PUBLIC

Code Assessment

of the Sturdy Aggregator Smart Contracts

October 13, 2023

Produced for

STURDY

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Sturdy team,

Thank you for trusting us to help Sturdy with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Sturdy Aggregator according to Scope to support you in forming an opinion on their security risks.

Sturdy implements Sturdy Aggregator, a lending optimizer with the ability to provide just-in-time liquidity by moving funds between different lenders.

The most critical subjects covered in our audit are functional correctness, asset solvency, and access control. Security regarding all the aforementioned subjects is high.

The general subjects covered are specification and gas efficiency. Security regarding the aforementioned subjects is high.

Note that the zkAllocation is not specified precisely and is treated as a black box.

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	6
• Code Corrected	5
• Risk Accepted	1
Low-Severity Findings	3
• Code Corrected	1
• Specification Changed	1
• Acknowledged	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 Sturdy Aggregator 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	14 Sept 2023	6ee0b32613ae239c349399690392d56ab24faa7b	Initial Version
2	02 Oct 2023	a218c477b172d1a073826d4e3535093554d97737	Version 2 with fixes
3	03 Oct 2023	abcdf18428f0eaff5ac441c339ffa0c3ab52d05a	Version 3 with fixes

For the solidity smart contracts, the compiler version `0.8.18` was chosen.

The following contracts are in the scope of the review:

```
core:
    SiloGateway.sol
    DebtManager.sol
```

2.1.1 Excluded from scope

Any contracts not explicitly listed above are out of the scope of this review, especially `VaultV3`, any implementation of `ISilo` or other lenders, and the ZK verifier. 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.

Sturdy offers Sturdy V2 Aggregators, in which users can lend a single asset to a Yearn-style lending optimizer, which in turn deposits the assets to whitelisted silos. The lenders interact with the aggregator, which is a vault similar to Yearn's `VaultV3`, linked to a `DebtManager`. The borrowers interact with the `SiloGateway`, which can move assets between Silos.

The scope of this review is limited to the `SiloGateway` and the `DebtManager`.

2.2.1 Silo Gateway

The `SiloGateway` allows borrowers to borrow assets from a silo. If the silo does not already have enough assets, the function `borrowAsset` will request just-in-time liquidity from the `DebtManager`. The amount requested will be such that the target silo's utilization does not exceed the configured utilization limit after borrowing. After the liquidity has been added, the `SiloGateway` will pull the collateral token from the `msg.sender`, and borrow the amount of asset tokens against the collateral for the provided `_receiver`.

2.2.2 Debt Manager

The `DebtManager` contract is linked to one aggregator (`VaultV3`) and is responsible for managing the token allocation to the different lenders, some of them being silos. The lenders are expected to be ordered by their APR, given by an APR oracle. The `admin` can sort the lenders whenever necessary by calling `sortLenderwithAPR()`.

Just-in-time liquidity: When a `SiloGateway` requests just-in-time liquidity, the needed liquidity will be withdrawn from other lenders following their ordering, i.e., from lowest to highest APR. If the needed amount cannot be withdrawn, the transaction will revert.

Token allocation: There are two whitelisted addresses that can change the token allocation of the aggregator between the different lenders: the contract's owner and a ZKverifier. The contract owner is a privileged address that can set arbitrary debt amounts for the lenders. The ZKverifier can be called by anyone. It will reallocate tokens whenever someone can prove that a certain allocation strategy will give a better yield than the current one. When reallocated, tokens can flow only between active lenders and within the maximum debt boundaries defined by the vault. Note that the DebtManager may have only a subset of the strategies that exist in the aggregator added as Lenders. As a result, JIT liquidity can only be pulled from those strategies.

2.2.3 Trust Model

Users: not trusted Debt manager's owner: trusted to add/remove lenders, set the APR oracle address, whitelist lenders, and correctly fill the `_pairToLender` mapping in a non-adversarial manner. Can sort the lenders' array whenever it is needed. Trusted to manually set token allocation in a non-adversarial manner. Silo gateway's owner: trusted to set the utilization limit in a non-adversarial manner. Lenders: lenders must be carefully chosen by Sturdy and are trusted to act non-maliciously. ZKverifier: assumed to only call `zkAllocation` with inputs that increase the APR of the system. The implementation and guarantees of the ZKproofs are not currently known and are outside the scope of this review.

2.2.4 Changes in V2

- Each lender has a utilization limit that can be set in the `DebtManager` instead of one global limit in the `SiloGateway`
- When requesting liquidity, the amount withdrawn from each lender is limited so their utilization limit is respected.

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:

- 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	1

- zkAllocation May Not Behave as Expected Risk Accepted

Low-Severity Findings	1

- borrowAsset() Slippage Protection Acknowledged

5.1 zkAllocation May Not Behave as Expected

Design Medium Version 1 Risk Accepted

CS-STUAGG-004

The `zkAllocation` function is assumed to only be called with a lender allocation that increases the total APR of the aggregator.

