Code Partially Corrected / Acknowledged

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.
4. `cliff` and `term` are set but `CliffSet` and `TermSet` are not emitted.

Additionally, compared to `SparkVault` (Spark Vaults V2), some events carry less information:

1. `VsrSet` and `MaxLiabilityCapSet` no longer include the previous value, while the corresponding v2 events `VsrSet` and `DepositCapSet` emitted both the old and the new value.
2. `drip()` returns early without emitting an event when called again within the same block, while v2 emitted `Drip` on every call.

Code Partially Corrected

In **Version 2**, `initialize()` emits `CliffSet`, `TermSet`, `VsrBoundsSet`, `VsrSet` and `Drip` for the values it sets, which addresses the four occurrences listed above.

Acknowledged

The remaining observations are not addressed: `VsrSet` and `MaxLiabilityCapSet` still emit only the new value, and `drip()` still returns without emitting `Drip` when called again within the same block. Additionally, **Version 2** removes the indexed `sender` parameter from `CliffSet`, `TermSet` and `VsrSet`.

#003 Liability Cap Can Be Exceeded by Rounding in Deposit Share Conversion

Informational **Version 1** **Specification Changed**

A liability cap violation is possible due to rounding. `_deposit` floors the shares as `shares_ = assets_ * RAY / chi` but stores `Position.principal = assets_`, while the deposit cap is enforced against `maxLiability() = totalShares * nowChi() / RAY`, the rounded-down share value. The principal-denominated obligation can therefore slightly exceed `maxLiabilityCap`. For example:

1. `chi = RAY + 1`
2. a user deposits `assets_ = 1e27`
3. `shares_ = floor(1e27 * RAY / (RAY + 1)) = 1e27 - 1`
4. `principal` stored `= assets_ = 1e27`
5. this position's `maxLiability` contribution `= floor(shares_ * chi / RAY) = 1e27 - 1`, one wei below the stored principal
6. the cap check uses this rounded value, so the true obligation can exceed `maxLiabilityCap` by about one wei per open position

The effect is minimal: bounded at about one wei per open position, not amplifiable, and principal stays fully backed so solvency is unaffected. It is typically outweighed because `maxLiability()` overestimates the actually-withdrawable liability, counting full accrued yield where `withdrawableOf` pays only principal plus vested yield and withholds unvested yield.

`maxLiability()` itself also floors, adding at most a further wei of aggregate slack, which is immaterial for the same reason.

Specification Changed

In **Version 2**, the documentation was updated: `maxLiability()` is now specified as returning the total *share-valued* liability, and `maxLiabilityCap()` is documented to gate new deposits only, with the explicit note that `maxLiability()` grows with `chi` and can exceed the cap.

#004 Library Initializers Unused

Informational **Version 1** **Code Corrected**

The internal initializer of `AccessControlEnumerableUpgradeable` is not called in `initialize` (e.g., `__AccessControlEnumerable_init`). While these functions are currently no-ops, it is generally recommended to call them in case the library is updated and includes relevant code. Note that `UUPSUpgradeable` in the OpenZeppelin version used (5.6.1) no longer defines an initializer.

Code Corrected

In **Version 2**, `initialize()` calls `__AccessControlEnumerable_init()`.

#005 VSR Out-of-Bounds

Informational	Version 1	Acknowledged

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.

Acknowledged

`setVsrBounds()` is unchanged in **Version 2** and still does not re-clamp the active `vsr`.

SparkDAO notes that governance can correct the VSR itself.

8 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.