

Code Assessment of the SparkLendConduit Smart Contracts

October 30, 2023

Produced for



by



Contents

1	Executive Summary	3
2	Assessment Overview	5
3	Limitations and use of report	9
4	Terminology	10
5	Findings	11
6	Resolved Findings	13

1 Executive Summary

Dear all,

Thank you for trusting us to help MakerDAO with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of SparkLendConduit according to [Scope](#) to support you in forming an opinion on their security risks.

MakerDAO implements a conduit contract for funnelling sNST into Spark, an Aave v3 fork.

The most critical subjects covered in our audit are functional correctness and frontrunning resistance.

Functional correctness is high.

While the conduit `withdraw()` function can be frontrun, the function is only called by members of the SubDAO which are able to mitigate the risk, if necessary, by using more private channels for the inclusion of such transactions into the blockchain.

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

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

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

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

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

Critical -Severity Findings	0
High -Severity Findings	0
Medium -Severity Findings	0
Low -Severity Findings	3
• Code Corrected	1
• Code Partially Corrected	1
• Risk Accepted	1

2 Assessment Overview

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

2.1 Scope

The assessment was performed on the source code files inside the SparkLendConduit repository based on the documentation files.

All files in the `src` folder of the `sparklend-conduits` repository are part of the scope of this review. This includes:

1. `src/DailInterestRateStrategy.sol` (older versions)
2. `src/SparkConduit.sol` (`SparkLendConduit.sol` in newer versions)
3. various interfaces

Additionally, the following files of the `erc20-helpers` repository are in scope:

1. `src/SafeERC20.sol`

The table below indicates the code versions relevant to this report and when they were received.

sparklend-conduits

V	Date	Commit Hash	Note
1	17 September 2023	1085a2363de06347ad77a6051198b2d998bfabcf	Initial Version
2	9 October 2023	880d64b91c3f073739750d633246765dbe882dcb	Second Version
3	16 October 2023	2d559911963ca6e5fde88c46ff22ec7b2e515ead	Third Version
4	20 October 2023	729ba8c69e29da75f140f1abcaf649972eb47c7e	Fourth Version

erc20-helpers

V	Date	Commit Hash	Note
1	27 October 2023	8fe5ef3e85ea9ca5bc19df7f6ae605bc848647cc	Initial Version

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

2.1.1 Excluded from scope

Any other file not explicitly mentioned in the scope section. In particular tests, scripts, external dependencies, and configuration files are not part of the audit scope.

2.2 System Overview

This system overview describes the initially received version (**Version 1**) 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.



MakerDAO offers Spark Conduit, a conduit that AllocatorDAOs can use to supply funds as borrowable liquidity to the Spark lending platform.

Spark Conduit conforms to the standard Allocator Conduit interface, `IAAllocatorConduit` of `dss-allocator`, allowing integration in the `dss-allocator` system of SubDAOs.

Spark Conduit provides a way for AllocatorDAOs to supply pre-approved tokens as liquidity to the Spark lending protocol. The interest accrued by the supplied tokens is tracked by Spark Conduit, and funds plus accrued interest can be withdrawn by the lending AllocatorDAOs provided that enough liquidity is available in Spark. In case an AllocatorDAO which is supplying liquidity wants to withdraw, but no liquidity is currently available in Spark (due to borrowing), a withdraw request can be initiated which causes an increase in the interest rate for DAI/NSD borrowers, in order to incentivize the repayment of outstanding debt so that the requesting AllocatorDAO can withdraw its liquidity.

2.2.1 SparkConduit Contract

The *SparkConduit* contract is a conduit, meaning it implements the `IAAllocatorConduit` interface, which receives deposits from AllocatorDAOs and invests those deposits as liquidity in the Spark lending protocol, earning interest from the supplied liquidity.

Only AllocatorDAOs can access its general functionality, exposed through the functions `deposit()`, `withdraw()`, `requestFunds()`, `withdrawAndRequestFunds()`, and `cancelFundRequest()`. These functions are therefore guarded by the `ilkAuth` modifier which limits their use to addresses that have been authorized by the AllocatorDAOs themselves, identified by their `ilk` code, to interact with *SparkConduit* on their behalf.

The contract is deployed behind an upgradeable proxy.

