Report

v. 3.0

Customer

Aave DAO

Smart Contract Audit

Aave v 3.5

code developed by BGD Labs

18th July 2025

Report prepared by

ABDK Consulting

Contents

1 Changelog

#	Date	Author	Description
0.1	16.07.25	A. Zveryanskaya	Initial Draft
0.2	17.07.25	A. Zveryanskaya	Minor revision
1.0	17.07.25	A. Zveryanskaya	Release
1.1	18.07.25	A. Zveryanskaya	Added comment to CVF-3. CVF-6, 7, 8 downgraded
2.0	18.07.25	A. Zveryanskaya	Release
2.1	18.07.25	A. Zveryanskaya	Title/description adjustments
3.0	18.07.25	A. Zveryanskaya	Release

2 Introduction

The following document provides the result of the audit performed by ABDK Consulting (Mikhail Vladimirov and Dmitry Khovratovich) at the customer request. The audit goal is a general review of the smart contracts structure, critical/major bugs detection and issuing the general recommendations.

Aave v3 is a smart contracts protocol, for users to supply and borrow liquidity on Ethereum, and other networks

Bored Ghosts is a Web3 development initiative contributing to Aave.

3 Project scope

We were asked to review:

- Original Code
- Code with Fixes

Files:

instances/		
ATokenInstance.sol	PoolInstance.sol	VariableDebt TokenInstance.sol
VariableDebtToken MainnetInstanceGHO.sol		
interfaces/		
IAToken.sol	ICreditDelegation Token.sol	IPool.sol
IVariableDebtToken.sol		
protocol/libraries/helpers/		
TokenMath.sol		
protocol/libraries/logic/		
FlashLoanLogic.sol	BorrowLogic.sol	GenericLogic.sol
LiquidationLogic.sol	PoolLogic.sol	ReserveLogic.sol
SupplyLogic.sol	ValidationLogic.sol	
protocol/libraries/math/		
MathUtils.sol	PercentageMath.sol	WadRayMath.sol
protocol/pool/		
Pool.sol		
protocol/tokenization/base/		
IncentivizedERC20.sol	ScaledBalance TokenBase.sol	
protocol/tokenization/		
AToken.sol	ATokenWith Delegation.sol	ariableDebtToken.sol

4 Methodology

The methodology is not a strict formal procedure, but rather a selection of methods and tactics combined differently and tuned for each particular project, depending on the project structure and technologies used, as well as on client expectations from the audit.

- **General Code Assessment**. The code is reviewed for clarity, consistency, style, and for whether it follows best code practices applicable to the particular programming language used. We check indentation, naming convention, commented code blocks, code duplication, confusing names, confusing, irrelevant, or missing comments etc. At this phase we also understand overall code structure.
- **Entity Usage Analysis**. Usages of various entities defined in the code are analysed. This includes both: internal usages from other parts of the code as well as potential external usages. We check that entities are defined in proper places as well as their visibility scopes and access levels are relevant. At this phase, we understand overall system architecture and how different parts of the code are related to each other.
- **Access Control Analysis**. For those entities, that could be accessed externally, access control measures are analysed. We check that access control is relevant and done properly. At this phase, we understand user roles and permissions, as well as what assets the system ought to protect.
- **Code Logic Analysis**. The code logic of particular functions is analysed for correctness and efficiency. We check if code actually does what it is supposed to do, if that algorithms are optimal and correct, and if proper data types are used. We also make sure that external libraries used in the code are up to date and relevant to the tasks they solve in the code. At this phase we also understand data structures used and the purposes they are used for.

We classify issues by the following severity levels:

- **Critical issue** directly affects the smart contract functionality and may cause a significant loss.
- **Major issue** is either a solid performance problem or a sign of misuse: a slight code modification or environment change may lead to loss of funds or data. Sometimes it is an abuse of unclear code behaviour which should be double checked.
- **Moderate issue** is not an immediate problem, but rather suboptimal performance in edge cases, an obviously bad code practice, or a situation where the code is correct only in certain business flows.
- **Recommendations** contain code style, best practices and other suggestions.

5 Our findings

We provided the client with some recommendations.

