Aave GSM Audit Report

GHO Stability Module (GSM)

Prepared by: Emanuele Ricci (StErMi), Independent Security Researcher

Date: Sept 11 to Sept 20, 2023

Introduction

A time-boxed security review of the **GHO Stability Module (GSM)** protocol was done by **StErMi**, with a focus on the security aspects of the application's smart contracts implementation.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where I try to find as many vulnerabilities as possible. I can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

About GHO Stability Module (GSM)

The GHO Stability Module (GSM) is a mechanism that enables the seamless conversion of an asset to/from GHO with the goal of facilitating peg stability. It includes configurable functionality such as modular price and fee strategies that determine conversion behaviour as well as roles that enable a variety of safety-oriented functionality such as the ability to freeze conversions. The GSM is intended to be deployed as a GHO facilitator.

References:

- GSM Temp Check
- GSM Development Update
- gho-core Repository

About StErMi

Emanuele, aka **StErMi**, is an independent smart contract security researcher. He serves as a Security Researcher at Spearbit and has identified multiple bugs in the wild on Immunefi and on protocol's bounty programs like the Aave Bug Bounty.

Do you want to connect with him?

- stermi.xyz website
- @StErMi on Twitter

Summary & Scope

***review commit hash* - 841b0aa59c71f634b5bd8c251824d94cd1412687 *fixes review commit hash* - 7c03c52c0a271120837cd8e2c2750fe12b9f00c4**

Scope

The following smart contracts were in scope of the audit:

- `src/facilitators/gsm/feeStrategy/interfaces/IGsmFeeStrategy.sol`
- `src/facilitators/gsm/feeStrategy/FixedFeeStrategy.sol`
- `src/facilitators/gsm/interfaces/IGsm.sol`
- `src/facilitators/gsm/interfaces/IGsm4626.sol`
- `src/facilitators/gsm/misc/GsmRegistry.sol`
- `src/facilitators/gsm/misc/IGsmRegistry.sol`
- `src/facilitators/gsm/misc/SampleLiquidator.sol`
- `src/facilitators/gsm/misc/SampleSwapFreezer.sol`
- `src/facilitators/gsm/priceStrategy/interfaces/IGsmPriceStrategy.sol`
- `src/facilitators/gsm/priceStrategy/FixedPriceStrategy.sol`
- `src/facilitators/gsm/priceStrategy/FixedPriceStrategy4626.sol`
- `src/facilitators/gsm/Gsm.sol`
- `src/facilitators/gsm/Gsm4626.sol`

Note: during the fix period Aave has introduced the `OracleSwapFreezer` contract. Inside the finding you will see suggestions about it but it's worth noting that such contract is not part of the initial scope and the review period's scope.

Severity classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact - the technical, economic and reputation damage of a successful attack **Likelihood** - the chance that a particular vulnerability gets discovered and exploited **Severity** - the overall criticality of the risk

Findings Summary

ID	Title	Severity	Status
[M-01]	The `Gsm` contract can be initialized with the treasury equal to `address(0)` that could lead to the total loss of the accrued fees and seized `underlying`	Medium	Fixed
[M-02]	`Gsm` do not check the `receiver` of `transfer` / `safeTransfer` operations, possibly allowing the loss of tokens	Medium	Acknowledged
[M-03]	`getAssetAmountForSellAsset` and `getAssetAmountForBuyAsset` could return different values compared to `getGhoAmountForSellAsset` and `getGhoAmountForBuyAsset` because of rounding errors	Medium	Fixed
[M-04]	`Gsm._updatePriceStrategy` should further validate the new Price Strategy, ensuring its compatibility with the previous one	Medium	Acknowledged
[L-01]	`Gsm` will return erroneous values for excess, liquidity and exposure even after that it has been seized and all deficit burned	Low	Fixed
[L-02]	`Gsm.sellAssetWithSig` and `Gsm.buyAssetWithSig` do not follow the EIP-2612 standard when checking the `deadline`	Low	Fixed
[L-03]	`GsmRegistry.constructor` does not perform any sanity check on `owner` , allowing to set it to `address(0)`	Low	Fixed

ID	Title	Severity	Status
[L-04]	`FixedPriceStrategy` and `FixedPriceStrategy4626` should not be deployed with `PRICE_RATIO` equal to `0`	Low	Fixed
[L-05]	Consider avoiding executing the `safeTransfer` in `seize()` if `underlyingBalance` is equal to zero to avoid reverts in case of "weird" `ERC20` implementations	Low	Fixed
[L-06]	`Gsm` could be initialized with `admin` as `address(0)` preventing the configurator to operate and setup additional role members like Liquidator and Freezer	Low	Fixed
[L-07]	`Gsm4626._cumulateYieldInGho` should always mint as many `GHO` as possible of the yield instead of skipping the fee accrual	Low	Fixed
[L-08]	Contracts are using a floating pragma declaration	Low	Fixed
[L-09]	`CONFIGURATOR` can execute `Gsm.backWith` after that the `Gsm` has been seized, locking the `underlying` used to back it with	Low	Fixed
[L-10]	Some "gifted" underlying will be locked forever in `Gsm` if gifted after `seize()`	Low	Fixed
[L-11]	`Gsm.buyAsset` will stop working as soon as the `priceStrategy` is updated to a one with a higher (of just `1 wei`) `PRICE_RATIO`	Low	Acknowledged
[I-01]	Consider renaming `Gsm _ghoTreasury` as `_gsmTreasury` to be consistent with the new role of the treasury	Info	Acknowledged
[I-02]	Consider to explicitly initializing the `feeStrategy` during the execution of `Gsm.initialize`	Info	Acknowledged
[I-03]	Consider emitting an explicit event for the `Gsm.initialize` function	Info	Acknowledged
[I-04]	`Gsm.burnAfterSeize` should revert early if `amount` is equal to zero for a better DX	Info	Fixed
[I-05]	`Gsm.rescueTokens` should revert when `amount == 0` to avoid wasting gas and emitting spam events	Info	Fixed

