OpenZeppelin | security

AAVE GHO Audit

August 12th, 2022

This security assessment was prepared by **OpenZeppelin**, protecting the open economy.

Table of Contents

Summary

The Aave team asked us to review the code for their GHO stablecoin, as described by governance. This includes the Aave facilitator for GHO, GHO aToken, GHO debt tokens, and the interest rate mechanism.

Type	Stablecoins
Timeline	From 2022-07-11 To 2022-07-27
Languages	Solidity

Total Issues	13 (6 resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	0 (0 resolved)
Medium Severity Issues	2 (0 resolved)
Low Severity Issues	0 (0 resolved)
Notes & Additional Information	11 (6 resolved)

Scope

We audited the `aave/gho` repository at the `8e693b33f344eeab6f3e9938cb43d6fe9abbb9fa` commit.

In scope were the following contracts:

```
contracts
├── facilitators
│   └── aave
│       ├── interestStrategy
│       │   ├── GhoDiscountRateStrategy.sol
│       │   └── GhoInterestRateStrategy.sol
│       ├── oracle
│       │   └── GhoOracle.sol
│       ├── stkAaveUpgrade
│       │   ├── StakedAaveV2Rev4.sol
│       │   └── interfaces
│       │       └── IGhoVariableDebtToken.sol
│       └── tokens
│           ├── GhoAToken.sol
│           ├── GhoVariableDebtToken.sol
│           ├── base
│           │   ├── GhoDebtTokenBase.sol
│           │   └── GhoIncentivizedERC20.sol
│           └── interfaces
│               ├── IAaveIncentivesController.sol
│               ├── IGhoAToken.sol
│               ├── IGhoDiscountRateStrategy.sol
│               ├── IGhoVariableDebtToken.sol
│               ├── IScaledBalanceToken.sol
│               └── IVariableDebtToken.sol
└── gho
    ├── GhoToken.sol
    └── interfaces
        ├── IBurnableERC20.sol
        ├── IGhoToken.sol
        └── IMintableERC20.sol
```

System overview, privileged roles and trust assumptions

High-level overview

GHO is a decentralized ERC20-compliant stablecoin that will be available on Ethereum mainnet. The token will be minted and burned under certain strategies, defined by different entities called Facilitators.

Facilitators are chosen by Aave governance and can be very different in nature as they all provide different stabilization mechanisms to keep GHO pegged.

Each Facilitator can mint up to a certain amount of GHO, defined by the maximum capacity of their buckets (a value set for each Facilitator by the Aave governance). In particular, the sum of the bucket levels for each Facilitator at any moment in time is the total supply of the token and the sum of all the buckets' maximum capacities is the maximum total supply.

Aave governance will propose the first Facilitator be the Aave protocol, with specific GHO aToken and GHO Variable Debt Tokens to be deployed. The main difference between a regular Aave market and the GHO market, is that GHO tokens will not be supplied by users, but by the Facilitator.

Borrowing, repaying and liquidating GHO

Borrowing

As in any other Aave market, users are able to borrow GHO at a specific borrow rate. However, since there is no supply side, the borrow interest rate will be static and modified as needed by the governance. In regular Aave markets, the borrow amount limit is the amount supplied by other users, but for the GHO Aave market, the borrow amount limit is given by its bucket's max capacity. The protocol allows users to always borrow GHO tokens at 1 USD/GHO (price retrieved by a Chainlink Oracle) regardless of the price of the token in parallel markets — as long as the bucket's max capacity has not been reached. This mechanism allows users to arbitrage: when market prices are above 1 USD, users can borrow GHO in the Aave market at the price of 1 USD and sell it to make a profit in parallel markets.

Users wishing to borrow GHO are incentivized to stake AAVE tokens in order to benefit from the discounted interest rate charged on borrowing GHO. Specifically, the discount rate in the audited commit is 20% (i.e., borrowers holding stkAAVE pay 20% less interest for the amount of GHO borrowed, limited based on the amount of stkAAVE).

