CHAINSECURITY

PUBLIC

Code Assessment
of the Spark Prime Vault Smart Contracts

November 09, 2026

Produced for

Spark



Contents

1	Executive Summary	3
2	Assessment Overview	4
3	System Overview	5
4	Trust Model	8
5	System Considerations	11
6	Terminology	15
7	Findings	16
8	Limitations and use of report	20

1 Executive Summary

Dear Spark Team,

Thank you for trusting ChainSecurity with the review of Spark Prime Vault.

As a globally trusted security partner for top-tier DeFi protocols, ChainSecurity provides specialized, independent technical analysis of complex smart contract systems, including Spark Vault V2, on which Spark Prime Vault builds.

We greatly appreciate the opportunity to collaborate with Spark, a leading protocol in the Sky ecosystem, on this project.

Spark implements Spark Prime Vault, a vault in which users earn yield on USDC by funding Spark's lending to borrowers. Deposits and redemptions are queued and processed by Spark, while funds that are not yet deployed are held in Spark Vault V2 (`spUSDC`).

We have conducted a thorough review of the scope. The most important areas covered in our review are:

- The request queues, since deposits, cancellations and redemptions are processed in order, with partial fills and an instant fill, which makes their accounting and liveness critical. We found that queued requests can be reduced to dust, see **Dust Can Enter the Queues**.
- The precision of arithmetic operations, since every request is converted between USDC, `spUSDC` shares and `spPrime` at two independently growing exchange rates. We found that redemption payouts round against the vault, see **Redemptions Round Against the Vault**.
- The integration with Spark Vault V2 and USDC, since the vault relies on `spUSDC` for its liquidity and on the USDC blacklist to handle failed payouts. We describe the resulting dependencies in **Deposits Depend on the spUSDC Deposit Cap** and **Funds of Unprocessed Requests Are Not Reserved**.

We provide this audit report to assist in forming an opinion on the risk profile of Spark Prime Vault. In summary, we find that the codebase provides a good level of security, if operated under the assumptions documented in **System Overview**, **Trust Model** and **System Considerations**.

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 Spark Prime Vault repository based on the documentation files. The code versions relevant to this report and when they were received are indicated below.

Version	Date	Commit Hash	Note
1	19 October 2026	67609418dfca86d88130594056904d54d1f37982	v1.0.0-beta.0
2	22 October 2026	139dcc049263e9aacc94f1f2c267a6de258d8b05	v1.0.0-beta.1
3	5 November 2026	0256114fb304a8f0f94e28f05e1636087540c818	v1.0.0-rc.0
4	9 November 2026	a1b60c31ad987eac4baa987ef826b059c6e835d5	v1.0.0

For the Solidity smart contracts, the compiler version `0.8.34` was chosen with the EVM version set to `osaka`.

The following files were in scope:

```
src/
├── ISparkPrimeVault.sol
└── SparkPrimeVault.sol
```

2.1.1 Excluded from scope

All other files are out of scope, in particular the tests and all dependencies in `lib/`. Third-party libraries are assumed to function correctly and as documented. The following are explicitly out of scope:

- The savings vault (`spUSDC`, Spark Vault V2) and the underlying asset (USDC), including its blacklist (`isBlacklisted`). Both are assumed to behave as specified. The reference for the savings vault is Spark Vault V2 `v1.0.1`, which has been reviewed in a separate ChainSecurity audit.
- The off-chain operations of Spark's Planner, i.e., the holders of `PROCESSOR_ROLE` (`OPERATOR_ROLE` in **Version 2**) and `TAKER_ROLE`, and the management of funds after they are taken from the vault with `take()`.
- The deployment, initialization parameters and configuration of the vault.

3 System Overview

This system overview describes the latest version (**Version 4**) of the contracts as defined in the Assessment Overview. At the end of this report section, we have added subsections for each of the changes according to the versions.

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

Spark offers Spark Prime Vault, a vault in which users earn yield on USDC by funding Spark's lending to borrowers. Depositors receive `spPrime`, whose value grows at a rate set by Spark, the Vault Savings Rate (VSR). Because the funds are lent out, deposits are only filled up to a capacity that reflects what Spark can deploy, and redemptions are only paid as far as funds are available in the vault. Both therefore wait in queues that Spark processes. Funds that are not yet deployed are held in `spUSDC`, the Spark Vault V2 for USDC.