ID	Title	Severity	Status
[I-06]	`Gsm` functions executable behind an auth role should explicitly pass the `msg.sender` in the event emission	Info	Acknowledged
[I-07]	Consider refactoring `Gsm.setSwapFreeze` to make the code more clean	Info	Fixed
[I-08]	`SampleLiquidator` is not implementing the `triggerBurnAfterSeize` function that should be callable by the Liquidator Role	Info	Fixed
[I-09]	Consider validating the `underlyingAssetDecimals` constructor parameter of both the `FixedPriceStrategy` and `FixedPriceStrategy4626` contracts	Info	Acknowledged
[I-10]	Consider tracking the value change of `_currentExposure` with a specific event	Info	Acknowledged
[I-11]	`Gsm.getGhoAmountForBuyAsset` and `Gsm.getGhoAmountForSellAsset` should accept at max `uint128` amounts given that the `buyAsset/sellAsset` can accept at max `type(uint128).max` underlying	Info	Fixed
[I-12]	Consider differentiating between the sold asset and the gifted asset inside the `Gsm.seize` logic	Info	Acknowledged
[I-13]	Avoid distributing the fees to the treasury when `_accruedFees` is equal to `0`	Info	Fixed
[I-14]	Aave should extensively document how they plan to act based on the underlying price fluctuation	Info	Acknowledged
[I-15]	`Gsm.getAssetAmountForBuyAsset` and `Gsm.getAssetAmountForSellAsset` use `uint128` for both inputs and return values	Info	Fixed
[I-16]	The `FixedPriceStrategy4626` does not adhere to the `IGsmPriceStrategy` interface	Info	Fixed
[I-17]	`Gsm4626` that uses `ERC4626` vaults with fees will experience `GHO` excess or deficit based on the fee change	Info	Acknowledged
[I-18]	`USDC` and `USDT` could lock `Gsm` underlying if the `Gsm` is blacklisted	Info	Acknowledged

ID	Title	Severity	Status
[I-19]	Consider refactoring `FixedPriceStrategy4626` by inheriting `FixedPriceStrategy`	Info	Acknowledged
[I-20]	General Natspec documentation improvements, and style suggestions	Info	Partially Fixed
[I-21]	Open questions, suggestions and discussions	Info	Partially Fixed
[G-01]	Consider calling `_beforeBuyAsset` and `_beforeSellAsset` after performing sanity/base checks to save gas	Gas	Acknowledged
[G-02]	In `Gsm._sellAsset_` `_currentExposure` could be updated after the `require` to save gas on revert	Gas	Fixed
[G-03]	Consider saving `_ghoTreasury` in a local variable to avoid an additional `SLOAD` during `Gsm.seize()` execution	Gas	Acknowledged
[Fix Review Period-01]	Issues and Suggestions on the changes made during the fix period	Multiple	Partially Fixed
[Fix Review Period-02]	`OracleSwapFreezer` considerations	Multiple	Partially Fixed

Detailed Findings

[M-01] The `Gsm` contract can be initialized with the treasury equal to `address(0)` that could lead to the total loss of the accrued fees and seized `underlying`

Context

- Gsm.sol#L108

- Gsm.sol#L274
- Gsm.sol#L210

Severity

Impact: High

- `_accruedFees` (in GHO) will be lost when `gsm.distributeFeesToTreasury` is called by anyone
- the whole `underlying` balance held by the `Gsm` is lost when the Liquidator calls `gsm.seize()`

Likelihood: Low The deployment and configuration of a `Gsm` contract is done by the Aave DAO should verify that during the initialization of the contract the treasury is correctly passed as an input parameter.

Description

The `initialize()` function of the `Gsm` contract does not perform any sanity check on the input parameter `ghoTreasury` that could be passed as `address(0)` .

The `_ghoTreasury` is set as `address(0)` and never updated, the following scenarios could happen:

- All the `_accruedFees` will be lost when `distributeFeesToTreasury` is executed. `distributeFeesToTreasury` can be called by anyone as soon as some fees have been accrued because of a sell, buy or ERC4626 yield accrual. This is possible because `GHO` is an `ERC20` that uses a "Solmate-like" implementation that allows the transfer of an amount of token to the `address(0)` during a transfer or transferFrom operation.
- If the `underlying` token inherit from a `ERC20` implementation that allows a `transfer` or `transferFrom` to `address(0)` , all the underlying balance held by the `Gsm` will be lost when the `Liquidator` executes the `seize()` function.

Test

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import './TestGhoBase.t.sol';
import {ERC20 as SolmateERC20} from '../contracts/gho/ERC20.sol';

contract MintableSolmateERC20 is SolmateERC20 {
  constructor(
    string memory _name,
    string memory _symbol,
    uint8 _decimals
  ) SolmateERC20(_name, _symbol, _decimals) {}
```

```
  function mint(address account, uint256 amount) external {
    require(amount > 0, 'INVALID_MINT_AMOUNT');
    _mint(account, amount);
  }
}

contract SGsmTest is TestGhoBase {
  address internal gsmSignerAddr;
  uint256 internal gsmSignerKey;

  function setUp() public {
    (gsmSignerAddr, gsmSignerKey) = makeAddrAndKey('gsmSigner');
  }

  function testTreasuryAddressZero() public {
    // Create a Solmate-like ERC20 token to use as an underlying for Gsm
    // This is needed to prove that seize could "fail" silently sending tokens to addr
    MintableSolmateERC20 solmateUnderlying = new MintableSolmateERC20(
      'GsmUnderlying',
      'GsmUnderlying',
      6
    );

    // The Gsm contract can be initialized with an empty treasury
    Gsm gsm = setUpGsm(address(0), address(solmateUnderlying));

    // in Gsm there are functions that sends assets toward the treasury
    // that in this case would be `address(0)`
    // if the `asset` does not perform any check on the `receiver` of the transfer/tra
    // the Gsm contract could end up sending those asset to an empty address
    // Function to test:
    // - seize: it could send the gsm underlying liquidity to the address(0)
    // - distributeFeesToTreasury: it would send all the GHO accumulated as fee to the

