AAVE

GHO Stablecoin

Smart Contract Security Assessment Report

Version: 3.0

March, 2023

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the GHO smart contracts. The review focused solely on the security aspects of the Solidity implementation of the contract, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the smart contract. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the GHO smart contracts contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

Outputs of automated testing that were developed during this assessment are also included for reference (in the Appendix: Test Suite).

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the AAVE smart contracts.

Overview

GHO is a collateral backed stablecoin designed to maintain a stable value in spite of market volatility. GHO can be natively integrated into the AAVE Protocol.

GHO will be minted through various strategies. These strategies can be enacted by different entities that may employ different approaches for integrating with GHO. For this reason, GHO introduces the concept of facilitators. A facilitator can generate (and burn) GHO tokens. Each facilitator is assigned a maximum limit to the amount of GHO it can generate. These limits are called *buckets*. When a facilitator mints GHO, its bucket level increases. When it burns GHO, its level decreases. The bucket can never fill above its maximum capacity. Note that, as GHO is fungible, it is possible for facilitators to burn tokens that were generated by other facilitators.

Security Assessment Summary

This review was conducted on the files hosted on the aave/gho and were assessed at commit 9328367 and again at commit 15a6b45. Final retesting activities targeted commit 7ae72de.

Note: the OpenZeppelin libraries and dependencies were excluded from the scope of this assessment.

The manual code review section of the report is focused on identifying any and all issues/vulnerabilities associated with the business logic implementation of the contracts. This includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout). Additionally, the manual review process focused on all known Solidity anti-patterns and attack vectors. These include, but are not limited to, the following vectors: re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers. For a more thorough, but non-exhaustive list of examined vectors, see [1, 2].

To support this review, the testing team used the following automated testing tools:

- Mythril: `https://github.com/ConsenSys/mythril`
- Slither: `https://github.com/trailofbits/slither`

Output for these automated tools is available upon request.

Findings Summary

The testing team identified a total of 10 issues during this assessment. Categorised by their severity:

- High: 1 issue.
- Low: 4 issues.
- Informational: 5 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the AAVE smart contracts. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the contracts, including gas optimisations, are also described in this section and are labelled as “informational”.

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected contract(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
GHO-01	Repaying GHO on behalf of another user records interest as paid for the sender, not the borrower	High	Resolved
GHO-02	Under standard configuration, it is functionally impossible to ever fully repay all GHO debts	Low	Closed
GHO-03	Missing checks when removing non-existent facilitators	Low	Resolved
GHO-04	`GHO` could be rescued	Low	Closed
GHO-05	The total supply of `GhoVariableDebtToken` is incorrect if there are active discounts	Low	Closed
GHO-06	Extensive facilitator privileges	Informational	Resolved
GHO-07	`GhoFlashMinter` relies on other sources of `GHO`	Informational	Closed
GHO-08	`GhoFlashMinter` free borrowers could resell their privileges	Informational	Resolved
GHO-09	`GhoFlashMinter` fee changes could disrupt arbitragers within the same block	Informational	Resolved
GHO-10	Miscellaneous General Comments	Informational	Resolved

GHO-01	Repaying GHO on behalf of another user records interest as paid for the sender, not the borrower		
Asset	`GhoAToken.sol`,`BorrowLogic.sol`(outside scope)		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

When a GHO loan is repaid, it is done through a call to `Pool.repay()`, the last parameter of which dictates which account the repayment is to be made for. This in turn calls `BorrowLogic.executeRepay()` which, on line [255], makes the call:

```
IAToken(reserveCache.aTokenAddress).handleRepayment(msg.sender, paybackAmount);
```

This will be a call to `GhoAToken.handleRepayment()` but, critically, the parameters contain only the address of the payment sender. The borrower's address, `params.onBehalfOf`, is not sent to `GhoAToken.handleRepayment()` at all.

Within `GhoAToken.handleRepayment()`, the interest owed is assessed on line [170]:

```
uint256 balanceFromInterest = _ghoVariableDebtToken.getBalanceFromInterest(user);
```

However, in this context, the value of `user` will be `msg.sender`, and so the interest value used will be that of the sender, not the borrower. The interest payment will also be credited to the sender, not the borrower because `user` is used to determine which account to credit on line [284]:

```
_ghoVariableDebtToken.decreaseBalanceFromInterest(user, amount);
```

Note that any GHO spent is still all burned from the sender and any `GhoVariableDebtToken` burned are still always burned from the borrower. The main impact of this issue is in the accounting and in terms of where the interest is paid to.

