PUBLIC

Code Assessment

of the Spark PSM Smart Contracts

October 22, 2024

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

SparkDAO implements a peg stability module that supports three assets - two stable coins (USDC and USDS) and one yield-bearing wrapped stablecoin (sUSDS). That is intended to both stabilize the peg and offer liquidity on L2s.

The most critical subjects covered in our audit are functional correctness and precision of arithmetic operations. The general subjects covered are documentation, unit testing and gas efficiency. Security regarding all the aforementioned subjects is high.

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

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

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

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

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

Critical-Severity Findings	0
High-Severity Findings	1
• Code Corrected	1
Medium-Severity Findings	0
Low-Severity Findings	0

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Spark PSM 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	10 Sept 2024	000a72afd9daacd7d32b00bfa431f46ffbc5b353	Initial Version
2	02 Oct 2024	e7c806040161ab7e221fcffa0e3d82e351e64e65	After Fixes and Pocket Inclusion
3	05 Oct 2024	1e142338690c16c79fdccbd99b7dd8e963199769	Fix Pocket Accounting
4	21 Oct 2024	6a5a579cb503a461df254007064370f732fcd19d	Bump Version

For the solidity smart contracts, the compiler version `0.8.20` was chosen.

The files in scope were:

```
src/
    PSM3.sol
    interfaces/
        IPSM3.sol
        IRateProviderLike.sol
```

In version 2, the following file has been added to the scope:

```
deploy/
    PSM3Deploy.sol
```

2.1.1 Excluded from scope

All other files are out of scope. USDS and sUSDS are covered in separate reviews. USDC is out of scope and expected to work as documented. Note that the deployment script is in scope. However, governance should validate the deployment.

2.2 System Overview

This system overview describes the Version 2 of the contracts as defined in the Assessment Overview.

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

SparkDAO offers Spark PSM, a Peg Stability Module to facilitate the swapping, depositing, and withdrawing of three assets (both yield-bearing and non-yield-bearing assets).

Spark PSM works with three assets; USDC and USDS are stablecoins with a 1-to-1 conversion rate, and sUSDS is the yield-bearing counterpart of USDS. In the constructor, three assets and their precisions (decimals) are configured as immutables. An immutable `rateProvider` is also specified to retrieve the conversion rate between the yield-bearing asset and the non-yield-bearing ones.

The following entrypoints are provided for users to interact with the PSM:

- `deposit()`: Users can deposit one of the three assets into PSM to receive shares proportional to the current pool's valuation. The shares are purely for internal accounting and non-transferable.
- `withdraw()`: Users can withdraw one of the three assets with an expected `maxAssetsToWithdraw` from PSM by burning shares. The withdrawable amount is bounded by the PSM's current balance and user's total shares.
- `swapExactIn()`: Users can swap an exact amount of `assetIn` to another asset (`assetOut`). The conversion rate between USDC and USDS is always 1 (though rounding error may occur if USDC and USDS have different decimals). If it is converted to / from sUSDS, the conversion rate will be retrieved from the `rateProvider`. The user specified `minAmountOut` is used for slippage protection.
- `swapExactOut()`: Users can swap by specifying an exact amount of `assetOut`. It works similar to `swapExactIn()` while the spending token `assetIn` is bounded by `maxAmountIn`.

Users can preview the result of aforementioned entrypoints respectively with: `previewDeposit()`, `previewWithdraw()`, `previewSwapExactIn()`, `previewSwapExactOut()`.

In addition, the following getters are provided:

- `convertToAssets()`: Convert an amount of shares to a supported asset.
- `convertToAssetValue()`: Get the valuation (with 18 decimals) of an amount of shares.
- `convertToShares(uint256)`: Convert a specified `assetValue` to shares.
- `convertToShares(address, uint256)`: Convert an amount of supported asset to shares.
- `totalAssets()`: Estimate the valuation of PSM given three assets' balances (rounding down).

A pocket for USDC can be set (similar to PSMLite) to allow compatibility with Coinbase Custody. Namely, USDC shall be held in the `pocket` which can be set with function `setPocket` (callable by the owner of the contract). Note that pocket migration is supported.