    // GHO token is based on the Solmate ERC20 implementation and does not perform suc
    // For the Gsm `underlying` it depends on the specific underlying implementation

    // ALICE sell 100 underlying tokens to Gsm
    sellUnderlying(gsm, ALICE, 100e6);

    // get the info needed for later assertions
    uint256 accruedFees = gsm.getAccruedFees();
    uint256 gsmUnderlyingBalance = solmateUnderlying.balanceOf(address(gsm));

    // when someone (because ANYONE can execute this function) call `distributeFeesToT
    // it will trasfer the fees to `address(0)`

    address randomUser = makeAddr('randomUser');
    vm.prank(randomUser);
    gsm.distributeFeesToTreasury();

    // assert that now `address(0)` owns the treasury fees
    assertEq(gsm.getAccruedFees(), 0);
```

```
    assertEq(GHO_TOKEN.balanceOf(address(gsm)), 0);
    assertEq(GHO_TOKEN.balanceOf(address(0)), accruedFees);

    // the Liquidator now needs to seize the underlying that have been sold by Alice t
    // the operation will try to transfer it directly to the treasury
    vm.prank(address(GHO_GSM_LAST_RESORT_LIQUIDATOR));
    gsm.seize();

    // assert that the Gsm has no more underlying
    // and that the underlying has been transferred to `address(0)`
    assertEq(solmateUnderlying.balanceOf(address(gsm)), 0);
    assertEq(solmateUnderlying.balanceOf(address(0)), gsmUnderlyingBalance);
  }

  ////////////////////////////////////////////////////
  /////// Utility functions
  ////////////////////////////////////////////////////

  function setUpGsm(address treasury, address underlying) internal returns (Gsm) {
    FixedPriceStrategy fixedPriceStrategy = new FixedPriceStrategy(
      DEFAULT_FIXED_PRICE,
      underlying,
      6
    );

    Gsm gsm = new Gsm(address(GHO_TOKEN), underlying);
    gsm.initialize(address(this), treasury, address(fixedPriceStrategy), DEFAULT_GSM_U

    // set the fee strategy
    gsm.updateFeeStrategy(address(GHO_GSM_FIXED_FEE_STRATEGY));

    // setup roles
    gsm.grantRole(GSM_LIQUIDATOR_ROLE, address(GHO_GSM_LAST_RESORT_LIQUIDATOR));
    gsm.grantRole(GSM_SWAP_FREEZER_ROLE, address(GHO_GSM_SWAP_FREEZER));

    // setup the new gsm as a facilitator
    IGhoToken(address(GHO_TOKEN)).addFacilitator(
      address(gsm),
      'GSM Facilitator Tester',
      DEFAULT_CAPACITY
    );

    return gsm;
  }

  function sellUnderlying(Gsm gsm, address seller, uint256 amountToSell) internal {
    address gsmUnderlying = gsm.UNDERLYING_ASSET();

    vm.prank(FAUCET);
    MintableSolmateERC20(gsmUnderlying).mint(seller, amountToSell);

    vm.startPrank(seller);
    MintableSolmateERC20(gsmUnderlying).approve(address(gsm), amountToSell);
```

```
    gsm.sellAsset(uint128(amountToSell), seller);
    vm.stopPrank();
  }
}
```

Recommendations

Aave should add a sanity check on the `initialize()` function to prevent the `Gsm` contract to be initialized with `_ghoTreasury` equal to `address(0)`.

A more elegant solution, that would also emit the needed `GhoTreasuryUpdated` would be to create a `_updateGhoTreasury()` function that will be directly called by both the `initialize` and `updateGhoTreasury` function.

By doing this Aave can ensure that

- The `_ghoTreasury` cannot be initialized as `address(0)`
- The `GhoTreasuryUpdated` is correctly called

```
  function initialize(
    address admin,
    address ghoTreasury,
    address priceStrategy,
    uint128 exposureCap
  ) external initializer {
    _grantRole(DEFAULT_ADMIN_ROLE, admin);
    _grantRole(CONFIGURATOR_ROLE, admin);
-    _ghoTreasury = ghoTreasury;
+    _updatePriceStrategy(ghoTreasury);
    _updatePriceStrategy(priceStrategy);
    _updateExposureCap(exposureCap);
  }
```

```
  /// @inheritdoc IGhoFacilitator
  function updateGhoTreasury(address newGhoTreasury) external override onlyRole(CONFIG
-    require(newGhoTreasury != address(0), 'ZERO_ADDRESS_NOT_VALID');
-    address oldGhoTreasury = _ghoTreasury;
-    _ghoTreasury = newGhoTreasury;
-    emit GhoTreasuryUpdated(oldGhoTreasury, newGhoTreasury);
+    _updateGhoTreasury(newGhoTreasury);
  }
```