Consider a complete repayment of a GHO debt on behalf of another account in the two following scenarios:

1. **The sender has zero GHO debt**

   `balanceFromInterest` in `GhoAToken.handleRepayment()` will be zero, and so the full amount of the debt will be burnt on line [175]. The treasury will receive zero GHO when it should have received interest. Moreover, the value of `GhoVariableDebtToken._ghoUserState[user].accumulatedDebtInterest` will remain at the value of the interest owed by the borrower, even though that account would have no GHO, no `GhoAToken` and no `GhoVariableDebtToken`.

   The main impact of this issue is the lost interest to the treasury in this case.

   There is also an accounting error in `GhoVariableDebtToken._ghoUserState[user].accumulatedDebtInterest`. If the borrower borrows GHO in future, repayments will send this extra amount to the treasury, but without affecting the amount repaid. This has the effect of balancing the books by effectively paying the originally unpaid interest. However, the other effect of this accounting error is that calls to `GhoVariableDebtToken.getBalanceFromInterest()` will have unexpectedly high values, and this might cause security issues in future.

2. **The sender has a large GHO debt with a large owed interest amount**

   This scenario is more complex, but resolves very similarly to the first. The repaid amount is initially credited entirely to the sender's `GhoVariableDebtToken._ghoUserState[user].accumulatedDebtInterest`.

   The borrower's value of `GhoVariableDebtToken._ghoUserState[user].accumulatedDebtInterest` remains unchanged and the borrower can withdraw their collateral. The borrower's account behaves identically to the first scenario.

   The sender's account, despite having interest credited in the variable:

   ```
   GhoVariableDebtToken._ghoUserState[user].accumulatedDebtInterest
   ```

   still has the same balance of `GhoVariableDebtToken` as if it had not made any repayments, so this interest "credit" is illusory. The account will still need to repay its full amount of `GHO` and the amount that the treasury receives is the total amount owed by the sender.

   The borrower's interest, in this scenario, is lost in the same way as in the first.

   There are multiple accounting errors in `GhoVariableDebtToken._ghoUserState[user].accumulatedDebtInterest` throughout this scenario for both borrower and sender.

Recommendations

Consider modifying `BorrowLogic.executeRepay()` so that its calls to `GhoAToken.handleRepayment()` also pass on the value of `params.onBehalfOf`. This may be a desirable measure generally, given the stated purpose of `GhoAToken.handleRepayment()`:

```
* @dev The default implementation is empty as with standard ERC20 tokens, nothing needs to be done after the
* transfer is concluded. However in the future there may be aTokens that allow for example to stake the underlying
* to receive LM rewards. In that case, `handleRepayment()` would perform the staking of the underlying asset.
```

It is likely that a staking implementation of the kind described would stake for `user` (the sender) when it should stake for `params.onBehalfOf` (the borrower).

Resolution

The recommendation has been implemented in commit bfc76f0.

GHO-02	Under standard configuration, it is functionally impossible to ever fully repay all GHO debts		
Asset	`GhoToken.sol`, `GhoAToken.sol`, `GhoVariableDebtToken.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Because interest is charged on GHO debt without any new GHO entering the system, there is a constantly expanding gap between the total supply of GHO and the sum of all GHO debts as denoted by `GhoVariableDebtToken`. In essence, after any amount of time from the first GHO debt,

$$GHO.totalSupply < GhoVariableDebtToken.totalSupply$$

However, the difference between these two totals is equal to the amount of GHO that would be in the treasury if all the debts were repaid as much as possible with the existing GHO supply. In terms of the mathematics, if the treasury instantly recirculated these GHO tokens, the debts could be repaid.

In terms of practice on the blockchain, this does not seem to be possible, however. Attempts in tests to repay fully were all forced to wait at least one block by the AAVE pool error:

```
SAME_BLOCK_BORROW_REPAY = '48'; // 'Borrow and repay in same block is not allowed'
```

This means that interest is always accrued before debts can be repaid in full, even with a maximally cooperative treasury.

