PUBLIC

Code Assessment

of the CowSwap Adapters Smart Contracts

Nov 25, 2025

Produced for

by

Contents

1 Executive Summary

Dear Aave team,

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

AAVE implements convenience contracts for Aave users to perform the following actions more efficiently:

- Swap the collateral of a loan to a different asset
- Swap the debt of a loan to a different asset
- Repay a loan with posted collateral

The code is well structured and written. We carefully assessed if the protocol interactions are correct, amounts are passed correctly and there is any way to interact maliciously with the instance by e.g., using permits/approvals. We could not identify any major issues. However, because the solvers have freedom in their executions, there is no guarantee that the expected appData instructions are followed, so the interactions might need to be monitored carefully. Besides, we could not find an attack scenario for the intra-hook but this hook allows alternative execution paths for users that are not intended in normal operation.

In summary, we find that the codebase provides a high level of security. Yet, 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	0

2 Assessment Overview

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

2.1 Scope

The assessment was performed on the following source code files inside the CowSwap Adapters repository based on the documentation files.

- src/contracts/AaveV3AdapterFactory.sol
- src/contracts/AaveV3BaseAdapter.sol
- src/contracts/BalancerV3AdapterFactory.sol
- src/contracts/BaseAdapterFactory.sol
- src/contracts/CollateralSwapAaveV3Adapter.sol
- src/contracts/DebtSwapAaveV3Adapter.sol
- src/contracts/RepayWithCollateralAaveV3Adapter.sol
- src/libraries/DataTypes.sol
- src/interfaces/

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

V	Date	Commit Hash	Note
1	27 Oct 2025	85e0e648d64bc5d97afe4fc73d1ef4d24faddc4d	Initial Version
2	24 Nov 2025	114ec7b4a5f1142be53a4b37753a95fa089eebb2	Final Version

For the Solidity smart contracts, the compiler version `0.8.28` was chosen.

2.1.1 Excluded from scope

All third-party contracts like libraries or contracts called from the system are excluded from 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.

AAVE offers convenience contracts for users to perform the following actions more efficiently:

- Swap the collateral of a loan to a different asset
- Swap the debt of a loan to a different asset
- Repay a loan with posted collateral

Each action would require temporary capital and multiple calls to perform the actions without helper contracts. The adapter doesn't require additional capital from the user. The contracts will take a flash loan, swap and perform the action, and finally use the remaining funds to repay the flash loan.

For each scenario the high-level idea is:

1. Collateral swap:
    1. Get a flash loan in the old collateral asset (sellToken)
    2. Swap the old collateral asset (sellToken) for the new collateral asset (buyToken)
    3. Supply the new collateral asset as collateral for the loan (buyToken)
    4. Withdraw the old collateral asset from the loan (sellToken)
    5. Repay the flash loan in the old collateral asset (sellToken)
    6. Sweep any dust
2. Debt swap:
    1. Get a flash loan in the new debt asset (sellToken)
    2. Swap the new debt asset for the old debt asset (buyToken)
    3. Repay the old loan with the old debt asset (buyToken)
    4. Open a new loan with the new debt asset (sellToken)
    5. Repay the flash loan in the new debt asset (sellToken)
    6. Sweep any dust
3. Repay with collateral:
    1. Get a flash loan in the collateral asset (sellToken)
    2. Swap the collateral asset for the debt asset (buyToken)
    3. Repay (part of) the loan with the debt asset (buyToken)
    4. Withdraw (part of) the collateral (sellToken)
    5. Repay the flash loan in the new debt asset (sellToken)
    6. Sweep any dust

To initiate an action the user must place an order in the CowSwap protocol off-chain order-book. The order must include an appData hash and the corresponding appData JSON must be available for the solver. The appData must define the flash-loan parameters and the pre- and post-hook actions. The signature provided by the user in combination with the order will be an EIP-712 signature over the `AdapterOrderSig` object.

The solver who will execute the order will call into the `FlashLoanRouter` contract with `flashLoanAndSettle`. This will trigger the interaction with the `AdapterFactory.flashLoanAndCallBack`. The factory (depending on whether it takes a Balancer or Aave flash loan) will interact with either the Balancer vault or Aave pool to initiate the flash loan in `_triggerFlashLoan`. In case of the AAVE AdapterFactory, this will call the `flashLoan` function of the AAVE Pool. In case of the Balancer AdapterFactory, this will call the `unlock` function of the Balancer Vault.

AAVE and Balancer will use the provided callback data to call back into the `msg.sender` (Balancer) or the provided receiver (AAVE) address. In both cases this will be the `AdapterFactory` contract and ultimately end up in `_executeFlashLoan` to call `borrowerCallBack` on the `FlashLoanRouter` contract. After the flash loan is taken, the `FlashLoanRouter` will start to settle the order by calling `settle` on the settlement contract. The `settle` function calls the pre-hook interaction to deploy the

AdapterInstance contract and transfer the flash loaned assets to the instance. Using the GPv2Interaction.Data array provided by the user, this will trigger deployAndTransferFlashLoan from the BaseAdapterFactory contract, and deploy the AdapterInstance contract (by using cloneDeterministic), initialize the contract via setParameters and transfer the flash loaned assets to it. After deployment the instance will be initialized with the parameters from the order. The owner of the adapter instance is the user who signed the order. They have the ability to sweep the adapter instance with rescueTokens.