```
+  /**
+   * @notice Updates GHO Treasury
```

```
+   * @param newGhoTreasury The value of the new GHO Treasury
+   */
+  function _updateGhoTreasury(address newGhoTreasury) internal {
+    require(newGhoTreasury != address(0), 'ZERO_ADDRESS_NOT_VALID');
+    address oldGhoTreasury = _ghoTreasury;
+    _ghoTreasury = newGhoTreasury;
+    emit GhoTreasuryUpdated(oldGhoTreasury, newGhoTreasury);
+  }
```

Discussion

Security Researcher

The recommendations have been implemented correctly in the commit 7c03c52c0a271120837cd8e2c2750fe12b9f00c4 (included in the PR 369)

Aave

We consider this issue low severity, given that GSM configuration goes through DAO assessment and review before going in production.

Security Researcher

The likelihood of the issue is Low (the minimum) because the `GSM` configuration goes through DAO assessment and review before going into production. But the Impact is still high (funds loss). The `seize` (authed) function, if `_ghoTreasury` is empty, will lock the whole underlying contained in the GSM. The `_ghoTreasury` (permissionless) function, will lock the whole amount of fees accrued into the GSM.

Because the impact remains High, the severity (following any standard Severity Categorization matrix) is **Medium**.

[M-02] `Gsm` do not check the `receiver` of `transfer` / `safeTransfer` operations, possibly allowing the loss of tokens

Context

- Gsm.sol#L187
- Gsm.sol#L210
- Gsm.sol#L274

- Gsm.sol#L407
- Gsm.sol#L438

Severity

Impact: High

- If the `asset` is `GHO` it will be lost because `GHO` does not check if the `receiver` is `address(0)`
- If the `asset` is a generic ERC20 token, it will depend on the implementation. OZ-like tokens will revert, but other implementations like Solmate-like will allow the transfer of the token to `address(0)` with the result of the loss of funds

Likelihood: Low Specifying `address(0)` as the receiver should not happen

Description

There are different functions in the `Gsm` contract where `GHO`, `UNDERLYING` or a general `ERC20` like token is transferred to a `receiver` or the `treasury`.

In all those functions, the `receiver` part of the `*transfer` operation is never checked, and the contract allows performing a transfer to the `address(0)`. If the `ERC20` implementation of the token does allow transferring the tokens to the `address(0)` those tokens will be forever lost.

Recommendations

Aave should always revert if the `receiver` of the `safeTransfer` / `transfer` operation is `address(0)`

Discussion

Aave

Acknowledged This is the intended behaviour of the contract.

[M-03] `getAssetAmountForSellAsset` and `getAssetAmountForBuyAsset` could return different values compared to `getGhoAmountForSellAsset` and

`getGhoAmountForBuyAsset` because of rounding errors

Context

- Gsm.sol#L326-L337
- Gsm.sol#L302-L309
- Gsm.sol#L312-L323
- Gsm.sol#L292-L299

Severity

Impact: Low If integrators use the values returned by the function to verify that the result of the buy/sell operation matches what they expect (from the result of `getAssetAmountForSellAsset` and `getAssetAmountForBuyAsset`) they could end up reverting their operation

Likelihood: High It's very likely that a rounding error could happen given that the user can specify an arbitrary `amount` of asset to be sold/bought

Description

In the `IGsm` docs for Aave explicitly specify and suggest that `getAssetAmountForSellAsset` and `getAssetAmountForBuyAsset` should be used to calculate the amount based on the `GHO` amount to sell/buy when the user/contract executes the `sellAsset` , `sellAssetWithSig` , `buyAsset` or `buyAssetWithSig` .

By performing fuzz testing, it resulted that the values returned by `getAssetAmountForSellAsset` and `getAssetAmountForBuyAsset` could not match the one returned by `getGhoAmountForSellAsset` and `getGhoAmountForBuyAsset` .

`getGhoAmountForSellAsset` and `getGhoAmountForBuyAsset` functions have been used for the test because they return the values that are internally used by the `sellAsset` , `sellAssetWithSig` , `buyAsset` or `buyAssetWithSig` external functions (see `_calculateGhoAmountForSellAsset` and `_calculateGhoAmountForBuyAsset`).

If the sell/buy operations are executed by a smart contract that tries to match the result of the operation with the values returned by `getAssetAmountForSellAsset` and `getAssetAmountForBuyAsset` , depending on the arbitrary `amountOfAssetToSellOrBuy` input parameter specified it could decide to revert because those values do not match because of rounding error.

Test

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import './TestGhoBase.t.sol';
import {ERC20 as SolmateERC20} from '../contracts/gho/ERC20.sol';

contract MintableSolmateERC20 is SolmateERC20 {
  constructor(
    string memory _name,
    string memory _symbol,
    uint8 _decimals
  ) SolmateERC20(_name, _symbol, _decimals) {}

  function mint(address account, uint256 amount) external {
    require(amount > 0, 'INVALID_MINT_AMOUNT');
    _mint(account, amount);
  }
}

contract SGsmLensTest is TestGhoBase {
  using PercentageMath for uint256;
  using PercentageMath for uint128;

  address internal gsmSignerAddr;
  uint256 internal gsmSignerKey;

  function setUp() public {
    (gsmSignerAddr, gsmSignerKey) = makeAddrAndKey('gsmSigner');
  }

  function testGetAssetAmountForSellAssetAsExpected(
    uint256 startPriceRatio,
    uint256 assetAmount
  ) public {
    startPriceRatio = bound(startPriceRatio, 0.5 ether, 2 ether);
    assetAmount = bound(assetAmount, 100e18, 100_000_000e18);

    // Create a token with 18 decimals
    MintableSolmateERC20 solmateUnderlying = new MintableSolmateERC20(
      'GsmUnderlying',
      'GsmUnderlying',
      18
    );

    Gsm gsm = setUpGsm(TREASURY, address(solmateUnderlying), 100_000_000e18, 100_000_0

    // Price Strategy of 1:1
    FixedPriceStrategy fixedPriceStrategy = new FixedPriceStrategy(
      startPriceRatio,
      address(solmateUnderlying),
```

```
    18
  );
  gsm.updatePriceStrategy(address(fixedPriceStrategy));

  // Alice sell the asset
  (uint256 ghoSentToUser, uint256 ghoGrossAmountWithFee1, uint256 feeAmount1) = gsm
    .getGhoAmountForSellAsset(assetAmount);

  // is it equivalent to the returned amount here?
  (uint256 assetAmountSold, uint256 ghoGrossAmountWithFee2, uint256 feeAmount2) = gsm
    .getAssetAmountForSellAsset(ghoSentToUser);

  // validate that they return the same values
  assertEq(
    ghoGrossAmountWithFee1,
    ghoGrossAmountWithFee2,
    'ASSERT_EQ_ERROR__GHO_GROSS_AMOUNT_WITH_FEE'
  );
  assertEq(feeAmount1, feeAmount2, 'ASSERT_EQ_ERROR__GHO_FEE_AMOUNT');
  assertEq(assetAmount, assetAmountSold, 'ASSERT_EQ_ERROR__AMOUND_SOLD');
}

function testGetAssetAmountForBuyAssetAsExpected(
  uint256 startPriceRatio,
  uint256 assetAmount
) public {
  startPriceRatio = bound(startPriceRatio, 0.5 ether, 2 ether);
  assetAmount = bound(assetAmount, 100e18, 100_000_000e18);

  // Create a token with 18 decimals
  MintableSolmateERC20 solmateUnderlying = new MintableSolmateERC20(
    'GsmUnderlying',
    'GsmUnderlying',
    18
  );

  Gsm gsm = setUpGsm(TREASURY, address(solmateUnderlying), 100_000_000e18, 100_000_0

  // Price Strategy of 1:1
  // uint256 startPriceRatio = 1e18;
  FixedPriceStrategy fixedPriceStrategy = new FixedPriceStrategy(
    startPriceRatio,
    address(solmateUnderlying),
    18
  );
  gsm.updatePriceStrategy(address(fixedPriceStrategy));

  // Alice sell the asset
  (uint256 ghoSoldByUserWithFee, uint256 ghoNeededForExchange1, uint256 feeAmount1) =
    .getGhoAmountForBuyAsset(assetAmount);

  // is it equivalent to the returned amount here?
  (uint256 assetAmountBought, uint256 ghoNeededForExchange2, uint256 feeAmount2) = g
```

```
      .getAssetAmountForBuyAsset(ghoSoldByUserWithFee);

    // validate that they return the same values
    assertEq(
      ghoNeededForExchange1,
      ghoNeededForExchange2,
      'ASSERT_EQ_ERROR__GHO_NEEDED_FOR_EXCHANGE'
    );
    assertEq(feeAmount1, feeAmount2, 'ASSERT_EQ_ERROR__GHO_FEE_AMOUNT');
    assertEq(assetAmount, assetAmountBought, 'ASSERT_EQ_ERROR__ASSET_AMOUNT');
  }

  ////////////////////////////////////////////////////
  /////// Utility functions
  ////////////////////////////////////////////////////

  function setUpGsm(
    address treasury,
    address underlying,
    uint128 exposureCap,
    uint128 facilitatorCapacity
  ) internal returns (Gsm) {
    FixedPriceStrategy fixedPriceStrategy = new FixedPriceStrategy(
      DEFAULT_FIXED_PRICE,
      underlying,
      18
    );

    Gsm gsm = new Gsm(address(GHO_TOKEN), underlying);
    gsm.initialize(address(this), treasury, address(fixedPriceStrategy), exposureCap);

    // set the fee strategy
    gsm.updateFeeStrategy(address(GHO_GSM_FIXED_FEE_STRATEGY));

    // setup roles
    gsm.grantRole(GSM_LIQUIDATOR_ROLE, address(GHO_GSM_LAST_RESORT_LIQUIDATOR));
    gsm.grantRole(GSM_SWAP_FREEZER_ROLE, address(GHO_GSM_SWAP_FREEZER));

    // setup the new gsm as a facilitator
    IGhoToken(address(GHO_TOKEN)).addFacilitator(
      address(gsm),
      'GSM Facilitator Tester',
      facilitatorCapacity
    );

    return gsm;
  }
}
```