This issue is further accentuated by GHO-07

The practical impact of this issue is uncertain, and likely minimal. It is presumably unlikely that users will wish to all repay all GHO debts simultaneously, although it might harm the protocol if it is known that this is not possible. It may be that GHO may struggle with liquidity at times when many users wish to liberate their collateral. The behaviour of the treasury might also have an effect on these issues: if it trades away its GHO regularly, that will help to keep GHO supply up.

Also, if the system remains finely balanced in this way, any lost tokens would render debts theoretically unpayable simply because of limited GHO liquidity.

It might be theoretically possible to repay all GHO debts using accounts that pay no interest because of 100% discounts if those accounts paid off their debts last and the treasury cooperates by recirculating interest payments. However, 100% discounts may not be offered.

Recommendations

Be aware of this issue and, if the development team feels it is necessary, consider potential strategies to ensure that no user would ever be technically locked from repaying their GHO debt due to lack of supply.

Resolution

The development team have acknowledged the issue and decided to make no code changes at this time.

GHO-03	Missing checks when removing non-existent facilitators		
Asset	`GhoToken.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `_removeFacilitator()` function does not implement checks for non-existent facilitators which causes wrong events to be emitted.

When removing a facilitator, the `remove()` function of the `EnumerableSet` library will check if the facilitator exists prior to be removed, but the return value is not used. This will cause wrong events to be emitted.

Recommendations

One solution could be to add checks that require that the facilitator to be removed exists in the `facilitators` mapping by checking the length of its label. Another solution could be to handle the return value of the `remove()` function to revert when the facilitator to be removed does not exist.

Resolution

The recommendation has been implemented in commit 2933490.

GHO-04	`GHO` could be rescued		
Asset	`GhoAToken.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `rescueTokens()` function specifically blocks rescue of `_underlyingAsset`, which would be the `GHO` token. However, in the case of `GHO` it is not clear that this strictly needs to be the case. The `GhoAToken` contract does not hold a pool of `GHO` long term, although `GHO` is held during repayments to the pool.

Furthermore, it does seem like the token most likely to be sent to the `GhoAToken` contract in error would be `GHO`.

Recommendations

Consider whether it is in the protocol's interests to allow rescue of `GHO` tokens. If such a measure is implemented, it might be prudent to automatically send them to a specific safe address, perhaps the treasury.

Resolution

Changes made to the design of the `GhoAToken` contract have rendered this issue null as the contract now holds tokens.

However, the development team have clarified that the protocol's governance would have the option to distribute `GHO` funds that need to be rescued from this contract.

GHO-05	The total supply of `GhoVariableDebtToken` is incorrect if there are active discounts		
Asset	`GhoVariableDebtToken.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

`Openzeppelin`'s description of the `totalSupply()` function, inherited in `GhoVariableDebtToken.sol`, reads:

```
@dev Returns the amount of tokens in existence.
```

However, this is not quite what `GhoVariableDebtToken.totalSupply()` will do.

`GhoVariableDebtToken.balanceOf()` contains logic to reduce a user's debt token balance if that user has an interest discount from staked `AAVE`. However, this logic is (understandably) missing from `GhoVariableDebtToken.totalSupply()`.

As a result of this, the sum of all balances of all users will be less than the value returned by `GhoVariableDebtToken.totalSupply()`. This could potentially have security implications if any smart contract references the value of `GhoVariableDebtToken.totalSupply()` in its logic. It could also have broader financial implications if the value of `GhoVariableDebtToken.totalSupply()` is ever used in a financial estimate or report. If the difference is significant, this could potentially compromise the reputation of the protocol.

This issue is partially mitigated by the fact that `GhoVariableDebtToken` is not a tradeable token. It is also mitigated by the fact that discounts will be applied to the total supply whenever actions are taken that call `_mintScaled()` or `_burnScaled()`. There is nothing in the system, however, which prevents a holder of a large sum of `GhoVariableDebtToken` with a high discount from simply holding those tokens for a large period of time, creating a significant inaccuracy.

Consider that the value of `GhoVariableDebtToken.totalSupply()` is not a technical detail. It does have a financial meaning: the total `GHO` due to be repaid, and so an error in this value could realistically be consequential.

