PUBLIC

Code Assessment

of the Sparklend Cap Automator Smart Contracts

February 15, 2024

Produced for

by

Contents

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 Sparklend Cap Automator according to Scope to support you in forming an opinion on their security risks.

SparkLends CapAutomator manages supply and borrow caps for assets in SparkLend. It allows anyone to trigger updates to these caps based on predetermined parameters, with the new cap values calculated using the current supply and a specified gap, subject to maximum limits and cooldown periods.

The most critical subjects covered in our audit are functional correctness, manipulation resiliency and the integration of the CapAutomator into the existing SparkLend protocol. A notable issue was identified in the original code where setting caps to zero is not restricted, leading to the potential bypass of the cooldown period and risks of lifting the cap (see Cap of 0 ignores increase cooldown).

After the intermediate report, all identified issues have been addressed or acknowledged.

The general subjects covered are optimizations and adherence to the specifications.

In summary, we find that the codebase provides a good 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
• Risk Accepted	2

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 Sparklend Cap Automator 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 Jan 2024	b88ea670f39f659bcf6785a1ff92969dad3dcaf7	Initial Version
2	13 Feb 2024	07bc65499da8d59abaeb32164b00a3489922a62a	After Intermediate Report

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

The following contracts are in the scope of this review:

```
src/CapAutomator.sol
src/interfaces/ICapAutomator.sol
```

2.1.1 Excluded from scope

Any file not explicitly listed above as well as third-party libraries are out of the scope of this review.

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.

SparkLends CapAutomator implements the functionality to manage supply and borrow caps for assets in SparkLend. Governed by the set configuration parameters, anyone can trigger the update of the respective cap. The new cap value is calculated as the current supply + intended gap, limited by the max limit set. Increasing the supply is limited by a cooldown period. Update of a specific cap can only happen once per block.

Technically each CapConfiguration (both borrow and supply configurations) are stored in a CapConfig struct:

```
struct CapConfig {
    uint48 max;              // Full tokens
    uint48 gap;              // Full tokens
    uint48 increaseCooldown; // Seconds
    uint48 lastUpdateBlock;  // Blocks
```

```
    uint48 lastIncreaseTime; // Seconds
}
```

Upon deployment the contract is initialized with the Pool and PoolConfigurator addresses derived from the provided poolAddressesProvider. The deployer (`msg.sender`) is set as the initial owner of the contract.

Public Functions:

Triggering the update of the caps is designed to be permissionless.

- `execSupply()`: Triggers the update of the SupplyCap of the given asset.
- `execBorrow()`: Triggers the update of the BorrowCap of the given asset.
- `exec()`: Triggers the update of both, first the SupplyCap and then the BorrowCap for the given asset.

View Functions:

View functions return contents of the respective mapping entry for the configuration of the specified asset.

- `supplyCapConfigs()`
- `borrowCapConfigs()`

Furthermore public getters for the `pool`, `poolConfigurator` and the `owner` address are provided.

Permissioned owner only functions:

- `setSupplyCapConfig()` & `setBorrowCapConfig()`: Enables the owner to set or update cap configurations for supply and borrow, respectively, for specific assets.
- `removeSupplyCapConfig()` & `removeBorrowCapConfig()` : Allows the owner to remove cap configurations for specific assets.

Ownership is implemented using OpenZeppelin's Ownable implementation. Functions `renounceOwnership` and `transferOwnership` are present to modify the ownership.

The contract emits events for setting or removing cap configurations (`SetSupplyCapConfig`, `SetBorrowCapConfig`, `RemoveSupplyCapConfig`, `RemoveBorrowCapConfig`) and for updating caps (`UpdateSupplyCap`, `UpdateBorrowCap`).

2.2.1 Trust Model & Roles

- Owner: Fully trusted. Expected to be the Governance Multisig. Can update the configurations of all assets, including the max cap, gap and cooldown period between increases. Incorrect actions can severely impact SparkLend (i.e. all future borrows of an asset could be paused in case the owner sets `CapConfig.max` of the asset to a very small value).
- Permissionless Keeper: Untrusted role. This role is responsible for triggering updates as needed.

For a contract to update values in the PoolConfigurator, it must possess specific privileges. These privileges are held by two roles: the `RISK_ADMIN` and the `POOL_ADMIN`. The `POOL_ADMIN` role encompasses all privileges of the `RISK_ADMIN`. To adhere to the principle of least privilege, it's advised to assign the minimal level of access necessary, which would be the `RISK_ADMIN` role.

