PUBLIC

Code Assessment

of the Vault Revenue Splitter Smart Contract

August 18, 2025

Produced for

by

Contents

1 Executive Summary 3
2 Assessment Overview 5
3 Limitations and use of report 9
4 Terminology 10
5 Open Findings 11
6 Resolved Findings 13
7 Informational 20
8 Notes 21

1 Executive Summary

Dear Aave Team,

Thank you for trusting us to help you with this security audit. Our executive summary provides an overview of subjects covered in our audit of the Vault Revenue Splitter according to the defined Scope to support you in forming an opinion on the security risks.

Aave implements `ATokenVaultRevenueSplitterOwner` a new contract that will act as the owner of an `ATokenVault` and split the fees and rewards generated by the `ATokenVault` among fixed recipients according to predefined shares.

The most critical subjects covered in our audit are the correct distribution of funds among the recipients, the security of the funds held inside the contract, and the property that the recipients will jointly receive all the funds they are entitled to.

Furthermore, we generally reviewed the code for functional correctness, access control, arithmetic precision, and gas efficiency.

We originally found multiple issues. Very minor risks regarding blacklistable tokens have been accepted. All remaining issues have been addressed in subsequent versions. Hence, no security concerns remain from our side.

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	2
• Code Corrected	2
Low-Severity Findings	7
• Code Corrected	3
• Specification Changed	2
• 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 file:

`src/ATokenVaultRevenueSplitterOwner.sol`

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

V	Date	Commit Hash	Note
1	21 July 2025	efd4dff613ca3772cc4a34ad80fc683d88fdf3df	Initial Version
2	01 August 2025	657db341e646989278ba759a37cf1b9928c0d123	Fixes
3	15 August 2025	080865b91e0d34b9041bd21cb5afe4db2f65f845	Final Version

For the solidity smart contracts, the compiler version `0.8.10` was chosen.

2.1.1 Excluded from scope

Many contracts that the `ATokenVaultRevenueSplitterOwner` directly or indirectly interacts with were excluded from the scope of this assessment. In particular, the following contracts were not assessed:

- `ATokenVault`
- `AToken`
- `Pool`
- `RewardsController`

2.2 Assumptions

Due to the restricted scope of this assessment, we assume that out-of-scope contracts are secure. Furthermore, we cannot foresee the future use cases of the reviewed contract. Hence, we document that we assume the following properties hold:

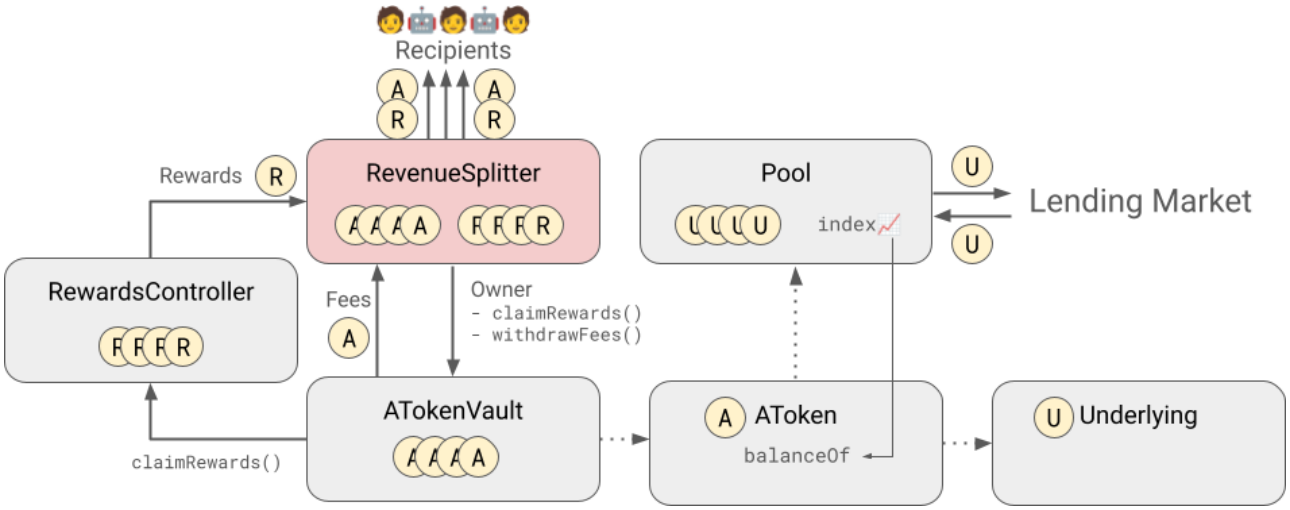
- Underlying Token
  - ERC-20 compliant
  - Cannot have hooks on `transfer`, e.g., cannot be ERC-777
  - Non-malicious
- Rewards Controller
  - Correctly computes rewards
  - Reward Emission is independent of the frequency of calls it receives
  - Only emits non-malicious reward tokens
- AToken