Moderate	Info	Fixed
	0	2

Fixed 2 out of 2 issues

6 Moderate Issues

CVF-1 FIXED

- **Category** Flaw
- **Source** MathUtils.sol

Description This function silently returns zero in case “c” is zero.

Recommendation Revert in such a case.

```
108 +d := add(div(product, c), iszero(iszero(mod(product, c))))
```

CVF-3 FIXED

- **Category** Suboptimal
- **Source** AToken.sol

Description This seems overcomplicated. Let’s use the following notation:

- amount – the amount passed as an argument to the “transfer” or “transferFrom” call;
- allowance_spent – the value allowance is reduced by;
- amount_out: the value sender’s balance is reduced by;
- amount_in: the value recipient’s balance is increased by;
- amount_logged: the amount logged in the “Transfer” event.

For most of the tokens, all these values are the same, however, for scaled balances, these values could differ due to rounding errors. However, we can easily guarantee `amount=allowance_spent=amount_logged` Also, it would be reasonable to guarantee `amount >= amount_out`, as the sender wouldn’t be happy to see his balance decreased more, than expected. Regarding `amount_in` there are no strict constraints.

Recommendation Just deduct “amount” from allowance, and make sure sender’s balance decrease never exceeds “amount”.

Client Comment *Invalid, because the code change’s goal is to track allowance consumption from the owner’s perspective, which it accurately does.*

```
+// According to the ERC20 specification, the spent allowance should
   ↪  reflect the amount transferred.
+// Following the spec exactly is impossible though, as the
   ↪ allowance references the "scaled up" amount while the transfer
   ↪  operates with "scaled down" amount.
+// Because of this different handling of amounts, there are amounts
   ↪  that are impossible to accurately reflect on the balance.
+// As an example: transferFrom(..., 1) at a liquidity index of 2e27
   ↪ , can never transfer exactly `1` as the smallest unit that can
   ↪  be accounted for would be 2.
```

```
+// In addition to that the existing balance has an effect on the
   ↪ final "scaled up" balance after transfer.
+// As an example: transferFrom(..., 1) at a liquidity index of 1.1
   ↪ where the recipient has a "scaled up" balance of `9 * 1.1 =
   ↪ 9.9 = 9` before the transfer, will have a balance of `10 *
   ↪ 1.1. = 11` after the Transfer.
+// While this problem is not solvable without introducing breaking
   ↪ changes, on Aave v3.5 the situation is improved in the
   ↪ following way:
+// - The `correct` amount to be deducted is considered to be `
   ↪ scaledUpFloor(scaledDownCeil(input.amount))`. This replicates
   ↪ the behavior on transfer, followed by a balanceOf.
+// - In order to not introduce a breaking change for existing
   ↪ integrations, the deducted allowance is based on the available
   ↪  allowance as `Max(availableAllowance, (amount, correctAmount)
   ↪ )`
```

7 Recommendations

CVF-6 INFO

- **Category** Unclear behavior
- **Source** IAToken.sol

Description It is unclear, why both values are passed.

Recommendation Either remove one of the arguments, or clearly explain, why both are needed.

Client Comment *Both amounts are passed as they are used within the code for backwards compatibility.*

```
46  * @param amount The amount being burned
+* @param scaledAmount The scaled amount being burned
```

```
82 + uint256 amount,
+ uint256 scaledAmount,
```

CVF-7 INFO

- **Category** Unclear behavior
- **Source** IVariableDebtToken.sol

Description It is unclear why both amounts are needed.

Recommendation Either remove one of the arguments or clearly explain why both are needed.

Client Comment *Both amounts are passed as they are used within the code for backwards compatibility.*

```
19 +* @param amount The amount of debt being accounted for on the
      ↪ allowance
20 +* @param scaledAmount The scaledAmount of debt being minted
```

CVF-8 INFO

- **Category** Unclear behavior
- **Source** VariableDebtTokenMainnetInstanceGHO.sol

Description This check makes the “initializingPool” argument redundant.

Recommendation Remove the “initializingPool” argument or clearly explain, why this argument is actually needed.