The vault therefore owes its users mainly in two units. Holders of `spPrime`, including those waiting for a redemption, are owed the current value of their `spPrime`. Users with a queued deposit or cancellation are owed `spUSDC` shares. Against this, the vault holds `spUSDC`, while the remaining funds are with Spark, which keeps any surplus and is expected to cover any shortfall. In addition, recipients of failed payouts are owed USDC, which the vault holds directly (see below).

The value of one `spPrime` in USDC is the rate accumulator `chi`, compounded to the current time. It grows every second at the VSR, using the same mechanism as Spark Vault V2. The operator chooses the VSR within bounds that the admin sets between 0% and 100% APY.

Spark operates the vault through a set of roles. Its Planner holds the `OPERATOR_ROLE`, with which it processes the queues and sets the VSR and the capacity. It manages liquidity through Spark's PAU, whose `ALMProxy` holds the `TAKER_ROLE` and whose rate limits bound how much `spUSDC` it can take and return.

The admin (`DEFAULT_ADMIN_ROLE`) grants all roles, sets the minimum request sizes, the VSR bounds and the maximum redeem fee, and can upgrade the contract. The `LOSS_MANAGER_ROLE` writes down the value of `spPrime` after a loss. The `REDEEM_FEE_SETTER_ROLE` sets the redeem fee within the bound set by the admin. The `PAUSER_ROLE` can pause deposits and redemptions, including their processing, and the `UNPAUSER_ROLE` can unpause the vault.

3.1 Deposits

A user deposits USDC with `requestDeposit()`, at least `minDeposit`, and names a recipient for the `spPrime`. The vault deposits the USDC into `spUSDC` and queues a request for the received `spUSDC` shares. The depositor becomes the owner of the request and is the only one who can cancel it or change its recipient.

If no earlier deposit is pending, the request is filled in the same transaction as far as the free capacity allows, and the rest waits for the operator. The free capacity is the capacity set by the operator minus the `spPrime` supply, including `spPrime` queued for redemption. The operator fills waiting deposits with `processPendingDeposits()` in the order they arrived, until the free capacity is used up or it reaches the amount or number of entries it passes as limits. A request that only partly fits is filled partially and stays first in line. To fill a request, the vault values its `spUSDC` shares at the current `spUSDC` price and mints `spPrime` to the recipient at the current `spPrime` value. No funds move in this step, as the `spUSDC` stays in the vault and now backs `spPrime` instead of the deposit request. While a deposit waits, it earns the `spUSDC` rate.

3.2 Cancellations

As long as part of a deposit is unfilled, its owner can cancel all or part of it. Each cancellation must be at least `minDepositCancellation` `spUSDC` shares, so a smaller remainder can only be filled. The cancelled shares

are removed from the deposit request and appended to the cancellation queue with a recipient for the USDC, which the owner chooses and can change later. Fully cancelled deposit requests stay in the deposit queue until the operator processes them, or until a new deposit with nothing pending ahead of it skips them.

The operator pays out cancellations with `processPendingDepositCancellations()`, again in the order they arrived. It redeems the queued `spUSDC` shares and sends the resulting USDC, including the `spUSDC` yield, to the recipient. The amount it can process is limited by the `spUSDC` the vault holds and by the USDC available in `spUSDC`. As with deposits, a partially processed cancellation stays first in line. Unlike deposits and redemptions, cancellations can be requested and paid out while the vault is paused.

3.3 Lending

The Planner moves `spUSDC` out of the vault with `take()` to lend the funds to borrowers, and returns `spUSDC` when redemptions are due. `take()` can also move the `spUSDC` of queued deposits and cancellations, so cancellations, like redemptions, may depend on the Planner returning funds.

3.4 Redemptions

To exit, a holder calls `requestRedeem()` with at least `minRedeem` `spPrime` and names a recipient for the USDC. The `spPrime` are transferred to the vault, the redeem fee, at a rate of at most 1% and rounded up, is burned, and the remaining `spPrime` are appended to the redeem queue. They keep growing in value until they are paid out. The redemption cannot be cancelled, but its owner can change the recipient. Burning the fee reduces what the vault owes without changing the value of other holders' `spPrime`, so the fee stays with Spark.