- `balanceOf(a)` for an address a cannot decrease unless a transfers away tokens using `transfer/transferFrom/transferOnLiquidation`
- has no blacklisting functionality
- has no special functionality besides rebasing
- all balances fit into 128 bits

- Pool
  - `liquidityIndex` (we also refer to it as `index`) is monotonically increasing
- ATokenVault
  - ERC-4626 compliant
  - No inflation attacks

Lastly, many of these contracts are upgradable and could change their behavior in the future. In case of such changes, our review might no longer apply.

2.3 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.

Aave Token Vaults (`ATokenVault`) provide an ERC-4626 compliant vault, whose shares represent a user's deposit in the Aave protocol in one particular underlying. The `ATokenVault` allows users to deposit Aave tokens (ATokens) and receive shares in return. The shares will increase in value over time, as the ATokens accrue interest from the Aave protocol. A part of this value increase that the vault observes can be withdrawn as fees by the vault owner. The new contract `ATokenVaultRevenueSplitterOwner` can collect the fees from the vault and distribute them to different recipients, according to a predefined split.

The `ATokenVaultRevenueSplitterOwner` will become the new owner of the vault, and has the following functions:

- **constructor**: The constructor initializes the contract with:
  - the address of the vault
  - the address of the owner

- the addresses of the recipients
- the percentages for each recipient, specified in basis points (1 basis point = 0.01%). It is checked that the sum of all percentages is `10_000` (100%).

Crucially, the recipients and their percentages are **immutable**, meaning they cannot be changed after the contract is deployed.

- **withdrawFees**: This function allows anyone to withdraw the fees collected by the vault to the `ATokenVaultRevenueSplitterOwner` contract.
- **claimRewards**: This function allows anyone to claim extra rewards from the Aave protocol that have been collected by the vault. The rewards are then transferred to the `ATokenVaultRevenueSplitterOwner` contract.
- **splitRevenue**: This function allows anyone to distribute the collected fees and rewards to different recipients according to a predefined split.
- **setFee**: This function allows the owner to set the fee percentage that will be collected from the vault. The fee is a percentage of the value increase of the ATokens in the vault. The fee can be up to 100%.
- **emergencyRescue**: This function allows the owner to rescue any tokens that are stuck in the vault contract. These tokens can be transferred anywhere.
- **transferVaultOwnership**: This function allows the owner to transfer the ownership of the vault to another address. This is useful if the owner wants to change the recipients of the fees and rewards.

2.3.1 System Invariant

An important invariant of the system is that the configured recipients of the fees and rewards will always receive their share according to the predefined split for the period in which the `ATokenVaultRevenueSplitterOwner` has been the owner of the vault. In particular, the `owner` of the `ATokenVaultRevenueSplitterOwner` contract must not be able to act so that already accrued and claimable fees and rewards are not distributed to the recipients.

2.3.2 Token Dust

Very small amounts of distributed tokens ("dust") may remain in the `ATokenVaultRevenueSplitterOwner` contract after the `splitRevenue` function is called. This is because they cannot be split evenly among the recipients. Most of the time, this dust will be negligible. Furthermore, it will likely be distributed to the recipients in the next call to `splitRevenue`. Hence, there should be no dust accumulation over time.