The CowSwap protocol will then verify and compute the trade executions put together by the solver in computeTradeExecutions. The verification includes verifying the user's EIP-712 signature over the AdapterOrderSig by calling back into the instance's isValidSignature function. The instance uses the order to rebuild the AdapterOrderSig digest (via the factory), verifies userSig, and also re-hashes order to ensure orderHash matches. This ensures that AdapterOrderSig (user signature) is under the Factory's EIP-712 domain (name/version/chainId/Factory-address) and includes the instance address. The returned inTransfers from computeTradeExecutions will then be distributed to the accounts affected by the trades.

Users have the option to execute another intra-hook but this should be empty in normal operation. The execution then distributes the outTransfers to the accounts involved in the trades, including the AdapterInstance contract. With the funds available, the AdapterInstance will perform one of the three actions: collateral swap (collateralSwapWithFlashLoan), debt swap (debtSwapWithFlashLoan) or repay with collateral (repayWithCollateralWithFlashLoan). Each function will perform the steps outlined in the detailed walkthrough above and call _repayFlashLoan to repay the flash loan. After repaying it will sweep any dust or leftover assets from the contract.

In _repayFlashLoan it will approve the factory to pull the funds to repay the flash loan and call notifyRepayFlashLoan on the factory. Via the internal function _repayFlashLoan it will transfer the funds from the contract to the factory and:

1. In case of AAVE, it approves the factory to pull the funds from the instance. The factory will transfer the funds to msg.sender which is the AAVE Pool from the very beginning which will now continue the flashLoan function and pull the funds back to the pool. However, in case the borrower cannot return the funds, the pool will check if there is enough collateral and eventually open a debt position.
2. In case of Balancer, it transfers the funds to the Balancer Vault and calls settle on the Balancer Vault.

In the end the position should be changed according to the user's intent and the flash loan repaid.

The final version of the code added the feature to cancel an action by disallowing the factory to deploy a new instance.

2.3 Trust Model

The CowSwap solvers are untrusted when it comes to hook execution. However, execution of hooks is part of the CowSwap social consensus rules and intentionally omitting hook execution might result in the slashing of a solver's bond through a DAO vote.

The owner of the AdapterFactory contracts can rescue tokens from the contract and set the adapter implementation from which instances are cloned during execution.

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 our findings. The findings are split into these different categories:

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

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

6 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.

6.1 Missing Interface Definitions

Informational **Version 1**

CS-AAVECow-003

Some of the implemented functions are not declared in the interface. This might be intentional but it is not clear. Therefore, we highlight the following functions:

1. `IAaveBaseAdapter` does not declare the `isValidSignature` function
2. `IAaveBaseAdapter` does not declare the `rescueTokens` function
3. `IAaveBaseAdapter` does not declare the `owner` getter
4. `IBaseAdapterFactory` does not declare the `rescueTokens` function
5. `IBaseAdapterFactory` does not declare the `isAdapterImplementation` mapping getter
6. `IBaseAdapterFactory` does not declare the `openFlashLoans` mapping getter

6.2 Missing `supplyAmount` Upper Bound Check

Informational **Version 1**

CS-AAVECow-001

In the `_supply` function of the `AaveV3BaseAdapter` contract, there is no check that the `supplyAmount` is at most the `currentBalance`. This could lead to a situation where the Aave `supply` function reverts when trying to pull more funds from the adapter than the actual balance of the adapter instance.

In the other functions in the `AaveV3BaseAdapter` contract, there are `require` checks for such cases that will cause a revert with `InvalidAmount()` or the value is capped at `currentBalance`.

6.3 Salt Collisions

Informational **Version 1**

CS-AAVECow-002

The salt to deploy a new adapter instance includes the hook data. Consequently, the same user can only deploy with the same parameters once. The function `deployAndTransferFlashLoan` is unpermissioned. This opens up a trolling DoS vector where a third party can front-run the order by pre-deploying an instance at the deterministic address.

The cost would be quite high as the attacker needs to fund the factory with the flash loan amount. The implications would be low as the user might just change the order parameters slightly and try again. The validTo field might easily be changed and would be enough to deploy a new instance. Additionally, as the legitimate user must be the owner of the instance, the attacker would lose their funds to the legitimate user because they could sweep the adapter instance with `rescueTokens`.

In case the attacker would not pre-fund the factory with the loan amount, they must execute a flow to return the flash-loan funds. The only way to perform this would be to execute the intended functions on behalf of the user as requested.

7 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.

7.1 Uncontrollable Execution Paths

Note Version 1

There is no way to guarantee that solvers will actually execute the appData instructions as defined. They could provide the correct hash but not adhere to the instructions. However, as defined in the system overview, the solvers are incentivized to act in a non-malicious way according to CowSwap's social-consensus rules (otherwise their bond might get slashed through a DAO vote).

The interaction hooks could include additional or different calldata than needed to perform the intended actions. But we could not find a way to exploit this.

But we still advise to carefully monitor the interactions and the appData instructions to ensure that the intended actions are performed.