Recommendations

The logic changes required to make the value of `GhoVariableDebtToken.totalSupply()` precisely account for all discounts would not be trivial. The testing team suggests that a multiplier could be maintained which reflects the global discount of the entire balance of `GhoVariableDebtToken`. This would need to be updated whenever discounts or user balances of `GhoVariableDebtToken` change. Whenever the multiplier changed, it would also be necessary to apply the previous multiplier to the balance since the last change and record this new balance.

Alternatively, if the development team considers such measures unnecessary, communicate wherever possible that the value of `GhoVariableDebtToken.totalSupply()` will be an overestimate.

Resolution

The development team has acknowledged this issue as a limitation of the current implementation. The function comment header has been changed to reflect this in commit ae9704e.

GHO-06	Extensive facilitator privileges
Asset	`GhoToken.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

As facilitators have the ability to mint `GHO` simply by calling `GhoToken.mint()`, and because the `AAVE` price oracles will set `GHO`'s value at a hardcoded $1, it is worth emphasising the power that facilitators have.

Any new facilitator should be thoroughly reviewed, in particular for any exploit that might cause it to call `GhoToken.mint()` in an unapproved manner.

The bucket allowance system is a worthwhile mitigation for this risk, but it offers a limited protection. If `GHO` were minted excessively by a compromised facilitator, the price of the token on decentralised exchanges could suddenly drop, allowing other actors to purchase it and continue exploiting the protocol even if the bucket limits have been reached.

Recommendations

Be aware of this issue and keep all relevant considerations in mind with each facilitator added.

Resolution

The development team are aware of the issue and have provided the following comments.

Introducing new facilitators is a major risk. All new facilitators should be reviewed and go through the utmost level of security review. Additionally new facilitators should be added with low capacities, which can be increased over time.

GHO-07	`GhoFlashMinter` relies on other sources of `GHO`
Asset	`GhoFlashMinter.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

`GhoFlashMinter` mints and burns exclusively within `flashLoan()`, where it mints and burns the exact same amounts. It is therefore neither a net producer or consumer of `GHO`.

However, because there is a transfer of `GHO` on line [**103**] of `flashLoan()`, the contract does require access to additional `GHO` to function. This is not necessarily a problem but it does mean that the flash minter is dependant on other facilitators for this supply of `GHO` and will have the long term effect of transferring their `GHO` to the treasury.

Recommendations

Be aware of this issue and consider its effect on the protocol as a whole, in particular its potential impact on GHO-02.

Resolution

The development team have acknowledged the issue and decided to make no code changes at this time.

GHO-08	`GhoFlashMinter` free borrowers could resell their privileges
Asset	`GhoFlashMinter.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

`GhoFlashMinter` allows addresses to flash loan `GHO` with no fee if `_aclManager.isFlashBorrower()` returns true for that address.

It would be possible for an address granted this status to simply resell the ability to make these loans at a discount from the main protocol's fee, and keep the fees themselves.

Recommendations

Be aware of this issue, especially when granting borrower status via `_aclManager.isFlashBorrower()`.

Resolution

The development team are aware of the issue and have provided the following comments.

Only Aave governance can assign this privilege to addresses, and likely would not assign this privilege to an address that would resell it, or would revoke this privilege if re-selling was observed.

GHO-09	`GhoFlashMinter` fee changes could disrupt arbitragers within the same block
Asset	`GhoFlashMinter.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

`GhoFlashMinter.updateFee()` modifies the flash loan fee. Consider that calls to this function could interact with attempts to arbitrage, possibly causing arbitragers to suffer a loss.

If a call to `GhoFlashMinter.flashLoan()` is made based on a careful calculation that allows a small profit, assuming the current fee level, then if it is unintentionally frontrun by a call to `GhoFlashMinter.updateFee()`, that calculation would be rendered incorrect, and the arbitrager who made it could suffer a loss.

Recommendations

Be aware of this issue, and update flash fees minimally with appropriate warnings and notice.

Resolution

The development team are aware of the issue and have provided the following comments.

This value can only be changed by Aave governance. It is unlikely to be updated often, and arbitragers will know when it will be updated since it will need to go through the public Aave governance process.