Recommendations

Aave should notify the integrators that, because of rounding error, the values of `getAssetAmountForSellAsset` and `getAssetAmountForBuyAsset` could differ (of a rounding error) from the result of the buy/sell operation.

Discussion

Security Researcher

The recommendations have been implemented in the commit 7c03c52c0a271120837cd8e2c2750fe12b9f00c4 (included in the PR 369)

Security Researcher (fix review period)

The behavior of `sellAsset*` and `buyAsset*` has changed. When users sell assets they could end up selling less assets for the same amount of `GHO` and when they buy assets they could end up receiving more assets compared to what has been specified as the function's input parameter. This behavior could produce side effects that could end up reverting the transaction based on both the `underlying ERC20` token used by the `GSM` or how the caller is implemented.

1. When user calls `sellAsset*`, the `GSM` will execute `IERC20(UNDERLYING_ASSET).safeTransferFrom(originator, address(this), assetAmount);` where `assetAmount` could be lower compared to `maxAmount` specified by the user. The user/smart contract has pre-approved `maxAmount` and if `maxAmount < assetAmount` the allowance that the user/smart contract has given to the `GSM` will not be fully consumed. Tokens like `USDT` will revert if the user/smart contract set the allowance `> 0` when the allowance is not equal to `0`. If this scenario happens (not all the allowance has been consumed) the transaction will revert.
2. In general, such behavior (selling less tokens, receiving more tokens) should be well documented to warn the user of such unexpected behavior. Aave should also document which are the configurations (given `amount`, `underlying` decimals and `buy/sell` fees) that this scenario could happen. Aave does not know which checks/requirements the smart contracts/integrators will put in place after that the `sell/buy` asset operation has been executed.

Aave

We acknowledge the issue, but also note that the documentation around `buyAsset` and `sellAsset` in `IGsm` makes it explicit that input parameters represent minimum/maximum amounts (and are named as such) and, thus, may deviate (we do not advertise it in our docs as an exact input). We also return the exact amount that was bought/sold for those functions to ensure the amount of an asset bought/sold can be known without guesswork. Integrators need to ensure they adhere to the interface provided, the same as for any other contract.

[M-04] `Gsm._updatePriceStrategy` should further validate the new Price Strategy, ensuring its compatibility with the previous one

Context

- Gsm.sol#L490-L498

Severity

Impact: High Deploying a price strategy that has, for example, a different `UNDERLYING_ASSET_DECIMALS` (compared to the previous one currently in use in the `Gsm`) would disrupt the correct calculation of the swap between the underlying and `GHO` on the `Gsm`.

Likelihood: Low The deployment and update of the new `FixedPriceStrategy` should be executed as a Governance proposal. I assume that all these steps are carefully reviewed by the DAO, community and all security party involved that would prevent such misconfiguration.

Description

With the current implementation of the `_updatePriceStrategy` the `Gsm` only checks that the new price strategy uses the same `underlying` used by the `Gsm` (and implicitly check that the `priceStrategy` is not `address(0)`).

The function should perform additional checks like, for example, checking that the new `priceStrategy` is also using the same `UNDERLYING_ASSET_DECIMALS` used by the current `Gsm _priceStrategy`.