However, the possible allocations depend on the state of the blockchain at execution time, which is likely impossible to know at proof generation time. In particular, the `aggregator.update_debt` function gives no guarantees on how much it will withdraw or deposit when it is called with a certain target debt. It may deposit/withdraw more or less than expected, depending on the current state.

In general, the aggregator will try to get "as close as possible" to the target debt, but will not revert even if far away from the target. For example, a call that tries to reduce debt by `100`, but due to tokens being locked in the strategy, only reduces debt by `1`, will not revert. However, there will be a revert if there is a call that would deposit or withdraw a zero amount.

Consider the following illustrative example:

1. There are two lenders, `A` and `B`. Both have a debt of `100`. The `minimum_total_idle` of the `aggregator` is set to `10`. There are `210` tokens in the `aggregator` in total.
2. The interest rates change such that `A` now has a slightly lower interest rate than `B`.
3. A zk proof is generated, that claims that a better allocation of tokens would be `90` tokens in `A`, and `110` tokens in `B`. This is true at proof generation time.
4. Someone withdraws `5 tokens` from the `aggregator`.
5. The zkVerifier verifies the proof, and calls `zkAllocation()`.
6. `update_debt(A,90)` is called. It was expected at proof generation time that this withdraws `10` tokens. However, since then, assume the internal balances of `A` have changed, and only `7` tokens are withdrawable. `7` tokens are added to the `total_ idle`.

7. `update_debt(B,110)` is called. It was expected at proof generation time that this deposits `10` tokens. However, now the `total_idle` of the aggregator is only `12` and the minimum is `10`, so only 2 tokens are deposited to `B`.
8. `zkAllocation()` successfully returns. Now the balances are `A: 93` and `B: 102`. This has a lower APR than if there had been no change and the balances had stayed `A: 100` and `B: 100`.

There can be many lender-specific conditions that limit how much can be added/withdrawn. These conditions can be dependent on the current state of the blockchain, with no way to know the limits in advance. As the allocation zk proofs must be generated ahead of execution time, it does not seem possible that they can take all of these limits into account. This may lead to cases where a zk proof verifies, but the resulting APR is lower. A malicious actor may even be able to frontrun the `zkAllocation` call to change the state such that the allocation becomes worse.

Zk proof generation should also consider the effects of `process_report`, which can change `current_debt`, otherwise `update_debt` may lead to more tokens deposited to that lender than expected.

The severity of this issue depends on what exactly is proven in the zk proofs, which is out of the scope of this audit and is treated as a black box.

Risk accepted:

Sturdy understands and accepts the risk.

Sturdy responded:

```
The time period between proof generation and execution time will be quite small, so changes are unlikely.
Given that there is no risk of lost funds (only suboptimal yield), we're accepting this risk for the time being
and will consider lender-specific limits in the future.
```

5.2 `borrowAsset()` Slippage Protection

Design | Low | Version 1 | Acknowledged

CS-STUAGG-010

`Silo.borrowAsset()` returns the amount of shares debited when borrowing. However, this value is ignored by `SiloGateway.borrowAsset()`. The received amount of shares may be smaller than expected.

Acknowledged:

Sturdy acknowledges and understands the issue. Sturdy states:

```
The value to be compared depends on the external silo's logic (ex: Fraxlend, Aave V3, Compound V3).
Slippage protection will be added where needed.
```

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	5

- Sorting of the Lenders Is Incorrect Code Corrected
- Idle Assets Not Used for requestLiquidity Code Corrected
- Utilization Limit Does Not Take Into Account JIT Liquidity Code Corrected
- Utilization Limit Only Enforced on Requesting Lender Code Corrected
- utilizationLimit Is Not Always Enforced Code Corrected

Low-Severity Findings	2

- Incorrect Code Comment Specification Changed
- Reentrancy Guards Applied Inconsistently Code Corrected

Informational Findings	3

- Gas Optimizations Code Corrected
- Missing Input Sanitization Code Corrected
- Misleading Error Names Code Corrected

6.1 Sorting of the Lenders Is Incorrect

Correctness Medium Version 2 Code Corrected

CS-STUAGG-013

The `sortLenderWithAPR` function was updated in Version 2.

The algorithm that sorts the lenders by APR only swaps the lenders' addresses in the array, but the positions in the new APRs array are not swapped. This leads to the list of APRs being out of sync with the list of lenders, which leads to incorrect sorting comparisons.

Example result of the implemented algorithm:

- lenders = [A, B, C, D]
- aprs = [0, 1, 0, 0]
- result of the sorting: [A, D, B, C]
- correct result should have B at the end of the array

Code corrected:

The codebase has been updated so that the array of APRs is also updated along with the array of lenders, fixing the issue.