GHO-10	Miscellaneous General Comments
Asset	`contracts/*`
Status	**Resolved:** See Resolution
Rating	Informational

Description

This section details miscellaneous findings that do not have direct security implications:

1. **Inline comments:** Whilst comment headers for functions are always present, there is no use of inline comments within the code. This makes the code unnecessarily difficult to read, especially in lengthy functions. Consider that it is in the interest of the protocol to make the code as clear as possible to all those who are working with it in common pursuit of the protocol's aims.
2. **Fixed oracle price potentially increases exposure:** In the event that a security vulnerability causes excess `GHO` tokens to be minted, consider the possibility that fixing the oracle price at $1 could potentially expose more of the protocol's assets than might be the case if the oracle were responsive to conditions in the market.
3. **Gas savings:**
   - In `GhoToken`, `GhoAToken` and `GhoVariableDebtToken` consider using Solidity's custom errors in place of the error strings.
   - In `GhoVariableDebtToken` line [**546**] is within an `else` block which ensures that the value of `newDiscountPercent` is zero. It is therefore possible to replace `newDiscountPercent` on this line with a hardcoded `0`.
   - In `GhoAToken`, the `onlyPool` modifier is unnecessary and can be removed from `mint()`, `burn()`, `mintToTreasury()` and `transferOnLiquidation()` as these functions will always revert. Removing this modifier will reduce the deployment cost.
4. **Empty mints and burns:** `GhoToken.mint()`, `GhoToken.burn()` can both accept `amount` values of zero. In this case, they will both still emit events. Consider whether this is desirable.
5. **Empty burns can be executed by any address:** Further to the previous point, `GhoToken.burn()` will accept zero burns from any address. In this case, it will emit a `BucketLevelChanged` event with that address as the `facilitatorAaddress`. This could potentially have security implications if this event data is ever used to compile a list of facilitators.
6. **`bucket.maxCapacity` can be set to below bucket.level:** In `GhoToken.setFacilitatorBucketCapacity()`, there is no check to ensure that the new bucket maximum is below the current level. This works as a way of lowering a facilitator's `maxCapacity` without first requiring burns. However, the development team and others building with `AAVE` should be careful never to assume that `bucket.maxCapacity >= bucket.level` is an invariant. Similarly, the total of all all bucket capacities cannot be assumed to be greater than the total of all bucket levels.
7. **Zero address checks:** Consider adding checks for `address(0)` in the following places:
   - `GhoToken.initialize()` parameters `treasury`, `underlyingAsset` and `incentivesController`.
   - `GhoToken.updateGhoTreasury()` parameter `newGhoTreasury`.
   - `GhoVariableDebtToken.initialize()` parameters `underlyingAsset` and `incentivesController`.

- `GhoVariableDebtToken.setAToken()` parameter `ghoAToken`.
- `GhoFlashMinter.constructor()` parameter `ghoTreasury`.
- `GhoFlashMinter.updateGhoTreasury()` parameter `newGhoTreasury`.

8. **Solidity version:** Use a fixed solidity version in every contract to avoid deploying a contract with a different version than expected.
9. **Unused variables:**
   - In `GhoAToken`, the variable `PERMIT_TYPEHASH` is never used.
   - In `GhoAToken`, the variable `_treasury` is replaced by `_ghoTreasury` but is still used in the `initialize()` and `RESERVE_TREASURY_ADDRESS()` functions.
10. **Unused library:** In `GhoAToken` the `SafeCast` library is never used.
11. **Possessive apostrophes in comments:**
    - On line [**50**] of `GhoVariableDebtToken.sol`, the word `users` should be `user's`.
    - On line [**54**] of `GhoVariableDebtToken.sol`, the word `users` should be `users'`.
12. **Naming convention not respected:** In `GhoVariableDebtToken`, the internal function `refreshDiscountPercent()` doesn't follow the naming convention. Consider renaming this function to `_refreshDiscountPercent()`.
13. **Typo in topic name:** In `IGhoToken`, the first topic of the `BucketLevelChanged` event should be `facilitatorAddress`.
14. **Discount entitlement comment:** On line [**57**] of `GhoVariableDebtToken.sol`, the comment describes `_discountLockPeriod` as the *minimum amount of time a user is entitled to a discount without performing additional actions*. This implies that discounts would reduce over time. How a discount changes over time is determined by `_discountRateStrategy`, so consider changing this wording to something like *amount of time a user's discount will not change without performing additional actions* which has the advantage of not assuming the behaviour of `_discountRateStrategy` or implying that discounts automatically decline over time due to the logic within `GhoVariableDebtToken`.