2.2.1.1 deposit()

Deposits are made through the `deposit()` function. An `ilk`, an asset to be deposited and an amount are specified. The asset needs to be enabled and the `ilk` (belonging to its respective AllocatorDAO) is required to not have any active withdrawal requests. The amount of assets is transferred from the `ilk`'s buffer (obtained through the `dss-allocator` registry) to the *SparkConduit* contract, and supplied to Spark through the `supply()` method of the pool. The accounting of the amount of outstanding deposits for an `ilk` and asset is done through the emission of shares, which consist of the deposited amount divided by the interest accrual index of the Spark pool at the time of the deposit.

2.2.1.2 withdraw()

Withdrawals are attempted through a call to `withdraw()`, specifying the `ilk`, asset and maximum withdrawal amount denominated in that asset. The actual withdrawal is the minimum between the amount specified as parameter, the accrued balance of the `ilk`, and the available liquidity in the Spark reserve. The amount of assets is withdrawn from the Spark pool, and transferred to the `ilk`'s buffer. The amount to withdraw is also converted to amount of shares, and those are deducted from the `ilk`'s balance and the total amount for the asset. If the `ilk` has an outstanding share request amount, it is decreased by the corresponding shares that have been removed.

2.2.1.3 requestFunds()

If no liquidity is available, `ilks` can still signal their intention to withdraw by starting a fund request. This process triggers an interest rate increase that incentivizes borrowers to repay their debt.

`requestFunds()` accepts three arguments, the `ilk`, the asset, and the amount to request. It requires the current available liquidity of the asset in Spark to be empty (otherwise, regular withdrawals can be made instead). The amount requested is converted to shares. This is done to ensure that the `ilk` has enough shares to cover the request, and the amount is stored in the `requestedShares` mapping. A side-effect of requested shares is that `getInterestData()`, which is called by *DailyInterestRateStrategy*, returns values that indicate an unfavorable debt ratio. In this case, interest rates are adapted.

2.2.1.4 withdrawAndRequestFunds()

This method merges the functionality of `withdraw()` and `requestFunds()`. It accepts as arguments the ilk, the asset, and the amount, and first withdraws whatever is available from Spark, up to the requested amount or the balance of the ilk. If the whole amount cannot be withdrawn, the difference between the requested amount and the withdrawn amount is requested through `requestFunds()`.

2.2.1.5 getInterestData()

This method returns data used by *DaiInterestRateStrategy* to compute the borrow rate for DAI/NST in Spark. It contains the `subsidyRate` - MakerDAO's DAI savings rate - and a `baseRate` which additionally contains a `subsidyRate` that defines interest that is funneled to the MakerDAO. The `currentDebt` field is populated with the current amount of DAI/NST supplied to the SparkConduit, including accrued interest, and the `targetDebt` field is the total amount of DAI/NST supplied to the conduit, minus the amount requested for withdrawals.

2.2.1.6 Administrative functions

The *SparkConduit* contract can have multiple wards who are able to perform privileged actions. These actions consist of:

1. `rely()` a new ward or `deny()` an existing one.
2. Upgrading the implementation contract.
3. Updating the roles contract used for ilk authentication with `setRoles()`.
4. Updating the registry contract used to query an ilk's buffer with `setRegistry()`.
5. Updating the interest rate `subsidySpread` with `setSubsidySpread()`.
6. Enabling or disabling an asset with `setAssetEnabled()`. Disables assets cannot be deposited, but they still can be withdrawn.

The main functionality of the contract is permissioned and guarded through the `ilkAuth` modifier. `ilkAuth` queries the roles contract of `dss-allocator`, where the administrators of each ilk can specify which addresses are allowed to call specific functions on specific contracts. Each SubDAO can choose operators that can call some or all of the functions guarded by `ilkAuth`.

2.2.2 DaiInterestRateStrategy

The new *DaiInterestRateStrategy* is set to replace the currently deployed interest rate strategy for DAI (as well as for the new NST pool). It queries the Spark conduit for data about base interest rate, current DAI/NST supply from the AllocatorDAOs and target DAI/NST supply, and outputs the variable borrow rate and the lending rate through the `calculateInterestRates()` function. The variable borrow rate is selected such that it is simply the base rate plus a spread, when the target DAI/NST supply from the AllocatorDAOs is equal to the current supply, but it increases up to `maxRate` as the ratio of current supply over target supply becomes greater than 1 (i.e., shares have been requested). This increase in interest rate incentivizes borrowers to repay their debt so that the supply can ultimately be reduced.