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
- Correctness: Mismatches between specification and implementation

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

- CurrentSupply Can Be Underestimated Risk Accepted
- DoS Cap Increase Risk Accepted

5.1 CurrentSupply Can Be Underestimated

Design Low Version 1 Risk Accepted

CS-SPRKCAP-001

CapAutomator computes the new SupplyCap based on the current supply, the gap, and the max. The current supply is estimated based on the scaled aToken total supply, the cached amount that should be accrued to treasury, and the last updated liquidity index.

```
uint256 currentSupply = (
            IScaledBalanceToken(reserveData.aTokenAddress).scaledTotalSupply()
            + uint256(reserveData.accruedToTreasury)
        ).rayMul(reserveData.liquidityIndex)
        / 10 ** ERC20(reserveData.aTokenAddress).decimals();
```

The `liquidityIndex` could be underestimated if it hasn't been updated up to now (interest has not been accrued for the period between `reserveLastUpdateTimestamp` and `block.timestamp`).

In addition, decimals have been removed for `currentSupply` to align with gap and cap, while this rounds `currentSupply` down.

As a result, `currentSupply` could be underestimated and influence the new cap computation (`_calculateNewCap()`).

Risk accepted:

MakerDAO states:

```
We acknowledge and accept the fact that the current total value of the pool can
be assumed slightly inaccurately.
```

5.2 DoS Cap Increase

Security | Low | Version 1 | Risk Accepted

CS-SPRKCAP-002

CapAutomator allows any user to increase the supply cap by `gap` if the `increaseCooldown` has passed.

```
// Cap cannot be increased before cooldown passes, but can be decreased
if (newCap > currentCap
    && block.timestamp < (capConfig.lastIncreaseTime + capConfig.increaseCooldown)) {
        return currentCap;
```

Furthermore, the computation of the new cap relies on the current amount of liquidity supplied or borrowed:

```
uint256 newCap = _min(currentValue + capConfig.gap, max);
```

A malicious user supplying or borrowing funds can frontrun a call to `execSupply()`, `execBorrow()` or `exec()` to remove supply or repay his loan in order to manipulate `currentValue`.

More precisely, it is possible to manipulate `currentValue` to increase the supply or borrow cap by 1 instead of `gap`. It is then not possible to increase the cap again before `increaseCooldown` expires.

Selecting the values of `increaseCooldown` and `gap` with care is crucial, as poor choices can worsen the problem, although even carefully selected values cannot fully eliminate this issue.

Risk accepted:

MakerDAO states:

```
The same vector is present in other Maker modules and it was never proved to be
used. In case of users abusing this possibility, we will upgrade to a more
robust solution.
```

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

6.1 Cap of 0 Ignores Increase Cooldown

Correctness Low Version 1 Code Corrected

CS-SPRKCAP-008

CapAutomator intends to prevent the supply and borrow cap to be increased for `increaseCooldown` seconds after an increase.

```
if (newCap > currentCap &&
    block.timestamp < (capConfig.lastIncreaseTime + capConfig.increaseCooldown)) {
    return currentCap;
```

In SparkLend, a cap of 0 means no cap at all (see SparkLend Caps Specifications). In other words, the cap becomes infinity.

In CapAutomator, a `newCap` equal to 0 is considered as a cap decrease instead of an increase from the condition (`newCap > currentCap`). As a result, the `increaseCooldown` period will not be taken into account when the `newCap` is 0.

The aforementioned behavior could happen in case the `gap` is set to 0 and there is no supply or borrows (`currentValue==0`).

```
uint256 newCap = _min(currentValue + capConfig.gap, max);
```

In summary:

1. The code does not eliminate the possibility of setting `gap` to 0, which implies the risk of lifting the cap by setting cap to 0.
2. The `increaseCooldown` period will not be respected per specification in case the cap becomes infinity (`cap==0`).

Code corrected:

An extra check has been added in `setSupplyCapConfig()` and `setBorrowCapConfig()` to ensure the gap is greater than 0.

6.2 Event Emission Before State Change

Informational | Version 1 | Code Corrected

CS-SPRKCAP-006

In functions `_updateSupplyCap` and `_updateBorrowCap`, events are emitted before the configs (state variables) are updated. The emission of an event should indicate that changes have been made to state variables, whereas by the time the event is emitted, state changes haven't been made yet.

Code corrected:

Emission of the event `UpdateSupplyCap` and `UpdateBorrowCap` has been moved to the end of `_updateSupplyCap()` and `_updateBorrowCap()` respectively.

6.3 Remove Non-Existing Config Is Possible

Informational | Version 1 | Code Corrected

CS-SPRKCAP-007

The owner of CapAutomator can remove the supply or borrow caps via `removeSupplyCapConfig()` and `removeBorrowCapConfig()` respectively. Both of them do not check if the config has been set in the first place. As a result, removing non-existing config will succeed and an event will be emitted even though there is no state changes.