2.3.3 Comments

- The `ATokenVaultRevenueSplitterOwner` does not enforce unique recipients, so it is possible to have multiple recipients with the same address.
- Native Tokens are not supported by the `ATokenVaultRevenueSplitterOwner` contract. The `ATokenVaultRevenueSplitterOwner` cannot receive native tokens, as it has a reverting `receive` function. This is fine, as currently, inside the Aave protocol, there are no native tokens.
- As the `ATokenVaultRevenueSplitterOwner` is the owner of the vault, all the vault's `onlyOwner` functions can only be called by the `ATokenVaultRevenueSplitterOwner` contract. The `ATokenVaultRevenueSplitterOwner` contract does not implement the `renounceOwnership` function. However, there is also no clear need for it.

2.3.4 Changes in Version 2

In Version 2, some clarifications regarding the use of the `ATokenVaultRevenueSplitterOwner` were made. Furthermore, internal changes were made to address the findings of Version 1. Those changes mostly aim at ensuring a fair split between the recipients and therefore reduce the effect of rounding errors.

2.3.5 Changes in Version 3

In Version 3, duplicate recipients are not permitted, as they lead to incomplete revenue distribution in the `splitRevenue` function of Version 2. A corresponding check has been added to the `_setRecipients` function called inside the constructor.

2.4 Trust Model

The following roles are relevant to this scope:

- Owner of the `ATokenVaultRevenueSplitterOwner` contract
  - Trust Level: Fully trusted
  - At any point, they can transfer the ownership of the vault to another address, e.g., themselves.
- Recipients of the `ATokenVaultRevenueSplitterOwner` contract
  - Trust Level: Untrusted
  - They need to ensure that `withdrawFees` is called at a reasonable frequency, as they otherwise lose out on potential revenue.
- Vault Share Holders
  - Trust Level: Untrusted

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 Open 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

- Blacklisted Recipients Block Reward Distribution for All Risk Accepted
- Blacklisted Splitter Contract Can Block Reward Distribution and Ownership Transfer Risk Accepted

5.1 Blacklisted Recipients Block Reward Distribution for All

Security Low Version 1 Risk Accepted

CS-AAVEVRS-003

In case one of the reward tokens is a blacklistable token, e.g., USDC, one of the recipients of the `ATokenVaultRevenueSplitterOwner` can be blacklisted. Then, the transfer of the reward token to that recipient will revert. As the function `splitRevenue` is looping over all recipients, this will cause the whole transaction with all transfers to revert:

```
for (uint256 j = 0; j < recipients.length; j++) {
    uint256 amountForRecipient = amountToSplit * recipients[j].shareInBps / TOTAL_SHARE_IN_BPS;
    if (amountForRecipient > 0) {
        IERC20(assets[i]).safeTransfer(recipients[j].addr, amountForRecipient);
```

Hence, none of the recipients (including the non-blacklisted ones) will receive any further rewards of this token. The rewards are stuck in the `ATokenVaultRevenueSplitterOwner` contract. If these rewards have not been distributed in a while, this could lead to a significant amount of funds being stuck.

Risk accepted:

Aave accepts the risk, stating:

```
[The contract is] unlikely to have a relevant revenue in a token with blacklist
(given that revenue comes from fees and rewards, fees are in aToken which does
not have blacklist, and rewards are unlikely to be in a token with blacklist).
In addition to that, it requires to get some recipient blacklisted, increasing
the unlikelihood even more.
```

5.2 Blacklisted Splitter Contract Can Block Reward Distribution and Ownership Transfer

Security | Low | Version 1 | Risk Accepted

CS-AAVEVRS-004

In case one of the reward tokens has a blacklist, e.g., USDC, then the `ATokenVaultRevenueSplitterOwner` could theoretically be blacklisted by the token contract. This would have the following consequences:

1. It would prevent the `ATokenVaultRevenueSplitterOwner` from receiving rewards from the blacklisted token, as the function `claimRewards` would revert.
2. It would prevent the `ATokenVaultRevenueSplitterOwner` from distributing already collected rewards of the blacklisted tokens.
3. It would prevent the execution of the function `transferVaultOwnership` on the `ATokenVaultRevenueSplitterOwner` contract, which is used to transfer ownership of the vault to a new owner. This is because this function also calls `claimRewards` internally, which would revert if the contract is blacklisted. Hence, the ownership could never be transferred again, and hence the recipients could never be changed again.