The data to compute rate updates is fetched in the permissionless `recompute()` function which *should* be called any time the DSR or `requestedShares` change.

The contract doesn't have privileged roles and administrative functions.

2.2.3 Changes in Version 3

DaiInterestRateStrategy as well as the share request functionality of *SparkConduit* have been removed completely. The NST SparkLend pool now runs with a regular interest rate strategy and the sNST token (instead of the NST token) which tracks the Savings Rate directly.



2.2.4 Changes in Version 4

SparkConduit has been renamed to *SparkLendConduit*

2.2.5 Trust model and system assumptions

The wards of the contract are expected to act honestly towards depositors, as wards can potentially misappropriate funds deposited into the contract through contract upgrades or by changing the roles and registry settings.

Non-standard token implementations are assumed to be not supported. This includes tokens that:

1. Do not have a decimals field or have more than 18 decimals.
2. Do not revert and instead rely on a return value.
3. Implement fee on transfer.
4. Include rebasing logic.
5. Implement callbacks/hooks.
6. Revert on 0-approval.
7. Transfer different amounts than intended.

SubDAO operators are expected not to inflate the interest rate in order to game the system and damage the borrowers, by requesting funds and subsequently not withdrawing them.

It is assumed that only DAI and NST will be an enabled asset in the conduit contract.

It is further assumed that no accounts on SparkLend will receive the BRIDGE role so that no unbacked aTokens can be minted. Additionally, it is assumed that the respective pools will not have stable borrowing enabled.

Note: Since Version 3, it is assumed that only sNST will be used in the Conduit contract.

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
- **Design**: Architectural shortcomings and design inefficiencies

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

Critical -Severity Findings	0
High -Severity Findings	0
Medium -Severity Findings	0
Low -Severity Findings	2

- [Facilitators Have Incentive to Withdraw Funds](#) **Risk Accepted**
- [withdraw\(\) and requestFunds\(\) Can Be Prevented](#) **Code Partially Corrected** **Risk Accepted**

5.1 Facilitators Have Incentive to Withdraw Funds

Design **Low** **Version 1** **Risk Accepted**

CS-SPC-001

The allocation system assigns facilitator roles to some accounts chosen by the respective SubDAO. Facilitators can, amongst other things, call the ConduitMover contract which gives them access to the `SparkConduit.withdraw()` function.

Withdrawing all available liquidity from Spark increases the utilization of the pool to 100%. Since utilization is a factor of the supply rate of the DAI/NST pools, and because third party supplying is allowed on these pools, facilitators that have an open supply position on the pool can increase their interest rate by withdrawing funds.

Risk accepted:

MakerDAO accepts the risk giving the following statement:

This will be mitigated through Maker disincentivizing this behaviour.

5.2 withdraw() and requestFunds() Can Be Prevented

Security **Low** **Version 1** **Code Partially Corrected** **Risk Accepted**

CS-SPC-005

External attackers can conduct Denial of Service attacks against the conduit by targeting `withdraw()` and `requestFunds()` requirements.



An attacker can supply 1 wei of liquidity to an aToken whose reserve balance is otherwise empty, and prevent `requestFunds()` from being callable.

Similarly, an attacker with enough collateral balance can borrow all the available liquidity before `withdraw()` or `withdrawAndRequestFunds()` operations from the SubDAOs, and repay it just after, preventing the SubDAOs from withdrawing their funds, while incurring little interest accrual since the debt is only held for the time of a few blocks.

An economic incentive for these attacks could be present if the attacker is also a third-party supplier. In that case, it could be within their interest to keep the interest rates high by preventing SubDAOs to withdraw after a `requestFunds()` has been triggered.

Code partially corrected:

The functions `requestFunds()` and `withdrawAndRequestFunds()` no longer exist.

Risk accepted:

Client states they will submit transactions that will not be frontrun in this way.