2.2.1 Deployment

Further, deployment scripts have been added. Namely, the library PSM3Deploy supports a function `deploy` that deploys the contract and deposits 1 USDS to the PSM. Note that the deployment should be validated before any usage thereafter.

2.2.2 Roles & Trust Model

Since version 2, the Spark PSM has an owner (fully trusted) which can set the pocket. Otherwise, Spark PSM is permissionless without any privileged roles, and its other configurations are set as immutables upon deployment. It is assumed it will be configured correctly with appropriate parameters.

The following assumptions are made regarding the assets:

- All the assets used are assumed to be regular ERC20 tokens. Tokens with callbacks, fee on transfers, and rebasing mechanism should not be used. Otherwise the PSM may be subject to reentrancy and asset unfairness issues.

- It is assumed USDC and USDS are always valued equally. If one of them depegs, users will be in a race to withdraw the other one.
- It is assumed the sUSDS is the only yield-bearing token, and the `rateProvider` always reports the current fresh rate regarding USDS.

In addition, Spark PSM is subject the risks of the underlying assets, i.e., centralization, upgradeability, depegging. As of version 2, the pocket is fully trusted to only hold the funds. It is expected that full approval over the USDC funds is always given to the Spark PSM.

Users of Spark PSM are not trusted and could manipulate the reserve of Spark PSM by deposit, swap, and withdraw. Third party systems should be careful of this when integrate with Spark PSM.

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 Findings

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

- Security: Related to vulnerabilities that could be exploited by malicious actors

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 Findings section.

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

Critical-Severity Findings	0
High-Severity Findings	1

- Pocket Not Used on Interaction Code Corrected

Medium-Severity Findings	0
Low-Severity Findings	0
Informational Findings	1

- Gas Optimizations Code Corrected

6.1 Pocket Not Used on Interaction

Security High Version 2 Code Corrected

CS-SPRKPSM-002

The pocket is intended to escrow funds. However, during swaps, deposits and withdrawals funds are neither pushed nor pulled from the pocket. For example, function `deposit` pulls funds always to PSM3 itself. As a consequence, all funds that are held by a pocket are not retrievable and cannot be used, limiting the capabilities of the PSM. Withdrawals could be temporarily DoSed. However, by setting the pocket back to the PSM, the funds can be retrieved again.

However, more notably, the pocket's funds are never used for the valuation of shares. For example, the function `totalAssets`

```
function totalAssets() public view override returns (uint256) {
    return _getUsdcValue(usdc.balanceOf(address(this)))
        +  _getUsdsValue(usds.balanceOf(address(this)))
        +  _getSUsdsValue(susds.balanceOf(address(this)), false);  // Round down
}
```

never considers the pocket's funds. As a consequence, funds can be stolen due to a lack of correct accounting. However, such an issue is mainly applicable if funds were mainly held by the pocket (e.g. post-migration).

To summarize, a pocket can be set but is not actively used. The consequence of setting a new pocket is a potential DoS scenario and potential stealing of funds.

Code corrected:

Code has been changed to integrate with pocket:

1. Function `totalAssets` now queries USDC balance of pocket to ensure a correct valuation of shares.
2. Functions `_pullAsset` and `_pushAsset` will handle the transfers of USDC with pocket correctly.

6.2 Gas Optimizations

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

CS-SPRKPSM-003

The code could be optimized to reduce the gas consumption of PSM3. Below, is a non-exhaustive list of potential optimizations:

1. Function `previewDeposit` requires `_isValidAsset`. However, that is checked in `_getAssetValue`, too.
2. Function `previewWithdraw` requires `_isValidAsset`. However, that is checked in `_getAssetValue`, too.
3. Function `deposit` can credit shares to the receiver in an unchecked scope since `totalShares` is always greater.

Code corrected:

1 has been applied. 2 and 3 are not applied.

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 Loss of Precision

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

CS-SPRKPSM-001

In several places, the code performs division twice in a single computation, causing the result to be rounded down at each step. This introduces potential inaccuracies due to multiple rounding errors.

