Aave

GHO Stability Module Contract Review

Version: 2.4

October, 2023

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Aave 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 Aave 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 Stability Module provides a way to exchange between GHO and an underlying asset through buying and selling mechanisms. The underlying asset could be an ERC20 token or an ERC4626 token.

The `Gsm` contract is a GHO facilitator.

Security Assessment Summary

This review was conducted on the files hosted on the gho-core repository and were assessed at commit e4ea98c, then again at commit 7c03c52.

Specifically, the files in scope are as follows:

- `Gsm.sol`
- `Gsm4626.sol`
- `FixedFeeStrategy.sol`
- `GsmRegistry.sol`
- `SampleLiquidator.sol`
- `SampleSwapFreezer.sol`
- `FixedPriceStrategy.sol`
- `FixedPriceStrategy4626.sol`
- `GsmToken.sol`

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`
- Surya: `https://github.com/ConsenSys/surya`

Output for these automated tools is available upon request.

Findings Summary

The testing team identified a total of 7 issues during this assessment. Categorised by their severity:

- Medium: 1 issue.
- Low: 4 issues.
- Informational: 2 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
GSM-01	Potential Loss If Underlying Asset Depegs	Medium	Resolved
GSM-02	After Assets Are Seized, Underlying Asset Tokens Cannot be Rescued	Low	Resolved
GSM-03	A Rogue 4626 Vault Can Acquire Its GSM's Assets Through Price Manipulation	Low	Closed
GSM-04	Underlying Assets with High Precision Can be Acquired for Free in Small Quantities	Low	Resolved
GSM-05	Large ERC4626 Yields Will Not Be Accrued	Low	Resolved
GSM-06	Upgradeable Contracts Must Disable Initializers In The Implementation Contracts	Informational	Closed
GSM-07	Miscellaneous General Comments	Informational	Resolved

GSM-01	Potential Loss If Underlying Asset Depegs		
Asset	`Gsm.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The GSM system as reviewed deploys a fixed price strategy. If the price of the `UNDERLYING_ASSET` changes, this could be exploited to extract value from the GSM.

Consider the following scenario in which the `UNDERLYING_ASSET` is USDC.

1. USDC drops to 0.98, possibly for just a few blocks.
2. Alice flash mints 1,000,000 GHO and sells it for 1,020,408 USDC on the open market.
3. Alice sells that 1,020,408 USDC to the GSM for 1,020,408 GHO.
4. Alice repays the GHO flash minter the original 1M GHO and profits 20,408 GHO minus flash mint fees and gas.

This issue is mitigated by the ability to limit the GSM exposure cap, as well as its GHO bucket size. It can also be mitigated by the ability call `setSwapFreeze()` before the price drop could be exploited. However, this mitigation relies on an account with the `SWAP_FREEZER_ROLE` being able to call the function before any exploiting transaction can be completed.

Recommendations

Consider adding a maximum swap size that the GSM will perform within a certain period of time. Alternative protections could also be to check price oracles for larger swap transactions and revert if there is any recent price fluctuation.

Resolution

The development team stated that the AAVE DAO will be able to monitor the price of the `UNDERLYING_ASSET` and call `setSwapFreeze()` potentially before any price drop could be exploited. In the event that this is not successful, the scope of potential loss is limited by the GSM exposure cap, as well as its GHO bucket size.

The development team has implemented in PR#361 a new contract `OracleSwapFreezer` that would freeze the GSM contract when the underlying asset price oracle is depegged and is out of specified price range. This contract can also unfreeze the GSM when the price is back to a reasonable range. The contract `OracleSwapFreezer` is compatible with *Chainlink Automation* system.

GSM-02	After Assets Are Seized, Underlying Asset Tokens Cannot be Rescued		
Asset	`Gsm.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

If underlying assets are seized, it becomes impossible to rescue underlying asset tokens from the GSM unless `Gsm.getAvailableLiquidity()` tokens are left in the GSM contract.

When `seize()` is called, it transfers all tokens of the underlying asset out of the GSM, except for those needed to redeem the tokenised version of the asset. However, the value of `_currentExposure` is not modified, even though these assets are no longer held by the GSM.

`_currentExposure` is used in the calculation on line [**208**], within `rescueTokens()`, where the balance available to be rescued is calculated. It is subtracted from the GSM contract's asset balance. This will cause an arithmetic overflow error unless the balance is greater than `_currentExposure+_tokenizedAssets`. Furthermore, the test on line [**211**] will only allow asset tokens above this sum to be rescued.