Risk accepted:

Aave accepts the risk stating:

```
[The contract is] unlikely to have a relevant revenue in a token with blacklist
(given that revenue comes from fees and rewards, fees are in aToken which does
not have blacklist, and rewards are unlikely to be in a token with blacklist).
In addition to that, it requires to get the splitter contract blacklisted,
increasing the unlikelihood even more.
```

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Open 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	2

- AToken Distribution Can Fail Code Corrected
- Incorrect Split of Revenue Code Corrected

Low-Severity Findings	5

- Duplicates in the Recipients Array Break _splitRevenue Code Corrected
- Conditional Event Emission Specification Changed
- Fee Accrual and Fee Withdrawal Code Corrected
- Input Validation Can Be Improved Code Corrected
- Reward Tokens With Hooks Allow Recipients to Block Split Specification Changed

6.1 AToken Distribution Can Fail

Correctness Medium Version 1 Code Corrected

CS-AAVEVRS-001

As the fees are in the form of `AToken`, the function `splitRevenue` must ensure that the distribution of these tokens is done correctly, given that it will be the primary source of revenue for the recipients.

The function `splitRevenue` contains the following code to distribute `AToken`:

```
uint256 amountToSplit = IERC20(assets[i]).balanceOf(address(this));
for (uint256 j = 0; j < recipients.length; j++) {
    uint256 amountForRecipient = amountToSplit * recipients[j].shareInBps / TOTAL_SHARE_IN_BPS;
    if (amountForRecipient > 0) {
        IERC20(assets[i]).safeTransfer(recipients[j].addr, amountForRecipient);
```

As we can see, the `amountToSplit` is split up according to the shares of each recipient. The `amountForRecipient` is then sent to each recipient.

However, as aToken are rebasing their behavior is special. In particular, the `balanceOf` function does not change as one might expect, for example:

1. AToken.balanceOf(alice) = 1,000
2. Alice calls AToken.transfer(bob, 100)
3. What is AToken.balanceOf(alice)?

One might expect that the balance of Alice is now 900, but this **is not guaranteed to be the case** due to rounding in the rebase mechanism.

Hence, **the last of multiple successive `transfer` calls in `splitRevenue` might fail due to insufficient balance**. Please consider the following example (small numbers for simplicity, most values taken from existing tests):

- There are two recipients with shares of 80% and 20%.
- The liquidity index is: 1015939556101939848435904568
- The AToken.balanceOf(`ATokenVaultRevenueSplitterOwner`) = 100
- `splitRevenue` is called:
    1. 80 AToken wei are sent to recipient 1 (80% of 100)
    2. AToken.balanceOf(`ATokenVaultRevenueSplitterOwner`) = 19
    3. There is an attempt to send 20 AToken wei to recipient 2 (20% of 100)
    4. This fails due to insufficient balance, as the balance is only 19 AToken wei.
    5. The whole transaction reverts, and no AToken is sent to any recipient.

Please note the following about this issue:

1. It can happen for two or more recipients.
2. It can happen randomly, without malicious intent.
3. The likelihood of a random occurrence seems to depend on the liquidity index, the number of recipients, and the share distribution. In our tests, it happened in roughly 2% of the attempts.
4. Once additional AToken are sent to the contract, e.g., from `withdrawFees`, the issue will likely resolve itself.
5. A malicious actor could exploit this issue by sending a small amount of AToken to the contract, which would then cause the split to fail, preventing the distribution of funds to the recipients.
6. This issue also applies to a contract outside of the scope of this review, the RevenueSplitter.

Code corrected:

In the new code version, the contract only tries to distribute `balanceOf(address(this)) - 1`. Hence, if the asset is an aToken it can no longer fail. This change increases the amount of dust that can remain inside the contract by 1 wei.

6.2 Incorrect Split of Revenue

Security | Medium | Version 1 | Code Corrected

CS-AAVEVRS-002

Anyone can trigger an incorrect split of revenue by calling the function `splitRevenue` **very often**. This will create significantly more rounding errors during the distribution of revenue.