Repaying and liquidating

Borrowers can also repay their debts or get liquidated as they would in typical Aave markets. For the former, in order to close their position and get their collateral back, borrowers have to return the GHO they borrowed plus any accrued interest. As with borrowing GHO, the protocol allows users that have borrowed to always repay GHO at 1 USD/GHO, which means that if borrowers can buy GHO in parallel markets at a discount, they can then repay in the Aave market at 1 USD/GHO and arbitrage a profit, helping maintain the peg.

Note: Unlike other over-collateralized stablecoins, the protocol does not provide a mechanism to influence demand of GHO through changes in monetary policy from the demand perspetive. At the time of this audit, there is no mechanism to earn yield with GHO, so the peg relies entirely on borrowers until new Facilitators are introduced.

Finally, as in any other Aave market, if the position becomes too risky because the collateral value has dropped below specified thresholds, the GHO borrower can get liquidated.

Privileged roles and trust assumptions

Users of GHO place utmost trust in the Aave DAO as well as GHO Facilitators. The Aave DAO can:

- Update the interest rate for GHO borrows to maintain a good ratio between supply and demand.
- Set the discount rate for `stkAAVE` holders who also make up a portion of the DAO votes.
- Set the max capacity of each Facilitator's bucket.

Additionally, since each Facilitator is capable of minting and burning GHO without additional permissions, they will need to be vetted by the DAO **AND** their implementation will need to be audited to ensure it maintains a peg of 1 USD/GHO. The DAO should carefully analyze how facilitators might impact one another when setting and updating bucket capacities in order to best maintain the peg.

Finally, the code provides functionality that is meant to retrieve the USD/ETH price from a Chainlink Oracle. We assumed this oracle to be honest, accessible, and properly functioning.

Findings

Here we present our findings.

Medium Severity

M-01 Facilitator bucket `maxCapacity` may inhibit peg dynamics

The `setFacilitatorBucketCapacity` function in the `GhoToken` contract allows the owner of the contract to either increase or decrease a given facilitator's bucket's `maxCapacity`. This storage variable represents the total amount of GHO tokens that a facilitator can mint. In the case of the Aave pool, the first facilitator, these tokens are lent to borrowers, and the amount of GHO tokens that can be borrowed is limited to this `maxCapacity`.

The Aave facilitator also works as a price stabilizer, since users can borrow, repay and liquidate GHO at 1 USD, which creates an arbitrage opportunity for borrowers:

- if the price is below the peg, borrowers can purchase GHO at a lower price in a parallel market, and then repay their debt using those tokens in the pool, making a profit.
- if the price is above the peg, users can borrow GHO at 1 USD and sell it in the parallel market.

The issue lies in the fact that when the `maxCapacity` is reached, under a scenario of high buy pressure in the market, arbitrageurs will not be able to borrow additional GHO from the Aave pool, and therefore not be able to sell and stabilize market prices around 1 USD. Likewise, if the `maxCapacity` is reached, it becomes impossible to source GHO from the facilitator to pay for accrued interest, unless bought in the parallel market, increasing the buy pressure and increasing its price further.

Given the necessary speed of response by the DAO in the case of worsening market conditions and potential GHO depegging, consider further research and implementing an algorithmic variable interest rate with a DAO permissioned backstop that can override the interest rate.

M-02 Aave facilitator requires recursive loans

GHO can support nearly limitless facilitators. However, at launch, in the audited commit, Aave will be the only GHO facilitator. As such, GHO can only be sourced by borrowing it through

Aave's lending pool. Users pay interest to borrow GHO, at a rate set depending on their staking status. When repaying GHO debt via the `LendingPool`, users need to transfer the principal borrowed plus interest on the debt using the underlying asset (i.e., GHO).

Since GHO can only be sourced by borrowing, and assuming the interest rate is above zero, it becomes impossible for a GHO loan to be repaid without another GHO loan first being originated. The additional loan originated is necessary to create a secondary market where the first borrower can source GHO to pay for accrued interest. This can have a domino effect on how GHO liquidity is borrowed and repaid depending on several factors, such as interest rates on borrowed GHO and supplied collateral, discounted interest rates for stakers borrowing GHO, and minting capacity limits.