Code corrected:

`removeSupplyCapConfig()` and `removeBorrowCapConfig()` have been updated to ensure the config exists by checking if `config.max>0`. As `config.max` can only be set to a positive value for valid configs, this check effectively prevents removing a non-existing config.

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 Cap Can Be Updated in the Flashloan Callback

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

CS-SPRKCAP-004

SparkLend supports flashloan (`flashLoan()`), at the end of which, a debt position (`executeBorrow()`) will be opened for the user if one does not repay the flashloan. In case the borrow limit has been reached, `executeBorrow()` will fail and the whole flashloan will revert.

With the deployment of `CapAutomator`, the user can increase the borrow cap in the flashloan callback via `execBorrow()` in case the borrow limit has been reached. Note that updating the borrow cap first and then doing the flashloan implies the same execution result, nevertheless, the ability to change of a key pool parameter may not be expected to happen in a flashloan callback.

7.2 Cap May Decrease Fast and Recover Slowly

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

CS-SPRKCAP-005

Situations may arise where a cap can be reduced significantly. E.g. a whale or many borrower repay significant amounts or significant amounts of supply are withdrawn at the same time.

Should the CapAutomator be activated it will significantly reduce the respective cap, leaving only gap amount available for use.

Bringing the respective cap back up to its original limit could take a considerable amount of time due to the increaseCooldown feature.

This is especially likely to happen in case a reserve is frozen, no more supply and borrow is possible, but users can withdraw and repay. This means it is possible that the current supply and borrow will decrease to a small value, hence the cap could be decreased significantly. Arguably a significant reduction in the cap coupled with a slow increase over a period of time may not be unwarranted. Special care needs to be taken in this scenario.

7.3 Gas Optimizations

Informational **Version 1**

CS-SPRKCAP-003

When updating the supply/borrow cap (`_updateSupplyCap()`, `_updateBorrowCap()`), the calculation of the new cap (`_calculateNewCap()`) will return early if the configuration is unset or the cap has already been updated in the current block.

```
if (max == 0 || capConfig.lastUpdateBlock == block.number) return currentCap;
```

This check could be brought forward in `_updateSupplyCap()` and `_updateBorrowCap()` to enable an early return, which would occur before the external call to `pool.getReserveData()` is executed, saving gas.

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 Supply and Borrow Are Subject to Front-Running

Note Version 1

When the current available volume (to be supplied / borrowed) is above the gap, a user's action to consume the available volume can be front-run by an update of the cap, which may leave insufficient volume to be consumed in case the cap is lowered. In the former case, users can only supply / borrow up to the current cap and wait at least cooldown period to increase the cap for the consecutive operations. For instance:

- Assume the current borrow cap = 15, current total borrow = 3, gap = 5.
- Alice send tx1 to borrow 10.
- Bob front-runs tx1 to update the cap to 8.
- Now tx1 will fail due to exceeding the cap.

In addition, it is obvious that user's operation (supply / borrow) can also be front-run by others' (supply / borrow) which may leave insufficient volume afterwards.

8.2 Token Unit, Decimals and Integer Division

Note Version 1

In CapAutomator, the new cap is determined by `currentValue`, `gap` and `max`. These value are in "full tokens", token amounts without decimals.

```
uint256 newCap = _min(currentValue + capConfig.gap, max);
```

The `currentValue` is computed as `currentSupply` in `_updateSupplyCap()`:

```
uint256 currentSupply = (
            IScaledBalanceToken(reserveData.aTokenAddress).scaledTotalSupply() +
            uint256(reserveData.accruedToTreasury)
        ).rayMul(reserveData.liquidityIndex)
        / 10 ** ERC20(reserveData.aTokenAddress).decimals();
```

and it is computed as `currentBorrow` in `_updateBorrowCap()`:

```
uint256 currentBorrow = ERC20(reserveData.variableDebtTokenAddress).totalSupply()
        / 10 ** ERC20(reserveData.variableDebtTokenAddress).decimals();
```

In both cases `currentValue` is rounded down when removing the decimals.

Due to this, the difference between the new cap and the actual current value could be below the gap even though the max has not been reached.

In practice, these minor rounding effects are typically negligible. Nonetheless, they can result in the following consequences:

- If `gap` is 0, the new `cap` could be set lower than the current amount. For instance, if `gap` is 0, `cap` is 10, and the actual current supply is 9.99. The calculation of `currentSupply` would result in 9 due to integer division. Consequently, the update would set the `cap` to 9, which is less than the actual current supply.
- Users should not expect to fully utilize the `gap` amount (expanded to its decimal representation) after triggering `exec()`. There may be less space due to rounding and hence this action might fail.