In function `_getAsset2Value`, the code performs two divisions when rounding down and two divisions when rounding up.

```
if (!roundUp) return amount * IRateProviderLike(rateProvider).getConversionRate()
    / 1e9 / _asset2Precision;

return Math.ceilDiv(
    Math.ceilDiv(amount * IRateProviderLike(rateProvider).getConversionRate(), 1e9),
    _asset2Precision
);
```

In functions `_convertToAsset2` and `_convertFromAsset2`, the code performs a division, followed by a multiplication, and then another rounding. The initial precision loss from the first division is amplified by the subsequent multiplication.

Note that performing all multiplication before both divisions shouldn't cause an overflow for assets within 18 decimals. For instance, in `_convertToAsset2`, assuming the precision of asset 2 is 18 decimals, then `amount * 1e27 * _asset2Precision = amount * 1e45` won't overflow unless amount is greater than `1e32`, which remains a sufficient amount for a token with 18 decimals in practice.

```
if (!roundUp) return amount * 1e27 / rate * _asset2Precision / assetPrecision;

return Math.ceilDiv(
    Math.ceilDiv(amount * 1e27, rate) * _asset2Precision, assetPrecision
);
```

Acknowledged:

SparkDAO is aware of the issue.

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 PSM Can Be Temporarily DoSed

Note Version 1

PSM can be temporarily DoSed by manipulating the funds it holds. An attacker can sandwich swaps and withdrawals by draining the desired output assets. Consequently, swaps may fail due to insufficient funds and withdrawals may only be partially filled.

Note that for partial filling of withdrawals, it could be possible to force the withdrawer to ultimately burn one more share due to rounding up. The withdrawer's loss could be amplified when this is repeated if the gas cost is less than the withdrawer's loss on L2.

8.2 Rounding of Total Assets

Note Version 1

The `totalAssets` function always rounds down the underlying value. However, in a few instances, the function could round up since its value is used as a divisor in operations where values should be rounded down.

Consider, function `previewDeposit` which should compute an amount of shares to mint that favours existing LPs. Note that rounding down is correctly done for the asset value (`_getAssetValue` used with `false`).

However, function `convertToShares` does not necessarily round properly for this use-case:

```
function convertToShares(uint256 assetValue) public view override returns (uint256) {
    uint256 totalAssets_ = totalAssets();
    if (totalAssets_ != 0) {
        return assetValue * totalShares / totalAssets_;
    }
    return assetValue;
}
```

Namely, `totalAssets` is a rounded down value that is used as a divisor. Consider the following example:

1. Assume that `assetValue` is 100 and `totalShares` is 101.
2. Assume that for `totalAssets`, the rounded down value is 100 and the rounded up value is 101.
3. The result will be either 101 shares (rounded down value) or 100 (rounded up value).

Thus, it becomes clear that due to the rounding down of `totalAssets`, the current depositor will be favoured and will receive more shares than one should receive.

8.3 Sandwiching Rate Updates

Note Version 1

Governance, LPs, and users should be aware of the potential abrupt rate changes from the `rateProvider` (no matter the rate is natively provided on L2s or bridged from L1). Thus, the rate may be under- or overestimated. As a result, rate updates may be sandwiched for profit.

8.4 Share Inflation

Note **Version 1**

Note that on deployment, the deployer must mint enough shares to ensure that no share inflation attack on the first depositor is possible. That is outlined in the `README.md`, too. Otherwise, the scenario below may be possible:

1. Alice mints 1 share with 1 value.
2. Bob wants to deposit 1M value.
3. Alice front-runs Bob's deposit with a donation of 1M.
4. Bob's transaction arrives. He mints 0 shares.
5. Alice profits.

Note that a similar scenario could occur (without inflating the shares) after the deployment. Consider the following scenario:

1. The contract is deployed.
2. Alice donates a small amount so that the value held is 1.
3. Bob deposits an arbitrary amount. Since, `totalShares == 0` but the amount of `totalAssets > 0`, the amount of minted shares will remain zero.
4. Alice calls withdraw and retrieves everything in the PSM.

Ultimately, the deployer must carefully deploy the contract and perform according actions directly after deployment (which includes safety checks on the results).