Imagine the following scenario:

- Underlying token is WBTC:
    - 8 decimals
    - High value, e.g., 120,000 USD at the time of writing
- There are 4 recipients with the following shares:
    - Recipient A: 85%
    - Recipient B-D: 5%

- The fees of the vault are 16 wei per block (~0.02 USD)

After 2,628,000 blocks (the number of blocks in a year on mainnet), we would expect the following distribution of revenue:

- Recipient A: 35,740,800 wei (~42,889 USD)
- Recipient B-D: 2,102,400 wei (~2,522 USD)

However, if the function `splitRevenue` is called every block, the distribution will be as follows:

- Recipient A: 42,047,997 wei (~50,457 USD)
- Recipient B-D: 0 wei (0 USD)

Hence, Recipient A receives ~100% of the revenue, while the other recipients receive nothing, even though Recipient A's share is significantly below 100%. Obviously, Recipient A has an incentive to call the function every block, as they would receive all the revenue. The cost of calling could be smaller than the additional revenue they would receive. This could especially apply to L2s where transaction costs are lower.

Generally, this can happen with any token that has a high value and few decimals, and when there is an uneven distribution among the recipients. Also, it does not require a call every block, but rather at the right frequency so that the best rounding errors appear.

Code corrected:

The issue has been addressed through the introduction of two new storage variables:

- `_previousAccumulatedBalance` tracks the amount already transferred to all recipients per asset
- `_amountAlreadyTransferred` tracks the amount already transferred per asset and recipient

This way, rounding errors cannot accumulate, as rounding errors in one call to `splitRevenue` will be corrected in subsequent calls.

6.3 Duplicates in the Recipients Array Break `_splitRevenue`

Correctness | Low | Version 2 | Code Corrected

CS-AAVEVRS-010

The `_setRecipients` function does not check for duplicates in the recipients array. Previously, this was not an issue because the `_splitRevenue` function distributed revenue for each entry in the recipients array, effectively resulting in the sum of shares for duplicate recipients. In Version 2, however, the function uses the `_alreadyAccumulatedBalance` mapping to track distributed balances. This change causes incorrect revenue splits when duplicates are present: the mapping is updated on the first occurrence and used to compute `amountForRecipient` on subsequent occurrences. The two potential results can be:

1. Locked funds, if the second share is equal to or greater than the first. This implies that a part of the funds can never be distributed.
2. A revert due to underflow, if the second share is smaller than the first. This implies that all of the shares cannot be distributed.

The following code shows how the amount for each recipient is calculated in Version 2:

```
uint256 amountForRecipient = accumulatedAssetBalance * recipients[j].shareInBps / TOTAL_SHARE_IN_BPS
    - _amountAlreadyTransferred[assets[i]][recipients[j].addr];
```

Code corrected:

In Version 3, the `_setRecipients` function, which is called inside the constructor, requires all recipients to be unique. Hence, the issue is resolved.

6.4 Conditional Event Emission

Design | Low | Version 1 | Specification Changed

CS-AAVEVRS-005

In the function `splitRevenue` there is the following code:

```
uint256 amountForRecipient = amountToSplit * recipients[j].shareInBps / TOTAL_SHARE_IN_BPS;
if (amountForRecipient > 0) {
    IERC20(assets[i]).safeTransfer(recipients[j].addr, amountForRecipient);
}
emit RevenueSplitTransferred(recipients[j].addr, assets[i], amountForRecipient);
```

The event `RevenueSplitTransferred` is emitted regardless of the value of `amountForRecipient`. This can lead to unnecessary event emissions when the amount is zero, which could be avoided by moving the event emission inside the conditional block. This would save gas costs and avoid clutter in the event logs.

Specification Change:

The specification was updated to clarify that an event should be emitted even when the amount is zero, ensuring consistent tracking by indexers.