Furthermore, the value `_isSeized` will be `true`, meaning that it will no longer be possible to reduce the value of `_currentExposure` by buying assets from the GSM.

```
198 function rescueTokens(
        address token,
200     address to,
        uint256 amount
202     ) external onlyRole(TOKEN_RESCUER_ROLE) {
        if (token == GHO_TOKEN) {
204         uint256 rescuableBalance = IERC20(token).balanceOf(address(this)) - _accruedFees;
            require(rescuableBalance >= amount, 'INSUFFICIENT_GHO_TO_RESCUE');
206     }
        if (token == UNDERLYING_ASSET) {
208         uint256 rescuableBalance = IERC20(token).balanceOf(address(this)) -
            _currentExposure -
210         _tokenizedAssets;
            require(rescuableBalance >= amount, 'INSUFFICIENT_EXOGENOUS_ASSET_TO_RESCUE');
212     }
        IERC20(token).safeTransfer(to, amount);
214     emit TokensRescued(token, to, amount);
    }
```

Recommendations

To resolve the issue modify the calculation in `rescueTokens` to disregard `_currentExposure` if `_isSeized` is `true`.

Resolution

The development team have fixed this issue by updating the `_currentExposure` to 0 in the function `seize()` in PR#361. Additionally, `seize()` updates the `_exposureCap` to 0.

GSM-03	A Rogue 4626 Vault Can Acquire Its GSM's Assets Through Price Manipulation		
Asset	`Gsm.sol`, `Gsm4626.sol` & `FixedPriceStrategy4626.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

A malicious vault installed as an underlying asset in a `Gsm4626` GSM can mismanage its own funds to manipulate its token's GHO price in the GSM, allowing large amounts of GHO to be acquired.

This attack is possible because the price calculations in `FixedPriceStrategy4626` rely on external calls to the vault's external functions `previewRedeem()` and `previewDeposit()`. The vault manager therefore only needs to incur losses from the vault to temporarily cause the asset price in the GSM to fluctuate.

Consider that the 4626 vault deploys a new strategy which deposits its underlying asset into another vault and then withdraws tokenised versions of this asset, which it counts as assets towards its internal asset/share ratio. If there is a point between deposit and withdrawal, the price of the 4626 asset as determined by `FixedPriceStrategy4626` would be vastly lower during this period.

Even if this period is within the middle of a single transaction, it would be possible for the vault manager to buy the `Gsm4626` GSM's entire balance of the underlying asset during this transaction, and then, once the share/asset values returned by the `ERC4626` vault token have returned to normal, sell them back to the GSM, or simply call the `withdraw()` method on the 4626 vault itself. If this were all within a single transaction, the attack could be implemented with a flash loan.

Note that it is not necessary in this attack for the vault to go entirely malicious, only for there to be any period at all in which the values returned by `previewRedeem()` and `previewDeposit()` are distorted. This could easily not be viewed as a significant vulnerability by the vault's governance or security views, as they would not be looking for this impact on GHO.

Recommendations

Consider recording the previous price or TWAP in `FixedPriceStrategy4626` and rejecting sudden price movements.

Consider keeping the values of `Gsm4626` GSMs low to limit this exposure.

Resolution

The development team have acknowledged this issue and have provided the following comments.

Gsm deployments must undergo assessment and review by the DAO to ensure appropriate configuration as well as underlying asset compatibility with the functionality of the GSM.

GSM-04	Underlying Assets with High Precision Can be Acquired for Free in Small Quantities		
Asset	`Gsm.sol` & `FixedPriceStrategy.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

If the underlying asset in `Gsm` has more decimal places than GHO, it is possible to buy small amounts of it from the `GSM` for free.

In a single call to `Gsm.buyAsset()`, it is possible to acquire for free an amount of the asset equal to the precision difference between that asset and GHO, minus one unit.

This is made possible by the calculation in `FixedPriceStrategy.getAssetPriceInGho()`, which has a precision defined by `PRICE_RATIO`. If the asset has a higher precision than this, it is possible to call `Gsm.buyAsset()` with an asset amount one below the precision boundary, and the price in GHO will not take this amount into account.

This issue is very low impact given the rarity of high precision tokens and the low amounts involved. In a single call to `Gsm.buyAsset()`, it is only possible to gain for free an amount equivalent to one base unit of GHO, ie. `1/1e18` of 1 GHO. Due to the cost of gas fees per transaction this attack is currently not economically feasible.

Recommendations

Consider modifying the calculation in `FixedPriceStrategy.getAssetPriceInGho()` to round the GHO amount up if there is a remainder.

Alternatively, never utilise the GSM with an underlying asset with a higher precision than GHO.

Resolution

The first recommendation has been implemented by adding a rounding direction to the price strategy. The development team use now in `FixedPriceStrategy.getAssetPriceInGho()` the function `mulDiv()` from the `Math` library with a rounding direction as input.

GSM-05	Large ERC4626 Yields Will Not Be Accrued		
Asset	`Gsm4626.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

If the accrued yield for a `Gsm4626` GSM is very large, no yield at all will be collected.