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	0
Medium -Severity Findings	0
Low -Severity Findings	1
• Withdrawer Can Steal 1 Wei Code Corrected	
Informational Findings	6
• Gas Optimizations Code Corrected • Floating Pragma Code Corrected • Missing Event Code Corrected • Inaccurate Naming and Comments Code Corrected • Outdated Aave Version Used Code Corrected • subsidySpread Overflow Code Corrected	

6.1 Withdrawer Can Steal 1 Wei

Design **Low** **Version 1** **Code Corrected**

CS-SPC-004

When withdrawing, an amount of tokens is specified and the corresponding amount of shares is deducted from the ilk's balance. Since `_convertToShares()` rounds down in its division, a too small amount of shares will be deducted. Specifically, if the ilk withdraws 1 wei, 0 shares will be deducted (since the index is greater than 1).

Code corrected:

When `withdraw()` is called, `_convertToSharesRoundUp()` is now used, which rounds up the amount of shares to deduct, removing the possibility of 1 wei stealing.

6.2 Floating Pragma

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

CS-SPC-006

The contracts have a floating pragma of `^0.8.13` and there is no fixed compiler version in `foundry.toml`. To make sure that the contracts are always compiled in a predictable manner, the pragma should be fixed to a stable compiler version.

Code corrected:

Solidity version has been fixed to 0.8.20 in foundry.toml.

6.3 Gas Optimizations

Informational Version 1 Code Corrected

CS-SPC-002

`withdrawAndRequestFunds()` is guarded by the `ilkAuth` modifier, and calls internally `withdraw()` and `requestFunds()` which are also guarded by `ilkAuth`. This causes `ilkAuth` to be evaluated at most 3 times within a call of `withdrawAndRequestFunds()`, which is inefficient in terms of gas, since `ilkAuth` includes an external call and several SLOADs.

Function `cancelFundRequest()` requires an unnecessary SLOAD when decreasing `requestedShares` instead of setting it to 0 directly.

`withdrawAndRequestFunds()` queries `getAvailableLiquidity()` twice, once in its own function body, and then again in `withdraw()`. If `withdraw()` would return early when `getAvailableLiquidity() == 0`, or generally when the amount computed at line 133 equals 0, a single querying of the liquidity would be sufficient.

Code corrected:

All mentioned functions have been removed.

6.4 Inaccurate Naming and Comments

Informational Version 1 Code Corrected

CS-SPC-003

In *SparkConduit*:

1. **The naming of the `_totalWithdrawals` return parameter of `getAssetData()` is ambiguous**
as it represents the total requested funds.
2. **The naming of the `_requestedShares` return parameter of `getPosition()` is inaccurate**
as it doesn't represent a share amount but a token amount.

In *DailInterestRateStrategy*:

1. The comment describing the contract references D3M, but the contract will be used in the context of the allocator system which sunsets D3M.
-

Code corrected:

1. `_totalWithdrawals` has been renamed to `_totalRequestedFunds`.
2. `_requestedShares` has been renamed to `_requestedFunds`.



DaiInterestRateStrategy still contains a comment about D3M.

6.5 Missing Event

Informational Version 1 Code Corrected

CS-SPC-007

`DaiInterestRateStrategy.recompute()` changes the storage but does not emit an event.

Code corrected:

The event `Recompute` is now emitted in `recompute()`.

6.6 Outdated Aave Version Used

Informational Version 1 Code Corrected

CS-SPC-008

The repository currently uses the Aave v3 version 1.17.2. The version still contains a bug that automatically enables tokens with an LTV of 0 as collateral as soon as they are sent to an address. This can be problematic in cases when the recipient holds a borrowing position as it prevents the withdrawal of any tokens with an LTV greater than 0.

While the `SparkConduit` contract currently does not hold a borrowing position, this might be changed in the future. In this case, the Aave version should be updated to prevent DoS attacks by simply sending 1 wei of `aTokens` to the contract.

Code corrected:

The Aave submodule commit hash has been updated to the `v1.18.0` version.

6.7 subsidySpread Overflow

Informational Version 1 Code Corrected

CS-SPC-009

`SparkConduit.setSubsidySpread()` does not contain a check to verify that `subsidySpread` is small enough to fit into a `uint128` variable when added up to the DSR rate. Therefore, it may be possible that the following line in `getInterestRate()` overflows on unsigned downcast:

```
baseRate: uint128(dsr + subsidySpread)
```

Code corrected:

`subsidySpread` is no longer used.