6.5 Fee Accrual and Fee Withdrawal

Security | Low | Version 1 | Code Corrected

CS-AAVEVRS-006

Inside the `_accrueYield` function of the `ATokenVault` contract, the fees (which the `ATokenVaultRevenueSplitterOwner` can later withdraw) are accrued:

```
uint256 newVaultBalance = ATOKEN.balanceOf(address(this));

uint256 newYield = newVaultBalance - _s.lastVaultBalance;
uint256 newFeesEarned = newYield.mulDiv(_s.fee, SCALE, MathUpgradeable.Rounding.Down);

_s.accumulatedFees += uint128(newFeesEarned);
_s.lastVaultBalance = uint128(newVaultBalance);
```

This fee accrual can behave differently than expected in multiple ways, which we describe below:

6.5.1 Yield generated by Fees

Once fees have been withdrawn, they reside in the `ATokenVaultRevenueSplitterOwner` contract. As the fees are `AToken` they will then generate yield in the same way as the `AToken` in the vault. That yield from the fees goes 100% to the `ATokenVaultRevenueSplitterOwner` contract.

If those fees had not been withdrawn, they would still be in the vault and would generate yield as part of the vault's total balance. Only the `fee` percentage of that yield would go to the `ATokenVaultRevenueSplitterOwner` contract.

Hence, the frequency of calls to `withdrawFees` is important. If the fees are not withdrawn frequently, they will continue to generate yield in the vault, which reduces the revenue that is received by the recipients.

6.5.2 Rounding Error in Fee Calculation

During each fee calculation, the fees are calculated using the `mulDiv` function, rounding down. Hence, up to 1 wei of fees can be lost per calculation. If that 1 wei has a lot of value, e.g., WBTC, then regular vault share owners would have an incentive to trigger that loss repeatedly. Regular vault share owners can execute the `_accrueYield` function by depositing, withdrawing, or calling `withdrawFees` on the `ATokenVaultRevenueSplitterOwner`.

Hence, this could be:

- happening accidentally, e.g., because vault share owners are depositing or withdrawing frequently, or
- performed as a grievance attack against the recipients of the fees, or
- performed to increase the vault share value if the transaction costs are low enough.

6.5.3 Fees are rounded down to Zero

As a special case of the previous point, if the fee is very small, it can be rounded down to 0. However, the `_s.lastVaultBalance` is updated to the current vault balance, which means that fewer fees will be accrued in the next call to `_accrueYield`. Theoretically, this can result in 0 fees being accrued for a long time, even if the vault is generating yield.

The `ATokenVaultRevenueSplitterOwner` contract might even trigger this behavior accidentally, because the `withdrawFees` in `ATokenVaultRevenueSplitterOwner` does the following:

```
function _withdrawFees() internal {
    uint256 feesToWithdraw = VAULT.getClaimableFees();
    VAULT.withdrawFees(address(this), feesToWithdraw);
}
```

Instead, the `ATokenVaultRevenueSplitterOwner` could skip calling `withdrawFees` if the `feesToWithdraw` is 0, which would prevent the update of the `_s.lastVaultBalance` and hence the future fee reduction.

Code corrected:

In the new code version, the behavior of `_withdrawFees` has been changed:

```
function _withdrawFees() internal {
    uint256 feesToWithdraw = VAULT.getClaimableFees();
    if (feesToWithdraw > 0) {
        VAULT.withdrawFees(address(this), feesToWithdraw);
```

```
    }
}
```

Hence, Fees are rounded down to Zero is resolved.

Regarding Yield generated by Fees, it has been clarified that the responsibility lies with the stakeholders to call `withdrawFees` often enough.

Regarding Rounding Error in Fee Calculation, Aave states that it is a very unlikely scenario and accepts the remaining risk. ChainSecurity agrees that it is a very unlikely scenario.

6.6 Input Validation Can Be Improved

Security | Low | Version 1 | Code Corrected

CS-AAVEVRS-007

In the `_setRecipient` function, the sum of recipient shares is validated to ensure it equals 100%. However, there is no validation for the provided recipient addresses.

If a recipient is set with an address equal to `0x0`, and one of the tokens processed in `splitRevenue` has a restriction against transfers to the `0x0` address, the entire transaction will revert, preventing the distribution of that specific token.

Furthermore, if the distributed token does not have such a restriction, the mistake could go unnoticed for a while, and funds would be lost by being transferred to the `0x0` address.