This issue only occurs in `_accrueYield()` only if the excess GHO that can be minted, `ghoExcess` is greater than the amount of GHO allowance remaining in the GHO facilitator bucket. According to the test on line [**63**], in this case no yield GHO is minted to the GSM at all.

This could conceivably occur if the amount of yield is unexpectedly high, perhaps spiking up due to some large unexpected windfall. In this case, the rewards collected by the GSM would be zero.

Ultimately, the issue could be resolved by increasing the bucket size for the GSM, or by users buying assets from the GSM, which would burn GSM from the bucket allowance, and also decrease the value of `ghoExcess`.

Recommendations

Consider modifying the calculation in `_accrueYield()` to mint GHO up to the facilitator bucket capacity in the case where minting the full excess would be too high.

Resolution

The recommendation has been implemented in PR#361. The function `_accrueYield()` has been renamed to `_cumulateYieldInGho()`, which would mint GHO up to facilitator bucket capacity if the GHO amount exceeds the amount available to mint, as shown in this code snippet.

```
function _cumulateYieldInGho() internal {
    (uint256 ghoCapacity, uint256 ghoLevel) = IGhoToken(GHO_TOKEN).getFacilitatorBucket(
      address(this)
    );
    uint256 ghoAvailableToMint = ghoCapacity > ghoLevel ? ghoCapacity - ghoLevel : 0;
    (uint256 ghoExcess, ) = _getCurrentBacking(ghoLevel);
    if (ghoExcess > 0 && ghoAvailableToMint > 0) {
      ghoExcess = ghoExcess > ghoAvailableToMint ? ghoAvailableToMint : ghoExcess;
      _accruedFees += uint128(ghoExcess);
      IGhoToken(GHO_TOKEN).mint(address(this), ghoExcess);
    }
  }
}
```

GSM-06	Upgradeable Contracts Must Disable Initializers In The Implementation Contracts
Asset	`Gsm.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The contract `Gsm` inherits from the contract `VersionedInitializable` which is a helper contract to support `initializer` functions. This latter doesn't implement a `_disableInitializers()` function and hence the `Gsm` implementation contract could be left uninitialized.

Recommendations

Avoid leaving the implementation contract uninitialized. This issue may be resolved by adding the `initializer` modifier to the `Gsm` constructor.

Resolution

This issue has been acknowledged by the development team.

GSM-07	Miscellaneous General Comments
Asset	`contracts/*`
Status	**Resolved:** See Resolution
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team that do not have direct security implications:

1. **No protection for mismatching token decimals**

   The following contracts may have the wrong number of decimals set in their constructors for their underlying assets: `FixedPriceStrategy4626` , `FixedPriceStrategy` , `FixedPriceStrategy4626` . Consider applying checks, noting that not all tokens implement the function `decimals()` .

2. **Update functions can update to same values**

   The following functions in `Gsm.sol` have no checks that the new value they are updating to is different from the previous value: `updateGsmToken()` , `updatePriceStrategy()` , `updateFeeStrategy()` , `updateExposureCap()` .

3. **Variable is misnamed**

   In `Gsm.seize()` , the variable `underlyingAfter` does not represent the amount of underlying asset that will be in the contract after the call, but rather the amount that will be transferred out by the call. In the interface, this value is referred to as `underlyingAmount` . Consider a more descriptive alternative, such as `underlyingSeizeAmount` .

Recommendations

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

Resolution

The comments above have been acknowledged by the development team, and relevant changes actioned where appropriate.

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 3 tests for test/FixedPriceStrategyTest.t.sol:FixedPriceStrategyTest
[PASS] test_FixedPriceStrategy_constructor() (gas: 9283)
[PASS] test_getAssetPriceInGho() (gas: 6932)
[PASS] test_getGhoPriceInAsset() (gas: 6978)
Test result: ok. 3 passed; 0 failed; 0 skipped; finished in 14.34s

Running 3 tests for test/FixedPriceStrategy4626Test.t.sol:FixedPriceStrategy4626Test
[PASS] test_FixedPriceStrategy4626_constructor() (gas: 12701)
[PASS] test_FixedPriceStrategy4626_getAssetPriceInGho() (gas: 50048)
[PASS] test_FixedPriceStrategy4626_getGhoPriceInAsset() (gas: 50338)
Test result: ok. 3 passed; 0 failed; 0 skipped; finished in 14.34s

Running 5 tests for test/GsmRegistryTest.t.sol:GsmRegistryTest
[PASS] test_addGsm() (gas: 139695)
[PASS] test_addGsm_reverts() (gas: 88430)
[PASS] test_constructor() (gas: 7701)
[PASS] test_removeGsm() (gas: 66670)
[PASS] test_removeGsm_reverts() (gas: 23352)
Test result: ok. 5 passed; 0 failed; 0 skipped; finished in 15.68s