6.2 Idle Assets Not Used for requestLiquidity

Design Medium Version 1 Code Corrected

CS-STUAGG-001

In `DebtManager`, `requestLiquidity()` has the following check:

```
if (requiredAmount > totalIdle) {
        unchecked {
            requiredAmount -= totalIdle;
        }
    }
```

This will use idle liquidity to partially fulfill a request, but only if the `requiredAmount` is more than the idle amount.

If there are enough idle assets to cover the entire `requiredAmount`, they will not be used at all.

Code corrected:

The code has been updated such that if the `totalIdle` amount is greater or equal to the `requiredAmount`, the idle assets will be used and nothing will be pulled from other lenders.

6.3 Utilization Limit Does Not Take Into Account JIT Liquidity

Design Medium Version 1 Code Corrected

CS-STUAGG-002

When borrowing from a silo, even if JIT liquidity can be performed, there is a limit on the utilization of the silo before the liquidity transfer. The utilization of the silo, before JIT liquidity is taken into account, cannot exceed 100%.

Example:

1. Silo A has 100k, silo B has 900k.
2. A user wants to borrow 300k from silo A, but this will revert since the computed utilization rate will be 3 * `PREC_UTIL` (300%).

Code corrected:

The utilization limit check before JIT liquidity is taken into account has been removed.

6.4 Utilization Limit Only Enforced on Requesting Lender

Design | Medium | Version 1 | Code Corrected

CS-STUAGG-014

When calling `borrowAsset()`, the utilization limit is only enforced on the requesting silo, but not on the other lenders, which can be fully utilized if JIT liquidity is used.

There could be situations where all the silos are fully utilized but one.

Code corrected:

The utilization limit can now be set per lender in DebtManager and is enforced in `borrowAsset` in the requesting lender, but also in the lenders from which the liquidity is being pulled.

6.5 `utilizationLimit` Is Not Always Enforced

Correctness | Medium | Version 1 | Code Corrected

CS-STUAGG-003

In `borrowAsset()`, `requestLiquidity()` is called with the `_amount` that should be deposited to the `Silo` such that the `utilizationLimit` is respected. However, `requestLiquidity()` can deposit a smaller amount than what is expected.

The amounts that are withdrawn from other lenders by the aggregator are calculated as follows:

```
newDebt = aggregator.update_debt(lenders[i], newDebt);
        unchecked {
            withdrawAmount = lenderData.current_debt - newDebt;
        }
```

This does not always correctly calculate the `withdrawAmount`. When withdrawing from a lender, there can be an unexpected loss. In this case, the `withdrawn` amount will be smaller than the change in debt.

Consider the following excerpt from `VaultV3`, which is the implementation of `aggregator`:

```
# making sure we are changing idle according to the real result no matter what.
# We pull funds with {redeem} so there can be losses or rounding differences.
withdrawn: uint256 = min(post_balance - pre_balance, current_debt)

# If we got too much make sure not to increase PPS.
if withdrawn > assets_to_withdraw:
    assets_to_withdraw = withdrawn

# Update storage.
self.total_idle += withdrawn # actual amount we got.
# Amount we tried to withdraw in case of losses
self.total_debt -= assets_to_withdraw

new_debt = current_debt - assets_to_withdraw
```

The `withdrawAmount` value that should be calculated in `requestLiquidity()` is actually `withdrawn`, the change in `total_idle`.

If there is a loss while withdrawing from the lender, an insufficient amount of `totalIdle` will be available when depositing to the `Silo`. In most cases this will lead to a higher utilization than `utilizationLimit`, but in cases where losses are large or `utilizationLimit` is configured to be close to `100%`, the `Silo.borrowAsset()` call at the end of `SiloGateway.borrowAsset()` will revert, as there will not be enough funds in the Silo.

Version 2:

The code now uses an accurate amount of tokens to reduce `requiredAmount`. However, two conditions have been added in the `DebtManager.requestLiquidity()` logic.

One early-return check in the `for` loop:

```
if (requiredAmount < minIdle) break;
```

And one require check after the `for` loop:

```
require(requiredAmount <= minIdle, Errors.AG_INSUFFICIENT_ASSETS);
```

Recall that `requiredAmount = amount + minIdle`. If `requiredAmount` is greater than `0`, it means that the current idle amount is smaller than `amount + minIdle`. When updating the debt of the requesting lender, the aggregator will still keep `minIdle` and the amount sent to the lender can be smaller than `amount`. This would lead to the requesting lender exceeding its `utilizationLimit`.

Code corrected:

The codebase has been updated so that `DebtManager.requestLiquidity()` compares the aggregator's `totalIdle` before and after the call to `update_debt()` to know exactly how many tokens have been withdrawn from the lender.