Code Corrected:

The `_setRecipient` function now requires recipient addresses to be non-zero.

6.7 Reward Tokens With Hooks Allow Recipients to Block Split

Security | Low | Version 1 | Specification Changed

CS-AAVEVRS-008

If a reward token contains a hook, e.g., ERC-777, one of the recipients can block the execution of `splitRevenue` **for that particular token**. This is because the function will loop over all recipients:

```
for (uint256 j = 0; j < recipients.length; j++) {
    uint256 amountForRecipient = amountToSplit * recipients[j].shareInBps / TOTAL_SHARE_IN_BPS;
    if (amountForRecipient > 0) {
        IERC20(assets[i]).safeTransfer(recipients[j].addr, amountForRecipient);
```

Hence, a misbehaving recipient can block the execution of the function by blocking the transfer of the token to itself. This is a security issue because it allows a recipient to prevent the distribution of funds to all other recipients. Theoretically, the misbehaving recipient could use this as a chance to blackmail the other recipients. The amount of funds that can be blocked depends on how many rewards have accrued since the last distribution.

Specification Change:

Aave changed the specification to not allow tokens with hooks, such as ERC-777. Hence, these are not supported by the `ATokenVaultRevenueSplitterOwner` contract.

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 Potential Gas and Code Size Optimization

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

CS-AAVEVRS-009

The following non-exhaustive list highlights where gas efficiency and code size can potentially be improved:

- **Internal Constant Declaration**: The `TOTAL_SHARE_IN_BPS` constant could be declared as internal rather than public. This would eliminate the need for generating a getter method, resulting in reduced code size.
- **Custom Errors Definition**: Custom errors could be implemented instead of using string-based errors. This would optimize gas usage and reduce the deployment size of the contract.

Acknowledged:

Aave states:

```
Both the public constant and the string errors are a standard on this repository.
We are aware of the small inefficiencies they introduce compared to the suggested
change.
```

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 Dust Balance Can Be Higher Than Expected

Note Version 1

When distributing the previously accumulated funds using the `splitRevenue` function, the `ATokenVaultRevenueSplitterOwner` contract may leave a small residual balance ("dust") due to floor-rounding when calculating each recipient's share of the total balance.

The code contains the following comment about this:

```
// Due to floor-rounding in integer division, the sum of the amounts transferred may be less than the
// total amount to split. This can leave up to `N - 1` units of each asset undistributed in this
// contract's balance, where `N` is the number of recipients.
```

As described in the comment, the maximum unallocated amount should be `N - 1` wei, where `N` is the number of recipients. However, due to rounding in the `balanceOf` function of the aToken, the actual dust could be up to `N` wei.

In Version 3, the final version of this audit, the dust balances can be up to `2 * N` where `N` is the number of recipients.

8.2 Limit on Number of Recipients

Note Version 1

The `_recipients` array's size has a direct impact on the gas cost of distributing tokens, as the larger the number of recipients, the higher the gas fees to call `splitRevenue`.

This is because the `splitRevenue` function iterates over the entire list of recipients, and the computational overhead increases with the size of the array. In the extreme case, the cost grows beyond the block's gas limit, which is currently limited to 45 million gas units, and the `splitRevenue` function becomes prohibitively expensive to execute, potentially locking funds in the contract.

Hence, a very conservative limit on the number of recipients could be introduced to avoid such a case of locked funds.

8.3 Theoretical Read-only Reentrancy

Note Version 1

A Read-only Reentrancy could theoretically occur in the `ATokenVaultRevenueSplitter` if a token with a hook `T`, e.g. ERC-777, is being distributed using `splitRevenue`. During the execution of the hook for one recipient, a part of the revenue for this token would have already been distributed, while another part is still held in the contract.

As a result, another contract observing the `ATokenVaultRevenueSplitter` could think that `T.balanceOf(ATokenVaultRevenueSplitter)` will be distributed among all recipients, while in

reality it will only be distributed among the recipients that have not yet been processed by `splitRevenue`. We are unaware of any current contract where this would be an issue, but we cannot foresee future contracts that could be affected by this.

During the audit the specification was clarified to not allow tokens with hooks, such as ERC-777. Hence, this is no longer a concern.