Running 3 tests for test/GsmTokenTest.t.sol:GsmTokenTest
[PASS] test_GsmToken_burn() (gas: 64193)
[PASS] test_GsmToken_constructor() (gas: 7834)
[PASS] test_GsmToken_mint() (gas: 60440)
Test result: ok. 3 passed; 0 failed; 0 skipped; finished in 15.68s

Running 8 tests for test/GsmGsm4626Test.t.sol:GsmGsm4626Test
[PASS] test_Gsm4626_buyAsset_Vanilla() (gas: 180737)
[PASS] test_Gsm4626_buyAssetwithSig_Tokenised() (gas: 243277)
[PASS] test_Gsm4626_distributeFeesToTreasury_Vanilla() (gas: 163839)
[PASS] test_Gsm4626_distributeFeesToTreasury_Yield() (gas: 195290)
[PASS] test_Gsm4626_redeemTokenizedAsset() (gas: 249695)
[PASS] test_Gsm4626_sellAsset_Errors() (gas: 94673)
[PASS] test_Gsm4626_sellAsset_Vanilla() (gas: 174772)
[PASS] test_Gsm4626_updatePriceStrategy() (gas: 442010)
Test result: ok. 8 passed; 0 failed; 0 skipped; finished in 15.68s

Running 5 tests for test/FixedFeeStrategyTest.t.sol:FixedFeeStrategyTest
[PASS] test_getBuyFee() (gas: 5864)
[PASS] test_getGrossAmountFromTotalBought() (gas: 6053)
[PASS] test_getGrossAmountFromTotalBought_zero_TotalAmount() (gas: 5623)
[PASS] test_getGrossAmountFromTotalSold() (gas: 6053)
[PASS] test_getSellFee() (gas: 5843)
Test result: ok. 5 passed; 0 failed; 0 skipped; finished in 17.77s

Running 31 tests for test/GsmTest.t.sol:GsmTest
[PASS] test_Gsm_backWith() (gas: 284668)
[PASS] test_Gsm_burnAfterSeize() (gas: 166508)
[PASS] test_Gsm_buyAsset_Errors() (gas: 34434)
[PASS] test_Gsm_buyAsset_Vanilla() (gas: 164358)
[PASS] test_Gsm_buyAssetwithSig_Errors() (gas: 78338)
[PASS] test_Gsm_buyAssetwithSig_Tokenised() (gas: 224545)
[PASS] test_Gsm_buyAssetwithSig_Vanilla() (gas: 202545)
[PASS] test_Gsm_constructor() (gas: 16728)
[PASS] test_Gsm_distributeFeesToTreasury() (gas: 147344)
[PASS] test_Gsm_getAssetAmountForBuyAsset() (gas: 24849)
[PASS] test_Gsm_getAssetAmountForSellAsset() (gas: 24780)
[PASS] test_Gsm_getGhoAmountForBuyAsset() (gas: 24651)
[PASS] test_Gsm_getGhoAmountForSellAsset() (gas: 24601)
[PASS] test_Gsm_highPrecision() (gas: 5535778)
[PASS] test_Gsm_initialize() (gas: 37401)
[PASS] test_Gsm_redeemTokenizedAsset() (gas: 233391)
```

```
[PASS] test_Gsm_rescueTokens_Asset() (gas: 210915)
[PASS] test_Gsm_rescueTokens_Gho() (gas: 175277)
[PASS] test_Gsm_seize() (gas: 225891)
[PASS] test_Gsm_seizeRescue() (gas: 95092)
[PASS] test_Gsm_sellAssetWithSig_Errors() (gas: 77948)
[PASS] test_Gsm_sellAssetWithSig_Vanilla() (gas: 201593)
[PASS] test_Gsm_sellAsset_Errors() (gas: 94707)
[PASS] test_Gsm_sellAsset_Frozen() (gas: 29003)
[PASS] test_Gsm_sellAsset_Vanilla() (gas: 163570)
[PASS] test_Gsm_setSwapFreeze() (gas: 42442)
[PASS] test_Gsm_updateExposureCap() (gas: 34302)
[PASS] test_Gsm_updateFeeStrategy() (gas: 32953)
[PASS] test_Gsm_updateGhoTreasury() (gas: 33382)
[PASS] test_Gsm_updateGsmToken() (gas: 1300069)
[PASS] test_Gsm_updatePriceStrategy() (gas: 389681)
Test result: ok. 31 passed; 0 failed; 0 skipped; finished in 17.77s
Ran 7 test suites: 58 tests passed, 0 failed, 0 skipped (58 total tests)
```

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.

Impact \ Likelihood	Low	Medium	High
High	Medium	High	Critical
Medium	Low	Medium	High
Low	Low	Low	Medium

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].