This can be mitigated with non-interest-baring GHO liquidity in the market. Consider implementing additional facilitators with alternative ways to source GHO.

Notes & Additional Information

N-01 Inconsistent naming for scaled and non-scaled values

Throughout the codebase there are many places where "wadray" math is used to scale and descale balances, amounts, discounts, etc. In some cases, variable names include the word "scaled" to explicitly state that the value is a scaled value while in other places it does not even though the value is indeed scaled.

To ensure clarity and easy readability, consider explicitly stating whether an amount is scaled in the variable name.

Update: *Fixed at commit `d3f0cd3` on pull request #98.*

N-02 Use custom errors

Since solidity version 0.8.4, custom errors provide a cleaner, consistent, and more cost-efficient way to explain to users why an operation failed.

Multiple instances of hardcoded require messages were found throughout the codebase. For example:

- In `GhoDebtTokenBase.sol` on lines 115, 127, 133, 144, 155, and 166
- In `GhoIncentivizedERC20.sol` on lines 181, 182, 201, 217, 237, and 238
- In `GhoAToken.sol` on lines 292, 294, 303, and 349
- In `GhoVariableDebtToken.sol` on lines 67, 75, 280, and 395
- In `GhoToken.sol` on lines 34, 38, 80, 107, 119, 120, 121, and 136-139

Additionally, there are instances where an `Errors` library is used. If this is not needed for backward compatibility, we suggest updating these cases as well.

For conciseness, consistency, and gas savings, consider replacing hardcoded require and revert messages with custom errors.

Update: *This has been acknowledged and retained by the Aave team. Their reply:*

> *The current plan is to leave errors as is.*
>
> *The aave protocol - both v2 and v3 does not use custom errors. Keeping errors as is for the AToken and VariableDebtToken and their dependent contracts for the GHO reserve will keep it consistent with all of the other reserves in the protocol.*

N-03 Deprecated pragma for ABIEncoder V2

The interface `IAaveIncentivesController.sol` is compiled using a solidity version, specifically 0.8.10, where the pragma `experimental ABIEncoderV2` is deprecated and the `ABIEncoderV2` is enabled by default.

To adhere to the solidity documentation, consider either removing the deprecated pragma or adding the recommended explicit pragma `pragma abicoder v2;` instead.

Update: *Fixed at commit `59d8edc` on pull request #97.*

N-04 Implicit casting

The current lack of explicit casting when handling unsigned integer variables used in the `GhoToken` contract, considering the different types of unsigned integers in use (*e.g.* `uint256`, and `uint128`), hinders code readability, making it more error-prone and hard to maintain.

In particular, the below events emitted by the `GhoToken` contract use `uint256` parameters for *capacities* and *levels*, while those values are stored as `uint128` integers. Casting is implicit where the following events are called:

- FacilitatorAdded
- FacilitatorBucketCapacityUpdated
- BucketLevelChanged

Consider explicitly casting all integer values to their expected type when sending them as parameters of functions and events. It is advisable to review the entire codebase and apply this recommendation to all segments of code where the issue is found.

Update: *This has been acknowledged and retained by the Aave team. Their reply:*

> *Implementing changes for this increases gas in the mint and burn functions when the BucketLevelChanged event is emitted.*
>
> *The implicit conversion is relatively easy to follow and allows for gas savings in expensive functions. In this case, prioritizing gas savings makes sense.*

N-05 Incorrect natspec definitions

The NatSpec comments in `IGhoDiscountRateStrategy.sol` and `GhoDiscountRateStrategy.sol` define the parameters `debtBalance` and `discountTokenBalance` as "The address of the reserve" and "The liquidity available in the reserve" respectively, when they should be defined as "The debt balance of the user" and "The discount token balance of the user" respectively as they are in `refreshDiscountPercent`.

For clarity and correctness, consider updating the NatSpec definitions to properly define the parameters. It is advisable to review the entire codebase and apply this recommendation to all NatSpec comments.

Update: *Fixed at commit `16d9d64` on pull request #99.*

N-06 Lack of indexed parameter in event

In the contract `StakedTokenV2Rev4`, the event `RewardsAccrued` has an address parameter, `user`, that is not indexed.