The early-return check has been removed and the require check corrected to

```
require(requiredAmount == 0, Errors.AG_INSUFFICIENT_ASSETS);
```

which ensures that enough assets have been retrieved.

6.6 Incorrect Code Comment

Correctness **Low** **Version 1** **Specification Changed**

CS-STUAGG-005

On the `manualAllocation` function, there is the following comment:

```
@dev Manual update the allocations.
Calculate the newAPR, curAPR and if newAPR < curAPR then it would be failed.
```

However, there is no check in the code that makes the call fail if `newAPR < curAPR`. The admin could input any manual allocation, no matter the resulting APR.

Spec changed:

The comment has been removed.

6.7 Reentrancy Guards Applied Inconsistently

Design | Low | Version 1 | Code Corrected

CS-STUAGG-012

The `SiloGateway` has a reentrancy guard on its `borrowAsset` function, but `DebtManager` does not have a reentrancy guard on `requestLiquidity()`.

Note that there can be multiple `SiloGateway` for each `DebtManager`, so it could technically be possible to reenter `requestLiquidity()`.

Code corrected:

A reentrancy guard has been added to `requestLiquidity()` in `DebtManager`.

It has also been clarified that the system is not intended to be used with reentrant tokens such as ERC-777.

6.8 Gas Optimizations

Informational | Version 1 | Code Corrected

CS-STUAGG-006

1. The functions `DebtManager.removeLender` and `SiloGateway.borrowAsset` could be `payable` to save gas.
2. In the function `DebtManager.requestLiquidity` the condition for `continue` could be moved at the beginning of the `for` loop. This avoids unnecessarily loading from storage.
3. The function `DebtManager.requestLiquidity` could avoid the big `for` loop if the `totalIdle` amount is enough to cover `requiredAmount`.
4. The function `DebtManager.sortLendersWithAPR` makes a lot of calls to the APR oracle. These calls could be cached to avoid querying the same value multiple times.

Code corrected:

1. No change. Sturdy states:

   ```
   Since anyone can call DebtManager.removeLender and SiloGateway.borrowAsset,
   they should not be payable in order to prevent the user from potentially
   sending ether and losing funds.
   ```

2. The condition has been moved at the beginning of the loop.
3. If the total idle assets are enough to cover the required amount, the loop is completely skipped.
4. The APR is queried once for every lender in an independent loop before sorting.

6.9 Misleading Error Names

Informational | Version 1 | Code Corrected

CS-STUAGG-007

- The error returned by `DebtManager._manualAllocation()` when the new debt exceeds the lender's max debt should be `AG_HIGHER_DEBT`

- The error returned by `DebtManager.requestLiquidity()` when the `requiredAmount` is not equal to zero should be `AG_INSUFFICIENT_ASSET`
- The errors returned when the lender's address is not active should be `AG_INVALID_LENDER`

Code corrected:

The error names have been changed to more accurately reflect the error.

6.10 Missing Input Sanitization

Informational | Version 1 | Code Corrected

CS-STUAGG-008

The `SiloGateway` constructor and `setUtilizationLimit` function do not sanitize `utilizationLimit_`. It could accidentally be set to more than `100%`.

Code corrected:

The logic related to the utilization limits has been moved to the `DebtManager`, where input sanitization is properly done. The limits are enforced to be strictly smaller than `UTIL_PREC`.

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 manualAllocation Can Ignore Unrealized Losses

Informational | Version 1 | Acknowledged

CS-STUAGG-011

In `manualAllocation()`, there is the following check:

```
if (lenderData.current_debt == position.debt) continue;
```

This is intended to skip a lender if there should be no change to its debt.

However, the `current_debt` value may be outdated, as there is no call to `assess_share_of_unrealised_losses()`.

As a result, the debt of the position when including unrealized losses may be different than expected.

Acknowledged:

Sturdy responded:

```
Unrealised losses will be very rare;
in the event they do occur, process_report() will be called before
manualAllocation() to prevent a discrepancy.
```

7.2 zkAllocation Could Contain Duplicates

Informational | Version 1 | Acknowledged

CS-STUAGG-009

In `_manualAllocation()`, there is no check that unique lenders are included in the input. `zkAllocation()` has a length check on the new positions array, but there may be duplicate entries.

The `admin` may call `_manualAllocation()` with any values.

Additionally, the existence check for positions happens after `continue`. As a result, a non-existent position could be included in the array if the new `position.debt` is `0`.

Acknowledged:

Sturdy responded:

```
To reduce gas costs, we don't check duplicate entries.
This is a permissioned function, so the admin and zkVerifier will avoid duplicated lenders.
```

Additionally, the existence check for positions has been moved to before the `continue` in the `_manualAllocation` loop.