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Resolution

The development team have acknowledged these findings, addressing them where appropriate as follows:

1. **Inline comments**
   The development team have acknowledged the issue and decided to make no code changes at this time.
2. **Fixed oracle price potentially increases exposure**
   The issue has been acknowledged and they have provided the following comments.
   GHO will not be allowed as collateral in markets where the oracle is fixed which greatly reduces this risk. Additionally, the fixed oracle price is a key piece of the price stability mechanism.

3. **Gas savings**

   The development team judged it preferable to leave the text errors in place to maximise consistency with the rest of the protocol's code, as well as for reasons of code clarity.

4. **Empty mints and burns**

   In the case of empty mints, the development team have acknowledged the issue and decided to make no code changes at this time, with the following comment:

   mint() can only successfully be called by a facilitator even when the amount is zero. The msg.sender must be a facilitator with a `maxCapacity > 0`*. Facilitators should not be given permission if they allow empty mints*.

   The case of burns is resolved in the next issue.

5. **Empty burns can be executed by any address**

   This issue has been fixed in commit b4c03ef.

6. **`bucket.maxCapacity` can be set to below bucket.level**

   The development team have acknowledged the issue and decided to make no code changes at this time, with the following comment:

   This is expected functionality.

7. **Zero address checks**

   The development team have acknowledged the issue and decided to make no code changes at this time, with the following comment:

   All these functions are designed to be called by the governance after detailed assessment and review.

8. **Solidity version**

   The development team have acknowledged the issue and specified particular Solidity versions for several contracts in commit 896b49a.

9. **Unused variables**

   In both cases, the development team judged it preferable to leave the variables in place to maximise compliance with the general `AToken` specification, as well as for reasons of clarity.

10. **Unused library**

    The development team removed `SafeCast` in commit 94403cc.

11. **Possessive apostrophes in comments**

    The development team made the suggested changes in commit efa1363.

12. **Naming convention not respected**

    The development team modified the function name in commit 1262131.

13. **Typo in topic name**

    This issue has been fixed in commit 803067f.

14. **Discount entitlement comment**

    The development team modified the wording in commit 19f3171.