The operator pays out redemptions with `processPendingRedeems()` in the order they arrived. For each request, it burns the queued `spPrime` and redeems the `spUSDC` shares their current value is worth to the recipient, as far as the vault's `spUSDC` and the USDC available in `spUSDC` allow. A partially paid redemption stays first in line. The vault does not reserve `spUSDC` for any queue, so redemptions and cancellations draw from the same balance, which also includes the `spUSDC` of queued deposits. The operator decides which queue to process first. Because redemptions burn `spPrime` and free capacity, processing them first lets the queued deposits be filled into the freed capacity, so entries and exits can net out without Spark returning funds.

3.5 Failed Payouts and Losses

USDC can blacklist addresses, and a payout that reverts would block every request behind it. If a payout to the recipient reverts, the USDC is therefore sent to the vault instead and credited to the recipient in `claimableAssetsOf`, and `totalClaimable` tracks the total amount owed. Anyone can call `claim()`, which pays the recipient once it can receive USDC again.

The VSR is set by Spark independently of how the loans perform. If the loans perform worse, Spark can realize the loss in the value of `spPrime`. The contract only allows this while the vault is paused, so that users cannot enter or leave at a value that is about to change. The loss manager then calls `impair()` with the loss in USDC, which sets `chi` so that the total value of all `spPrime`, including those queued for redemption, decreases by this amount. Impairing the full value sets `chi` to zero, after which deposits can no longer be filled and redemptions revert until the contract is upgraded. Gains cannot be realized, so holders only earn the VSR. Queued deposits and cancellations are denominated in `spUSDC` shares and are not affected.

3.6 Changes in Version 2

Compared to **Version 1**, **Version 2** introduced the following changes:

- The roles were reorganized:
  - The `OPERATOR_ROLE` replaces the `PROCESSOR_ROLE` and the `VSR_SETTER_ROLE`, and also sets the capacity, which was set by the admin in **Version 1**.
  - The `LOSS_MANAGER_ROLE` calls `adjustTotalAssets()`, which the admin called in **Version 1**.
  - The `PAUSER_ROLE` replaces the `GUARDIAN_ROLE`, and the `UNPAUSER_ROLE` unpauses the vault instead of the admin.
- Recipients are no longer checked against the USDC blacklist when a request is made or its recipient is changed. Any payout that fails for a recipient is credited to `claimableAssets`, and the new `totalClaimable` tracks the total amount owed.
- Deposit cancellations can be requested and processed while the vault is paused.
- `requestDepositCancellation()` removes the empty deposit requests at the front of the deposit queue, so that the instant fill does not have to iterate over them.
- Redemption requests of zero shares are no longer skipped, but processed and deleted.
- The paginated getters accept a length of zero.

3.7 Changes in Version 3

Compared to **Version 2**, **Version 3** introduced the following changes:

- `requestDepositCancellation()` no longer removes the empty deposit requests at the front of the deposit queue. Instead, a new deposit with nothing pending ahead of it moves the head of the queue directly to its own request.
- Redemptions are paid out by redeeming the `spUSDC` shares the redeemed `spPrime` are worth instead of withdrawing their USDC value.
- `adjustTotalAssets()` limits `chi` to `type(uint256).max / RAY`, and the capacity is limited to `type(uint128).max`.
- Requests record the time they were made. The vault exposes the length of each queue, and the paginated getters clamp the requested length to the end of the queue.
- Permits of signers with code, including EIP-7702 delegated accounts, are only validated with ERC-1271.
- The events of the role-restricted functions include the caller, and processing a queue emits an additional event.
- Several functions and variables were renamed, for example `claimableAssets` to `claimableAssetsOf` and the first pending request IDs to `processedDepositCount()`, `processedDepositCancellationCount()` and `processedRedeemCount()`.

3.8 Changes in Version 4

Compared to **Version 3**, **Version 4** introduced the following changes:

- `adjustTotalAssets()` is replaced by `impair()`, which can only decrease the value of `spPrime`. Gains can no longer be realized, and `chi` can become zero.

4 Trust Model

`spPrime` is not fully backed by assets held in the vault. The vault only holds the `spUSDC` that Spark has not yet taken, and the value of `spPrime` is set by Spark through the VSR and `impair()`. Users therefore rely on Spark to return the funds it takes, to realize losses correctly, and to process the queues. According to Spark, the admin is Spark governance.