Client Comment *Ack, would require an upgrade of the configurator which is not planned on this upgrade.*

```
31 +require(initializingPool == POOL, Errors.PoolAddressesDoNotMatch())
      ↪ ;
```

CVF-9 INFO

- **Category** Unclear behavior
- **Source** IncentivizedERC20.sol

Recommendation This function should return the actual amount spent.

Client Comment *As the return value would not be used we don’t see the point in that. If it’s ever needed it could be added without a breaking change as it’s internal.*

```
205 +function _spendAllowance(
```

CVF-10 INFO

- **Category** Unclear behavior
- **Source** AToken.sol

Description It is unclear why two amounts are needed.

Recommendation Remove on of the amounts or clearly explain why both are needed.

Client Comment *Both amounts are passed as they are used within the code for backwards compatibility.*

```
275  * @param amount The amount getting transferred
+* @param scaledAmount The scaled amount getting transferred
```

CVF-11 INFO

- **Category** Unclear behavior
- **Source** ATokenWithDelegation.sol

Description It is unclear why two amounts are needed.

Recommendation Remove one of the amounts or clearly explain why both are needed.

Client Comment *Both amounts are passed as they are used within the code for backwards compatibility.*

```
83  * @param amount The amount of tokens to transfer (non-scaled).
+* @param scaledAmount The amount of tokens to transfer (scaled).
```

CVF-12 INFO

- **Category** Unclear behavior
- **Source** VariableDebtToken.sol

Description It is unclear why two amounts are needed.

Recommendation Remove one of the amounts or clearly explain why both are needed.

Client Comment *Both amounts are passed as they are used within the code for backwards compatibility.*

```
85  uint256 amount,
+uint256 scaledAmount,
```

CVF-13 INFO

- **Category** Procedural
- **Source** DataTypes.sol

Recommendation The naming is inconsistent. Either use “scaledAmount” or “amountScaled” everywhere.

Client Comment *Ack, we consider aligning in a future release.*

```
241 -uint256 amount;
```

```
244 +uint256 scaledAmount;
```

```
301 -uint256 amount;
+uint256 amountScaled;
```

CVF-14 FIXED

- **Category** Documentation
- **Source** LiquidationLogic.sol

Description The returned value is not documented.

Recommendation Document it.

```
82 +) external returns (uint256) {
```

CVF-15 INFO

- **Category** Bad datatype
- **Source** ValidationLogic.sol

Recommendation The type for this argument should be more specific.

Client Comment *Ack, we consider introducing user defined types in a future version.*

```
491 +address oracle
```

CVF-16 FIXED

- **Category** Procedural
- **Source** TokenMath.sol

Recommendation Consider specifying as "^0.8.0" unless there is something special regarding this particular version.

```
2 +pragma solidity ^0.8.10;
```

CVF-19 FIXED

- **Category** Procedural
- **Source** VariableDebtTokenMainnetInstanceGHO.sol

Recommendation It is a good practice to put a comment into an empty block to explain why the block is empty.

```
15 +constructor(IPool pool, address rewardsController)
     ↪ VariableDebtToken(pool, rewardsController) {}
```

CVF-21 INFO

- **Category** Bad datatype
- **Source** VariableDebtTokenMainnetInstanceGHO.sol

Recommendation The type for this argument should be more specific.

Client Comment *Ack, we consider introducing user defined types in a future version.*

```
25 +address underlyingAsset,
```

CVF-23 FIXED

- **Category** Procedural
- **Source** VariableDebtTokenMainnetInstanceGHO.sol

Recommendation It is a good practice to put a comment into an empty block to explain why the block is empty.

```
52 +function updateDiscountDistribution(address, address, uint256,
   ↪ uint256, uint256) external {}
```

ABDK
Consulting

About us

Established in 2016, is a leading service provider in the space of blockchain development and audit. It has contributed to numerous blockchain projects, and co-authored some widely known blockchain primitives like Poseidon hash function.

The ABDK Audit Team, led by Mikhail Vladimirov and Dmitry Khovratovich, has conducted over 40 audits of blockchain projects in Solidity, Rust, Circom, C++, JavaScript, and other languages.

Contact

Email

dmitry@abdkconsulting.com

Website

abdk.consulting

Twitter

twitter.com/ABDKconsulting

LinkedIn

linkedin.com/company/abdk-consulting