Appendix A Test Suite

A non-exhaustive list of tests were constructed to aid this security review and are given along with this document. The `Foundry` framework was used to perform these tests and the output is given below.

```
Running 12 tests for test/ERC20Test.t.sol:ERC20Test
[PASS] test_approve() (gas: 42364)
[PASS] test_burn() (gas: 28788)
[PASS] test_decimals() (gas: 5633)
[PASS] test_mint() (gas: 28790)
[PASS] test_name() (gas: 9819)
[PASS] test_permit() (gas: 105763)
[PASS] test_permit_expired() (gas: 51083)
[PASS] test_permit_wrongSigner() (gas: 77216)
[PASS] test_symbol() (gas: 9905)
[PASS] test_transfer() (gas: 46927)
[PASS] test_transferFrom() (gas: 62922)
[PASS] test_transferFrom_Unlimited() (gas: 78107)
Test result: ok. 12 passed; 0 failed; finished in 76.25ms

Running 9 tests for test/GhoFlashMinterTest.t.sol:GhoFlashMinterTest
[PASS] test_constructor() (gas: 7689)
[PASS] test_distributeFeesToTreasury() (gas: 90326)
[PASS] test_flashFee() (gas: 28793)
[PASS] test_flashLoan() (gas: 324970)
[PASS] test_getFee() (gas: 7685)
[PASS] test_getGhoTreasury() (gas: 9859)
[PASS] test_maxFlashLoan() (gas: 20822)
[PASS] test_updateFee() (gas: 40326)
[PASS] test_updateGhoTreasury() (gas: 35821)
Test result: ok. 9 passed; 0 failed; finished in 118.69ms

Running 4 tests for test/RepayTest.t.sol:RepayTest
[PASS] test_repayMoreDebt() (gas: 1004414)
[PASS] test_repayMoreDebtComplete() (gas: 1076973)
[PASS] test_repayNoDebt() (gas: 685468)
[PASS] test_repaySelf() (gas: 664193)
Test result: ok. 4 passed; 0 failed; finished in 305.72ms

Running 20 tests for test/TestGhoAToken.t.sol:TestGhoAToken
[PASS] test_DOMAIN_SEPARATOR() (gas: 22508)
[PASS] test_RESERVE_TREASURY_ADDRESS() (gas: 14921)
[PASS] test_UNDERLYING_ASSET_ADDRESS() (gas: 14950)
[PASS] test_balanceOf() (gas: 37105)
[PASS] test_burn() (gas: 18745)
[PASS] test_constructor() (gas: 18073)
[PASS] test_distributeFeesToTreasury() (gas: 76170)
[PASS] test_getGhoTreasury() (gas: 14942)
[PASS] test_getVariableDebtToken() (gas: 14974)
[PASS] test_initialize() (gas: 24142)
[PASS] test_mint() (gas: 18860)
[PASS] test_mintToTreasury() (gas: 16482)
[PASS] test_nonces() (gas: 20889)
[PASS] test_permit() (gas: 18901)
[PASS] test_rescueTokens() (gas: 110740)
[PASS] test_setVariableDebtToken() (gas: 31429)
[PASS] test_totalSupply() (gas: 17284)
[PASS] test_transferOnLiquidation() (gas: 18785)
[PASS] test_transferUnderlyingTo() (gas: 64114)
[PASS] test_updateGhoTreasury() (gas: 39243)
Test result: ok. 20 passed; 0 failed; finished in 320.30ms

Running 2 tests for test/SupplyTest.t.sol:SupplyTest
[PASS] testNotEnoughTotalSupply() (gas: 739976)
[PASS] testNotEnoughTotalSupplyAliceRepay() (gas: 738806)
Test result: ok. 2 passed; 0 failed; finished in 378.96ms
```

```
Running 5 tests for test/TestStakedTokenV2Rev4.t.sol:TestStakedTokenV2Rev4
[PASS] test_revision() (gas: 15701)
[PASS] test_transferEventfulRecipientWithInterest() (gas: 588267)
[PASS] test_transferEventfulSender() (gas: 544821)
[PASS] test_transferEventfulSenderWithInterest() (gas: 605056)
[PASS] test_transferUneventful() (gas: 486279)
Test result: ok. 5 passed; 0 failed; finished in 378.50ms

Running 30 tests for test/TestGhoVariableDebtToken.sol:TestGhoVariableDebtToken
[PASS] test_UNDERLYING_ASSET_ADDRESS() (gas: 14996)
[PASS] test_allowance() (gas: 18616)
[PASS] test_approve() (gas: 16422)
[PASS] test_balanceOf() (gas: 915499)
[PASS] test_burnBurn() (gas: 78462)
[PASS] test_burnMint() (gas: 241460)
[PASS] test_constructor() (gas: 18168)
[PASS] test_decreaseAllowance() (gas: 16421)
[PASS] test_decreaseBalanceFromInterest() (gas: 248520)
[PASS] test_getAToken() (gas: 14932)
[PASS] test_getBalanceFromInterest() (gas: 241875)
[PASS] test_getDiscountLockPeriod() (gas: 14890)
[PASS] test_getDiscountPercent() (gas: 378829)
[PASS] test_getDiscountToken() (gas: 14953)
[PASS] test_getUserRebalanceTimestamp() (gas: 538757)
[PASS] test_increaseAllowance() (gas: 16487)
[PASS] test_initialize() (gas: 24238)
[PASS] test_mintDiscounted() (gas: 486396)
[PASS] test_mintSimple() (gas: 93065)
[PASS] test_mintZero() (gas: 19487)
[PASS] test_onlyAToken() (gas: 18219)
[PASS] test_onlyDiscountToken() (gas: 20454)
[PASS] test_setAToken() (gas: 31470)
[PASS] test_totalSupply_discount() (gas: 545028)
[PASS] test_totalSupply_noDiscounts() (gas: 475039)
[PASS] test_transfer() (gas: 16474)
[PASS] test_transferFrom() (gas: 18678)
[PASS] test_updateDiscountLockPeriod() (gas: 36933)
[PASS] test_updateDiscountRateStrategy() (gas: 39271)
[PASS] test_updateDiscountToken() (gas: 39200)
Test result: ok. 30 passed; 0 failed; finished in 377.70ms

Running 23 tests for test/TestGhoToken.t.sol:TestGhoToken
[PASS] testAddFacilitator() (gas: 120144)
[PASS] testAddFacilitatorsWrongValues() (gas: 42517)
[PASS] testBasicBorrow() (gas: 557456)
[PASS] testBasicBorrowFuzz(uint256,uint256)
[PASS] testBorrowAndRepay() (gas: 645145)
[PASS] testBorrowAndRepayFuzz(uint256,uint256)
[PASS] testBorrowWithDiscountCase1() (gas: 845919)
[PASS] testBorrowWithDiscountCase2() (gas: 839846)
[PASS] testBucketCapacityExceeded() (gas: 472791)
[PASS] testFailBorrowBigAmount() (gas: 352404)
[PASS] testFailBorrowWithoutCollateral() (gas: 84278)
[PASS] testInvalidFacilitator() (gas: 13679)
[PASS] testNonFacilitatorEmptyBurn() (gas: 9137)
[PASS] testRemoveFacilitator() (gas: 39380)
[PASS] testRemoveFacilitatorNonExistent() (gas: 17902)
[PASS] testRemoveFacilitatorsNonEmptyLevel() (gas: 542745)
[PASS] testSetFacilitatorBucketCapacity() (gas: 40274)
[PASS] test_burn() (gas: 55168)
[PASS] test_constructor() (gas: 16052)
[PASS] test_getFacilitator() (gas: 15413)
[PASS] test_getFacilitatorBucket() (gas: 10172)
[PASS] test_getFacilitatorsList() (gas: 4740)
[PASS] test_mint() (gas: 70751)
Test result: ok. 23 passed; 0 failed; finished in 3.51s
```