4.1 Roles

The roles below were introduced in **Version 2**. In **Version 1**, the `PROCESSOR_ROLE` and the `VSR_SETTER_ROLE` held the permissions of the `OPERATOR_ROLE` except setting the capacity, and the admin set the capacity, unpaused the vault and called `adjustTotalAssets()`. The `GUARDIAN_ROLE` of **Version 1** is the `PAUSER_ROLE`, which in **Version 1** could also pause deposit cancellations.

`DEFAULT_ADMIN_ROLE` (admin)

- Trust level: Fully trusted.
- Permissions: Grant and revoke all roles. Upgrade the vault (UUPS). Set the minimum request sizes, the VSR bounds and the maximum redeem fee.
- Worst-case impact: Upgrade the vault and take all funds, including the `spUSDC` of queued deposits and cancellations and the USDC owed to recipients of failed payouts. Grant itself any other role and act as described below. Without an upgrade, set the minimum cancellation and redemption sizes above all balances, so that no user can exit.

`LOSS_MANAGER_ROLE`

- Trust level: Fully trusted.
- Permissions: While the vault is paused, write down the value of `spPrime` with `impair()`. Before **Version 4**, `adjustTotalAssets()` could also raise it.
- Worst-case impact: Reduce the value of all `spPrime`, including those queued for redemption, to zero, so that deposits can no longer be filled and redemptions revert until the vault is upgraded. Before **Version 4**, raise it so far that the rate accumulator overflows and all requests and queue processing revert until the vault is upgraded.

`TAKER_ROLE` (Spark's PAU)

- Trust level: Fully trusted.
- Permissions: Take any amount of `spUSDC` from the vault with `take()`, also while the vault is paused.
- Worst-case impact: Take all `spUSDC` of the vault and never return it, so that no cancellation or redemption can be paid out.

`OPERATOR_ROLE` (Planner)

- Trust level: Partially trusted.
- Permissions: Process the three queues with an amount and number of entries of its choice, and choose the order in which the queues are processed. Deposits and redemptions can only be processed while the vault is not paused. Set the VSR within the bounds set by the admin, at most 100% APY, and set the capacity.
- Assumptions: The operator processes the queues regularly and fills requests in a fair order across queues.
- Worst-case impact: Stop processing, so that no cancellation or redemption is paid out and deposits are only filled instantly until the first deposit exceeds the free capacity, after which all deposits stay queued. Pay out redemptions before cancellations, or the reverse, when liquidity is insufficient for both. Pay out redemptions

after a loss but before it is realized, so that the remaining holders absorb it, or fill deposits in that period, so that new depositors absorb part of it. Set the VSR to the upper bound, so that the value of `spPrime` grows faster than Spark's returns and Spark's shortfall increases, or to the lower bound, so that holders earn less than expected. Set the capacity to zero, so that no deposit is filled.

`REDEEM_FEE_SETTER_ROLE`

- Trust level: Partially trusted.
- Permissions: Set the redeem fee up to the maximum set by the admin, at most 1%.
- Worst-case impact: Raise the fee to the maximum right before a user's redemption request. Users cannot limit the fee they accept.

`PAUSER_ROLE`

- Trust level: Partially trusted.
- Permissions: Pause deposit and redemption requests and the processing of deposits and redemptions.
- Worst-case impact: Block deposits and redemptions until the admin revokes the role and the vault is unpaused, since the pauser can pause again right after an unpause.

`UNPAUSER_ROLE`

- Trust level: Partially trusted.
- Permissions: Unpause the vault.
- Worst-case impact: Unpause the vault during an emergency or before a loss is realized with `impair()`, so that users can enter or leave at a value that is about to change.

Users

- Trust level: Untrusted.
- Permissions: Request deposits, deposit cancellations and redemptions of at least the minimum sizes, change the recipients of their own requests, transfer `spPrime`, and call `claim()` and `drip()`.
- Worst-case impact: Fill the queues with many small requests, which costs the operator gas, and leave remainders that keep later deposits from being filled instantly.

4.2 External Dependencies

`spUSDC` (Spark Vault V2)

- Assumed to behave as in the reviewed release `v1.0.1`. It is upgradeable by Spark governance.
- Its own `TAKER_ROLE` can remove its USDC liquidity. The vault then cannot withdraw its `spUSDC`, and cancellations and redemptions cannot be paid out.
- Its deposit cap applies to the vault. Once it is reached, or if the vault is granted the `TAKER_ROLE` of `spUSDC`, `requestDeposit()` reverts.