Recommendations

Aave should add a check that verifies that the `IGsmPriceStrategy(priceStrategy).UNDERLYING_ASSET_DECIMALS()` is equal to `IGsmPriceStrategy(_priceStrategy).UNDERLYING_ASSET_DECIMALS()`.

Aave could also add additional checks to ensure that the delta `PRICE_RATIO` between the two price strategy is between a lower/upper bound.

Note: The `PRICE_RATIO` sanity check has already been suggested in a separate issue.

Discussion

Aave

Acknowledged, this is an item to ensure is appropriately configured when deploying a GSM which will go through DAO assessment and review.

[L-01] `Gsm` will return erroneous values for excess, liquidity and exposure even after that it has been seized and all deficit burned

Context

- Gsm.sol#L340-L342
- Gsm.sol#L345-L347
- Gsm.sol#L350-L353

Severity

Impact: Low There is no loss of capital

Likelihood: Low The scenario would happen after that a `Gsm` has been `seized`. The seize operation should happen only in a very drastic scenario, with hopefully a low likelihood.

Description

After that a `Gsm` has been seized, all the underlying token contained in the `Gsm` itself will be transferred to the `_ghoTreasury`. The natural consequence of this is that the `Gsm` will have no more liquidity (of the underlying) inside the contract.

After a seize event, the `Liquidator` will also call `burnAfterSeize` to burn all the `GHO` that have been minted by the facilitator (the `Gsm` itself) and that is not backed anymore by any underlying.

After those events, if there have been no token "gifted" to the `Gsm` we should be in this scenario

- `GHO.balanceOf(address(gsm))` should return zero (consequence of `seize`)
- `(, uint256 ghoMinted) = GHO.getFacilitatorBucket(address(gsm));` should return zero (consequence of `burnAfterSeize`)
- All the operations on the `Gsm` reverts (there's a special case for `backWith` that has been reported in another issue). The only operations that should be possible to do would be

`rescueTokens` and `burnAfterSeize` (the last one would revert because there are no more minted `GHO` to burn)

While the `seize` has effectively transferred all the underlying to the treasury, it has not updated the `_currentExposure` (that represents the underlying sold to the `Gsm`) to zero.

Because of this, the following functions will return a wrong value:

- `getAvailableUnderlyingExposure()` returns a value that most probably is above zero. This function should return zero because it's not possible anymore to buy or sell assets
- `getAvailableLiquidity` returns a value greater than zero. This is wrong because there's no more underlying liquidity inside the `Gsm`
- `getCurrentBacking()` will return `(excess, 0)` with `excess > 0` because the `ghoMinted` will be equal to zero (the facilitator has been "resetted") but `_currentExposure` is still `> 0`

Test

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import './TestGhoBase.t.sol';
import {ERC20 as SolmateERC20} from '../contracts/gho/ERC20.sol';

contract MintableSolmateERC20 is SolmateERC20 {
  constructor(
    string memory _name,
    string memory _symbol,
    uint8 _decimals
  ) SolmateERC20(_name, _symbol, _decimals) {}

  function mint(address account, uint256 amount) external {
    require(amount > 0, 'INVALID_MINT_AMOUNT');
    _mint(account, amount);
  }
}

contract SGsmWrongValuesAfterSeizeTest is TestGhoBase {
  using PercentageMath for uint256;
  using PercentageMath for uint128;

  address internal gsmSignerAddr;
  uint256 internal gsmSignerKey;

  function setUp() public {
    (gsmSignerAddr, gsmSignerKey) = makeAddrAndKey('gsmSigner');
  }

  function testGsmWrongGetterValuesAfterSeize() public {
    // Alice sell 100 underlying
```

```
    // Price from 1:1 GHO goes to 1:0,5 GHO
    // Aave decide to seize the gsm
    (Gsm gsm, MintableSolmateERC20 solmateUnderlying) = setupTest(0.5e18);

    // Because the price of underlying has dropped there's a deficit and the GSM CONFI
    // could call the `backWith` (even if it shouldn't given that the contract has bee
    (, uint256 ghoMinted) = IGhoToken(GHO_TOKEN).getFacilitatorBucket(address(gsm));
    console.log('ghoMinted', ghoMinted);

    // Liquidator burn the deficit of GHO after the seize event
    ghoFaucet(address(GHO_GSM_LAST_RESORT_LIQUIDATOR), ghoMinted);
    vm.startPrank(address(GHO_GSM_LAST_RESORT_LIQUIDATOR));
    GHO_TOKEN.approve(address(gsm), ghoMinted);
    gsm.burnAfterSeize(ghoMinted);
    vm.stopPrank();

    (uint256 excess, uint256 deficit) = gsm.getCurrentBacking();
    // the excess is equal to the value in GHO of the `_currentExposure`
    // _currentExposure is 100 underlying and now the pice is 1:0,5
    // so excess will be equal to 50 GHO
    // but it should be equal to zero because there's no more GHO minted
    // and all the underlying has been transferred to the treasury by the `seize()` ex
    assertEq(excess, 50e18);

    // aftert the `burnAfterSeize` the deficit has been resetted
    assertEq(deficit, 0);

    uint256 availableUnderlyingExposure = gsm.getAvailableUnderlyingExposure();

    // this value is equal to the cap less the exposure.
    // the correct value in this case should be equal to the cap because there's no mo
    // but `_currentExposure` is still 100 underlying
    // note 100_000_000e18 is the exposure cap used when the gsm has been initialized
    assertEq(availableUnderlyingExposure, 100_000_000e18 - 100e18);

    uint256 availableLiquidity = gsm.getAvailableLiquidity();
    // this value should return zero because there's no more underlying in the Gsm
    // but returns 100 underlying because `_currentExposure` has not been resetted
    assertEq(availableLiquidity, 100e18);
  }

  function setupTest(uint256 newFixedPrice) internal returns (Gsm, MintableSolmateERC2
    // Create a token with 18 decimals
    MintableSolmateERC20 solmateUnderlying = new MintableSolmateERC20(
      'GsmUnderlying',
      'GsmUnderlying',
      18
    );