To enable off chain monitoring, consider making the parameter indexed.

N-07 Mismatch between implementation and whitepaper

The whitepaper mentions that the total supply of GHO in a certain time *t* is given by the following formula:

$$S_{GHO_t} <= \sum_{i=0}^{n-1} C_{B_i}$$

Where:

- S_{GHO_t} is the total supply of GHO at t
- C_{B_i} is the capacity of the bucket B_i

However, this does not hold in all cases of bucket capacity and level values. The `setFacilitatorBucketCapacity` function in the `GhoToken` contract allows the owner to change the bucket max capacity to values below the current bucket level.

This means that if the owner of the contract updates the bucket's capacity from P to M with a current bucket level of N (where $M < N <= P$), then the bucket would be virtually overflowed, and the inequation mentioned above will not hold, since the total circulating supply could be greater than the sum of all bucket's capacities.

Consider updating the whitepaper accordingly. Additionally, consider checking that the total supply S_{GHO_t} is not being used as the total supply in other out-of-scope sections of the code, to avoid miscalculations.

Update: *This has been acknowledged by the Aave team. Their reply:*

> *Update to the white paper is necessary.*
>
> *In cases where Aave governance wants to restrict a facilitator from minting more GHO, governance may set their max capacity to a value less than the facilitators current level. This is allowable and would prevent any further minting until the facilitator has burned enough GHO to be below their new max.*
>
> *In this circumstance, governance cannot force a facilitator or it's users to burn GHO, so it may be unlikely that the level ever returns to below the new decreased max capacity.*

N-08 Not using SafeCast

The `mint` and `burn` functions cast the `newBucketValue` variable, defined as `uint256`, to `uint128`. In the `mint` function, the `newBucketLevel` variable could potentially be greater than the size of a `uint128`, but it is being implicitly caught by this require statement.

To reduce the potential for errors in the future, consider using SafeCast functions like `toUint128` when down-casting integers throughout the codebase to revert in case the integer is too large.

Update: *This has been acknowledged and retained by the Aave team. Their reply:*

> *Since this is implicitly caught by the require statement, in this specific case it is worth the gas savings to not perform the safecast.*
>
> `require(maxBucketCapacity >= newBucketLevel, 'FACILITATOR_BUCKET_CAPACITY_EXCEEDED');`
>
> *The maxBucketCapacity is a cached value that comes from the field `uint128 maxCapacity` in the facilitator's bucket struct. Since maxCapacity is saved as a uint128, `maxBucketCapacity` even though used as a uint256, will never hold a value more that type(uint128).max. Therefore the require statement removes the possibility of overflow by requiring maxBucketCapacity to be greater than or equal to the new bucket level.*

N-09 GhoToken is unlicensed

The `GhoToken.sol` file is unlicensed. Consider choosing an open source license for the project and updating the `SPDX-License-Identifier` accordingly.

Update: *Fixed at commit `e33d8da` on pull request #96.*

N-10 Unused functionality in GhoAToken.sol

In the contract GhoAToken.sol, the following functions are implemented but have no use, since it is not possible to mint or transfer the GhoAToken:

- `balanceOf`
- `scaledBalanceOf`
- `getScaledUserBalanceAndSupply`

- `totalSupply`
- `scaledTotalSupply`
- `permit`
- `_transfer`
- `_transfer`

For clarity and conciseness, consider replacing the function bodies with a dev comment or revert statement as in `mint` and `burn`. In the cases of balances and token supply, a hard coded value might be more appropriate.

Update: *Fixed at commit `48ffc2c` on pull request #95.*

N-11 Unused variable

In `GhoToken` contract, the variable `_facilitatorsCount` is defined but never used. For code cleaniness and clarity, consider removing any unused variables.

Update: *Fixed at commit `25751d3` on pull request #94.*

Conclusions

Two medium severity issues were found. Some recommendations were proposed to follow best practices and reduce the potential attack surface. We also recommend implementing monitoring and/or alerting functionality on key metrics such as Facilitator bucket capacity utilization.