USDC

- Assumed to be a standard ERC-20 token with 6 decimals and a blacklist.
- Circle can blacklist recipients. Payouts to blacklisted recipients are credited to `claimableAssetsOf` and can only be claimed once they are removed from the blacklist.
- Circle can pause USDC or blacklist `spUSDC`, which blocks all deposits and payouts. If it blacklists the vault, deposits and `claim()` are blocked, and a payout to a blacklisted recipient reverts, which blocks its queue.

Spark's PAU and Planner

- The PAU and the off-chain Planner are out of scope. They are assumed to return `spUSDC` to the vault in time to serve cancellations and redemptions, and the PAU's rate limits are assumed to be configured appropriately.

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 Circular Reinvesting

System Consideration | Version 1

Funds moved out with `take()` are meant to be lent to external borrowers, which earns the yield to pay the VSR. If they are instead deposited back into the vault, directly or through another system, the vault's obligations grow without any additional lending:

1. If the taker or another Spark system holds the resulting `spPrime`, Spark owes the VSR to itself. This has no net effect, but the `spPrime` takes up capacity that is meant for external depositors.
2. If a third party ends up holding the `spPrime`, Spark owes it the VSR on funds that do not generate any yield.

The vault relies on the `TAKER_ROLE` not to create such circular flows. It does not prevent the taker from requesting deposits, being the recipient of a deposit or receiving `spPrime` through transfers. Unlike Spark Vault V2, the vault applies no such restrictions, as the PAU has no functionality to request deposits. The capacity limits how much `spPrime` such flows can create.

SC2 Configured Limits Are Not Fixed Bounds

System Consideration | Version 1

The limits set by the admin are only checked for new actions. Changing a limit does not affect the existing state:

- `setCapacity()` can set the capacity below `totalSupply`.
- The minimum request sizes do not apply to queued requests, and partial fills and cancellations can leave requests below them.
- `setVSRBounds()` does not check the current `vsr`.
- `setMaxRedeemFeeRate()` does not check the current `redeemFeeRate`.

Furthermore, the capacity and `minRedeem` are denominated in `spPrime` and `minDepositCancellation` in `spUSDC` shares. Their value in USDC therefore grows over time, while `minDeposit` is fixed in USDC.

In **Version 2**, the capacity is set by the `OPERATOR_ROLE` instead of the admin.

Spark specified that changes to the bounds will be paired with matching updates of the current values.

SC3 Deposits Depend on the spUSDC Deposit Cap

System Consideration | Version 1

`requestDeposit()` deposits the USDC into `spUSDC` immediately. A deposit of `assets` therefore only succeeds if

```
minDeposit <= assets && spUSDC.totalAssets() + assets <= spUSDC.depositCap()
```

The upper bound is outside of the vault's control. It is shared with all other `spUSDC` depositors and shrinks as `spUSDC` accrues interest. Once it falls below `minDeposit`, no deposit can be requested.

SC4 Extreme Values Can Stop the Vault

System Consideration | Version 1

`drip()` and `nowChi()` multiply the compounded rate with `chi`, and `drip()` multiplies `totalSupply` with `chi`. If these products exceed `type(uint256).max`, the computations revert. Since deposit and redemption requests (in **Version 2** also deposit cancellation requests), all processing, `setVSR()` and `adjustTotalAssets()` call `drip()`, and all conversions use `nowChi()`, the vault then stops working until it is upgraded. This can happen in the following ways:

1. `chi` grows at the VSR. At the maximum VSR of 100% APY, it reaches about 1024 times its initial value after 10 years. The capacity bounds `totalSupply`, so the product with `chi` stays far below the limit for any realistic configuration.
2. `adjustTotalAssets()` accepts any `chi` up to `type(uint192).max`. The compounded rate is at least `RAY`, so any `chi` above about `type(uint256).max / RAY` makes the first `drip()` overflow.
3. `_processPendingDeposits()` converts `availableCapacity()` with `convertToAssets()`, which multiplies it with `chi`. A capacity above about `type(uint256).max / chi`, for example `type(uint256).max` to express an unlimited capacity, makes every deposit fill revert, including the instant fill in `requestDeposit()`. As `chi` grows, this bound decreases over time. This only stops deposits, and the admin can resolve it by lowering the capacity. In **Version 2**, deposit cancellation requests perform this conversion as well, so that queued deposits can neither be filled nor cancelled until the `OPERATOR_ROLE`, which sets the capacity in **Version 2**, lowers it.