    // Deploy a Gsm with a 18 decimal underlying
    // The price strategy now is 1:1 -> 1 underlying === 1 GHO
    Gsm gsm = setUpGsm(TREASURY, address(solmateUnderlying), 100_000_000e18, 100_000_0
```

```
    // remove the price strategy just to make things easier
    gsm.updateFeeStrategy(address(0));

    // Alice sell 100 token to get 100 GHO
    sellUnderlying(gsm, ALICE, 100e18);

    // Alice should have received 100 GHO
    // Gsm should have 100 underlying
    assertEq(GHO_TOKEN.balanceOf(address(ALICE)), 100e18);
    assertEq(solmateUnderlying.balanceOf(address(gsm)), 100e18);

    // The price strategy has not changed so there should be no excess or deficit
    (uint256 excess, uint256 deficit) = gsm.getCurrentBacking();
    assertEq(excess, 0);
    assertEq(deficit, 0);

    // Update the fixed price based on the new underlying market value
    FixedPriceStrategy newFixedPriceStrategy = new FixedPriceStrategy(
      newFixedPrice,
      address(solmateUnderlying),
      18
    );
    gsm.updatePriceStrategy(address(newFixedPriceStrategy));

    // Aave decide to seize and stop the operation
    vm.prank(address(GHO_GSM_LAST_RESORT_LIQUIDATOR));
    gsm.seize();
    assertEq(solmateUnderlying.balanceOf(address(gsm)), 0);

    return (gsm, solmateUnderlying);
  }

  ////////////////////////////////////////////////////
  /////// Utility functions
  ////////////////////////////////////////////////////

  function setUpGsm(
    address treasury,
    address underlying,
    uint128 exposureCap,
    uint128 facilitatorCapacity
  ) internal returns (Gsm) {
    FixedPriceStrategy fixedPriceStrategy = new FixedPriceStrategy(
      DEFAULT_FIXED_PRICE,
      underlying,
      18
    );

    Gsm gsm = new Gsm(address(GHO_TOKEN), underlying);
    gsm.initialize(address(this), treasury, address(fixedPriceStrategy), exposureCap);

    // set the fee strategy
    gsm.updateFeeStrategy(address(GHO_GSM_FIXED_FEE_STRATEGY));
```

```
    // setup roles
    gsm.grantRole(GSM_LIQUIDATOR_ROLE, address(GHO_GSM_LAST_RESORT_LIQUIDATOR));
    gsm.grantRole(GSM_SWAP_FREEZER_ROLE, address(GHO_GSM_SWAP_FREEZER));

    // setup the new gsm as a facilitator
    IGhoToken(address(GHO_TOKEN)).addFacilitator(
      address(gsm),
      'GSM Facilitator Tester',
      facilitatorCapacity
    );

    return gsm;
  }

  function sellUnderlying(Gsm gsm, address seller, uint256 amountToSell) internal {
    address gsmUnderlying = gsm.UNDERLYING_ASSET();

    vm.prank(FAUCET);
    MintableSolmateERC20(gsmUnderlying).mint(seller, amountToSell);

    vm.startPrank(seller);
    MintableSolmateERC20(gsmUnderlying).approve(address(gsm), amountToSell);

    gsm.sellAsset(uint128(amountToSell), seller);
    vm.stopPrank();
  }

  function backWithUnderlying(Gsm gsm, address backer, uint256 amountToBack) internal
    address gsmUnderlying = gsm.UNDERLYING_ASSET();

    vm.prank(FAUCET);
    MintableSolmateERC20(gsmUnderlying).mint(backer, amountToBack);
    vm.startPrank(backer);
    MintableSolmateERC20(gsmUnderlying).approve(address(gsm), amountToBack);

    gsm.backWith(address(gsmUnderlying), uint128(amountToBack));
    vm.stopPrank();
  }

  function backWithGHO(Gsm gsm, address backer, uint256 amountToBack) internal {
    ghoFaucet(backer, amountToBack);
    vm.startPrank(backer);
    GHO_TOKEN.approve(address(gsm), amountToBack);
    gsm.backWith(address(GHO_TOKEN), uint128(amountToBack));
    vm.stopPrank();
  }
}
```

Recommendations

Aave should

- reset the `_currentExposure` state variable to zero when the `seize()` function is executed
- reset the `_exposureCap` state variable to zero when the `seize()` function is executed or simply by executing `updateExposureCap()` later on

By doing so, `getAvailableUnderlyingExposure`, `getAvailableLiquidity`, `getCurrentBacking` will return values that are in sync with the real state of the `Gsm`

Discussion

Security Researcher

The recommendations have been implemented in the commit 7c03c52c0a271120837cd8e2c2750fe12b9f00c4 (included in the PR 369)

[L-02] `Gsm.sellAssetWithSig` and `Gsm.buyAssetWithSig` do not follow the EIP-2612 standard when checking the `deadline`

Context

- Gsm.sol#L126
- Gsm.sol#L156

Severity

Impact: Low The buy/sell operation is rejected, but the signature can be rebuilt by the signer to re-execute the same operation in the future.

Likelihood: Low Edge case where the user has executed the operation at `block.timestamp` equal to `deadline`

Description

The EIP-2612 standard specification states the following about the `deadline` check that must be implemented

> The current blocktime is less than or equal to `deadline`

The current implementation of both `buyAssetWithSig` and `sellAssetWithSig` does not handle the edge case where `block.timestamp` is equal to the `deadline`.

By applying the `EIP-2612` standard, the operation, under that scenario, should be executed correctly, but it would revert using the `Gsm` implementation.

Recommendations

Aave should update the `buyAssetWithSig` and `sellAssetWithSig` to follow the `EIP-2612` standard and allow the execution of those operations even when the `block.timestamp` is equal to `deadline`

Discussion

Security Researcher

The recommendations have been implemented in the commit 7c03c52c0a271120837cd8e2c2750fe12b9f00c4 (included in the PR 369)

[L-03] `GsmRegistry.constructor` does not perform any sanity check on `owner`, allowing to set it to `address(0)`

Context

- GsmRegistry.sol#L22-L24
- Ownable.sol#L78-L82

Severity

Impact: Low While it's true that all the `onlyOwner` functions won't be usable anymore, the `GsmRegistry` could be instantly re-deployed and replaced. No other contracts in the `Gsm` ecosystem rely on the `GsmRegistry`.

Likelihood: Low The contract should be deployed by the Aave DAO that would perform this kind of sanity checks

Description

The `GsmRegistry.constructor` initialize the `_owner` variable (inherited by the OpenZeppelin `Ownable` contract) to the input parameter `owner` by executing the internal function `_transferOwnership(owner);` .