Appendix B Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

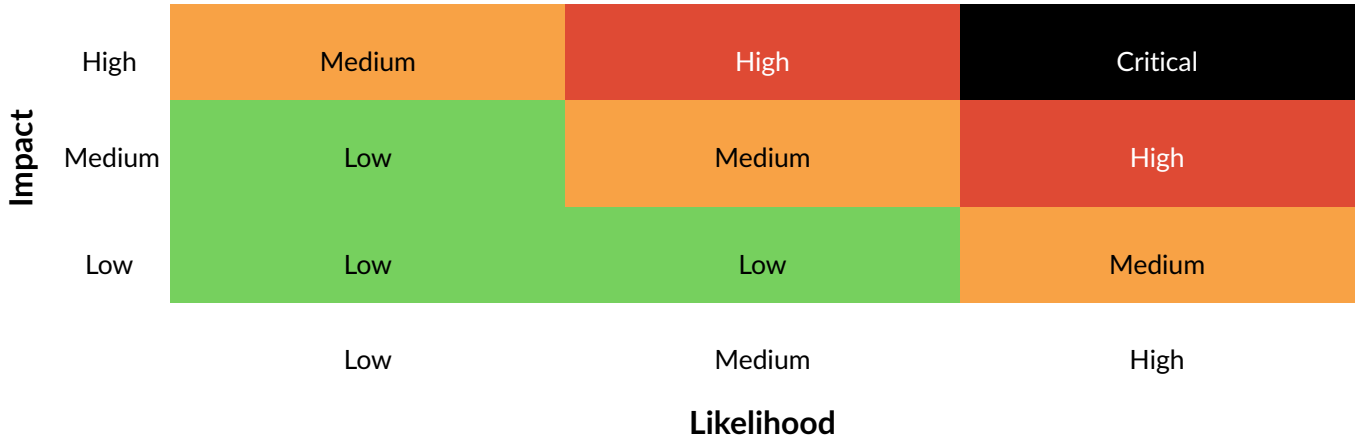

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References

[1] Sigma Prime. Solidity Security. Blog, 2018, Available: https://blog.sigmaprime.io/solidity-security.html. [Accessed 2018].

[2] NCC Group. DASP - Top 10. Website, 2018, Available: http://www.dasp.co/. [Accessed 2018].