The extreme values were bounded in later versions:

1. In **Version 3**, the capacity, and thereby `totalSupply`, is limited to `type(uint128).max`.
2. In **Version 4**, `adjustTotalAssets()` is replaced by `impair()`, which can only decrease `chi`.
3. In **Version 3**, the conversion only overflows once `chi` exceeds about 3.4e11 times its initial value, about 38 years at the maximum VSR. Deposit cancellation requests no longer perform it.

SC5 Funds of Unprocessed Requests Are Not Reserved

System Consideration | Version 1

The vault does not reserve the `spUSDC` of unprocessed deposits and cancellations:

1. `take()` can move any `spUSDC` the vault holds, including the `spUSDC` of unprocessed deposits and cancellations.
2. Cancellations and redemptions are paid out from the vault's entire withdrawable `spUSDC`, as given by `availableLiquidAssets()` and `maxRedeem()`, including the `spUSDC` of unprocessed deposits and of other cancellations.

As a consequence, users with unprocessed deposits or cancellations are treated unequally compared to `spPrime` holders:

- Their funds can be lent out, so they depend on Spark returning them, while they only earn the `spUSDC` rate and not the VSR. Unlike for `spPrime`, losses realized by the loss manager do not reduce what they are owed, so they are only exposed to the risk that Spark cannot return the funds.

- Their funds can be used to pay out other users. A user who cancels a deposit may have to wait for the payout while the same funds are used for redemptions.

According to the release notes, this is intended, as pending deposits and cancellations are liabilities owed by Spark, which manages them with the Planner.

SC6 Minimum Request Sizes Might Not Be Consistent

System Consideration | Version 1

The minimum request sizes are set independently and in different units. A deposit of `minDeposit` can only be cancelled and later redeemed if

```
minDeposit / spUSDCSharePrice >= minDepositCancellation
minDeposit / spPrimeSharePrice >= minRedeem
```

Both share prices grow over time, so these conditions can become violated for new deposits without any change to the parameters. A deposit close to `minDeposit` made after that point cannot be cancelled, and the `spPrime` it is filled into cannot be redeemed.

Spark specified that the minimums will be reviewed periodically as the share prices grow.

SC7 Value Loss When Processing Requests

System Consideration | Version 1

Deposits, cancellations and redemptions are converted when the processor executes them, not when they are requested. Neither users nor the vault can react to changes in between:

- Requests processed after a loss that is not yet realized with `impair()` (before **Version 4**, also a gain, realized with `adjustTotalAssets()`) are converted at a stale value of `spPrime`. Depending on the direction of the change and the type of request, either the user or the remaining holders lose value. After a loss, for example, deposits are filled at a value that is too high, while too much is paid out for redemptions.
- All conversions round against the user. Fills and payouts (since **Version 3**) round down, and the redeem fee rounds up. For dust requests, this can cost the user a large part of the value, while the difference stays with the vault or `spUSDC`.

Neither case can easily be prevented. Stale values cannot be detected on-chain before a loss (before **Version 4**, also a gain) is realized. Protecting against rounding would require a minimum outcome per request, which does not fit the FIFO processing, since a request that does not meet it would either block the queue or have to be skipped. This would add complexity for cases that are generally unlikely and, for rounding, are limited by the minimum request sizes.

We expect monitoring to be in place, so that losses and gains are detected and realized quickly. Since **Version 4**, gains are no longer realized, so only losses lead to stale values.

SC8 Claims Cannot Be Redirected

System Consideration | Version 1

Before a cancellation or redemption is processed, its owner can change the recipient, for example if the recipient is blacklisted. Once a failed payout has been credited to `claimableAssets`, however, only the

original recipient can receive it, and nobody can redirect it. If the recipient stays blacklisted, the funds are locked permanently.

Spark specified that this is by design. The fallback covers any address-specific revert of the token, of which USDC blacklisting is one case.

SC9 Deposits Cannot Be Processed Selectively

System Consideration | Version 1

Deposits are filled in the order they arrived, and the operator cannot skip deposits or choose which ones to fill. A dust deposit at the head of the queue therefore keeps all later deposits from being filled instantly until the operator processes it, see Dust Can Enter the Queues.

Spark specified that the FIFO processing is intended and that no selective mode is planned.

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

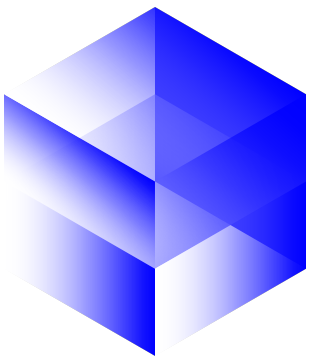
We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

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.

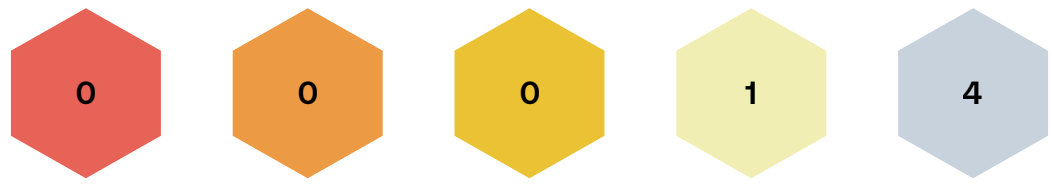
7 Findings

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.

Low	#001	Redemptions Round Against the Vault	Code Corrected
Info	#002	Dust Can Enter the Queues	Acknowledged
Info	#003	Inconsistencies	Code Part. Corrected / Acknowledged
Info	#004	No Slippage Protection for Requests	Code Part. Corrected / Acknowledged
Info	#005	Paginated Getters Are Hard to Use	Code Corrected

7.2 Descriptions

#001 Redemptions Round Against the Vault

Correctness **Low** **Version 1** **Code Corrected**

All conversions in the vault round in its favor, except the payout of redemptions. `_processPendingRedeems()` pays out a redemption with `_withdrawSavings()`, which calls `spUSDC.withdraw()` for the USDC value of the redeemed `spPrime`. `withdraw()` rounds the burned `spUSDC` shares up. Even with an exact `convertToAssets()`, the vault therefore burns more `spUSDC` than the redeemed `spPrime` are worth. The remaining holders bear the difference, which is at most one `spUSDC` share per payout, but occurs for every redemption and every partial fill.

For example, with an `spUSDC` share price of 1.06, a redemption worth 10 wei of USDC burns 10 `spUSDC` shares (`ceil(10/1.06)`), which are worth 10.6 wei. The vault loses 0.6 wei. Redeeming 9 shares (`floor(10/1.06)`) instead would pay out 9 wei, and the rounding would stay with the vault.

Computing the shares with `savings.convertToShares(convertToAssets(shares_))` and paying them out with `_redeemSavings()`, as for cancellations, would round in favor of the vault.

> **Code Corrected**
>
> The vault now redeems `savings.convertToShares(convertToAssets(shares_))` `spUSDC` shares, rounded down, as for cancellations.

#002 Dust Can Enter the Queues

Informational **Version 1** **Acknowledged**

The minimum request sizes are meant to keep dust out of the queues, so that processing stays efficient and the queues cannot be griefed. They are, however, only checked when a request is made, and queued requests can still be reduced to dust:

1. `requestDepositCancellation()` only checks that the cancelled amount is at least `minDepositCancellation`, not that the remaining deposit is worth at least `minDeposit`. For example, a user can deposit `minDeposit` and cancel all but 1 wei of `spUSDC` shares.
2. Partial processing can leave any remainder in all three queues. This includes the instant fill in `requestDeposit()` when capacity is short, and `processPendingDeposits()`, `processPendingDepositCancellations()` and `processPendingRedeems()` when their limits are reached.

While partial processing only occurs when the capacity, the available liquidity or the limits of the processor are reached, anyone can reduce their deposit to dust with a cancellation. However, it requires locking at least `minDeposit` until the cancellation is processed, and these funds are exposed as described in Funds of Unprocessed Requests Are Not Reserved. This limits the impact, as the minimum is expected to be set high.

Dust requests have to be processed like any other request, and a pending dust deposit prevents later deposits from being filled instantly. Furthermore, a dust deposit below `minDepositCancellation` can no longer be cancelled.

> **Acknowledged**
>
> Spark will set the minimum request sizes so that dust is uneconomic. Partial fills and partial cancellations are inherent to FIFO processing.

#003 Inconsistencies

Informational **Version 1** **Code Partially Corrected / Acknowledged**

The following inconsistencies were found:

1. All functions that use `chi` or change `totalSupply` call `drip()` first. Among the functions that do neither, `requestDeposit()` and `_processPendingDepositCancellations()` call `drip()` without needing it, while `requestDepositCancellation()` does not. In `requestDeposit()`, `chi` is only used by the instant fill, and `_processPendingDeposits()` calls `drip()` itself.
2. The `update*Recipient()` functions revert with `InvalidRequestId()` for request IDs beyond the end of the queue. `requestDepositCancellation()` does not check the request ID and reverts with an out-of-bounds panic instead.
3. The vault exposes its implementation through `implementation()`, while Spark Vault V2 and other Spark and Sky contracts use `getImplementation()`.
4. Further details differ from Spark Vault V2:
    1. `drip()` returns early without emitting `Drip` if no time has passed since the last drip, while Spark Vault V2 always emits it.
    2. `drip()` and `nowChi()` return `uint192` instead of `uint256`.
    3. In Spark Vault V2, `rho` and `chi` share a storage slot. In the vault, `decimals` is declared before `rho`, so `rho` is packed with `decimals` and `chi` occupies a separate slot. `drip()` therefore updates two storage slots instead of one.
5. The recipient checks are repeated in `requestDeposit()`, `requestDepositCancellation()`, `requestRedeem()` and the three `update*Recipient()` functions, and the request getters are implemented separately for each of the three queues. Internal helpers would reduce this duplication.
6. The comment `// Cancelled` in `_processPendingDepositCancellations()` marks the skipping of entries without shares. However, deposit cancellations cannot be cancelled, and their entries hold a nonzero amount of shares until they are fully processed and deleted. The check is therefore unreachable and the comment misleading. The same check in `_processPendingRedeems()`, commented `// Empty`, is reachable, as `requestRedeem()` queues zero shares if the redeem fee consumes the whole request.
7. In **Version 2**, deposit cancellations can be requested and processed while the vault is paused, but the NatSpec of `requestDepositCancellation()` and `processPendingDepositCancellations()`, as well as the README, still state that they revert while the vault is paused.

Code Partially Corrected

Points 2, 4.3, 6 and 7 were fully addressed.

Acknowledged

Spark acknowledged points 1, 3, 4.1, 4.2 and 5.

#004 No Slippage Protection for Requests

Informational **Version 1** **Code Partially Corrected / Acknowledged**

The request functions neither let users bound the outcome they accept nor reject outcomes the vault does not want, although both would be easy to enforce at request time:

1. `requestRedeem()` charges the redeem fee in force when the transaction is executed, which the `REDEEM_FEE_SETTER_ROLE` can raise, up to the maximum, right before. The vault could reject requests that are entirely consumed by the fee, and users could specify the maximum fee they accept.
2. `requestDeposit()` queues the `spUSDC` shares returned by `spUSDC.deposit()`, which rounds down and can return zero shares. The vault could reject zero-share requests, and users could specify the minimum number of shares they accept.

Additionally, zero-share requests leave dirty entries in the queues. They cannot be cancelled and their recipient cannot be updated, and processing skips them without deleting them, so they remain with their owner and recipient set.

With reasonable minimum request sizes, zero-share requests are unlikely, and the fee change is bounded by the maximum redeem fee.

Code Partially Corrected

In **Version 2**, zero-share redeem requests are processed and deleted, while zero-share deposit requests still remain in the queue. Points 1 and 2 remain unresolved.

Acknowledged

Spark acknowledged the remaining points. Fee changes are bounded by the maximum of 1%, and the minimum request sizes will be set to meaningful values.

#005 Paginated Getters Are Hard to Use

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

The vault exposes the first pending request ID of each queue, but not the length of the queue. As a result, the paginated getters `getPendingDepositRequests()`, `getPendingDepositCancellationRequests()` and `getPendingRedeemRequests()` cannot be used reliably:

- A `startId_` beyond the end of the queue causes the check `length_ <= queue.length - startId_` to underflow.
- A `length_` that extends past the end of the queue causes the same check to revert.

Code Corrected

The vault exposes the length of each queue, and the paginated getters clamp `length_` to the end of the queue.

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.