```
function _transferOwnership(address newOwner) internal virtual {
    address oldOwner = _owner;
    _owner = newOwner;
    emit OwnershipTransferred(oldOwner, newOwner);
}
```

This function is implemented inside the `Owanble` contract, and it does not perform any sanity check on the `newOwner` parameter. This could allow the deployer to mistakenly set the owner of the contract to `address(0)` .

By doing that, all the functions that use the `onlyOwner` modifier will be forever locked.

Recommendations

Aave should add a sanity check to the `owner` input parameter used in the `GsmRegistry.constructor` to prevent setting the owner of the contract to `address(0)` .

Another suggestion would be to make the `GsmRegistry` inherit from the `Ownable2Step` instead of the `Ownable` contract. This change would prevent the current owner from directly transferring the ownership to an address that would not be able to accept it.

Discussion

Security Researcher

The recommendations have been implemented in the commit 7c03c52c0a271120837cd8e2c2750fe12b9f00c4 (included in the PR 369)

Aave has decided not to add support to the `Ownable2Step` implementation.

[L-04] `FixedPriceStrategy` and `FixedPriceStrategy4626` should not be deployed with `PRICE_RATIO` equal to `0`

Context

- FixedPriceStrategy.sol#L33

- FixedPriceStrategy4626.sol#L34

Severity

Impact: Low When the `Gsm*` is updated to a `priceStrategy` that has `PRICE_RATIO` equal to `0`, every buy/sell assets operation will always revert. The `backWith` operation would allow the `CONFIGURATOR` to back the `Gsm` with an infinite amount of underlying.

Likelihood: Low The `FixedPriceStrategy` and `FixedPriceStrategy4626` contracts should be deployed by the Aave Governance. The governance will probably also update the different `Gsm*` to use the new price strategy. We can assume that those operations will be verified before executed.

Description

The current implementation of `FixedPriceStrategy` and `FixedPriceStrategy4626` allows the strategy to be initialized with a `PRICE_RATIO` equal to `0`. This means that when a user sells `X` amount of an asset, it would receive `0 wei` of `GHO` in exchange.

Consequences on the execution of `sellAsset*` or `buyAsset*`

When the user executes `sellAsset*` or `buyAsset*`, the functions will revert when they try to execute the `GHO.mint(0)` and `GHO.burn(0)` called inside `sellAsset*` and `buyAsset*` respectively.

Both `GHO.mint` and `GHO.burn` revert when the input parameter `amount` is equal to `0`.

Consequences of the execution of `backWith`

The `backWith` action could be seen with a lower likelihood because it's executed by an authenticated user that needs to have the `CONFIGURATOR` role.

Let's make this assumption

- `Gsm.priceStrategy` has `PRICE_RATIO` equal to `1` and no fees
- Users have sold and bought the asset and `_currentExposure` is equal to `1000e18` with an amount of `GHO` minted equal to `1000e18`
- Mistakenly, the `Gsm.priceStrategy` is updated to a price strategy with `PRICE_RATIO` equal to `0`

With this scenario, the `CONFIGURATOR` can execute `backWith` infinite times with an infinite `amount` if the `asset` passed as parameter is the `Gsm` 's underlying.

Let's simulate `backWith(underlying, 100_000e18)`

`(, uint256 ghoMinted) = IGhoToken(GHO_TOKEN).getFacilitatorBucket(address(this));` would return `ghoMinted == 1000e18`

`(, uint256 deficit) = _getCurrentBacking(ghoMinted);` would return `deficit` equal to `1000e18` because internally `_getCurrentBacking` would enter the `else` branch given that `ghoToBack` is equal to `0` because `getAssetPriceInGho` returns `0` (because `PRICE_RATIO` is `0`)

`uint256 ghoToBack = (asset == GHO_TOKEN) ? amount : IGsmPriceStrategy(_priceStrategy).getAssetPriceInGho(amount);`

will set `ghoToBack == 0` because `asset == underlying` and `getAssetPriceInGho` returns `0` (because `PRICE_RATIO` is `0`)

At this point, we enter the `else` branch and execute

```
_currentExposure += 100_000e18;
IERC20(UNDERLYING_ASSET).safeTransferFrom(msg.sender, address(this), 100_000e18);
```

Recommendations

Aave should add a sanity check on the `priceRatio` input parameter of the `constructor` of both the `FixedPriceStrategy` and `FixedPriceStrategy4626` contracts.

The price strategy contract should **not** be initialized with a `PRICE_RATIO` equal to `0` to avoid the issues described above.

In addition to that, it could make sense to also implement such a check directly in the `Gsm._updatePriceStrategy` and revert if `IGsmPriceStrategy(priceStrategy).PRICE_RATIO()` is equal to zero.

Discussion

Security Researcher

The recommendations have been implemented in the commit 7c03c52c0a271120837cd8e2c2750fe12b9f00c4 (included in the PR 369)

[L-05] Consider avoiding executing the `safeTransfer` in `seize()` if `underlyingBalance` is equal to zero to avoid

reverts in case of "weird" `ERC20` implementations

Context

- Gsm.sol#L210

Severity

Impact: Low The impact would normally be higher, but the Governance could (and probably have already) stop the swap operation by freezing the `Gsm` . All the other executable functions (when the contract is not seized) are function executable only by authed users.

Likelihood: Low The `UNDERLYING_ASSET` asset used by the `Gsm` is an asset that would be "approved" and verified by the Governance before the deployment of the `Gsm` . We can assume that the DAO and all the parties involved have done their due diligence on the `ERC20` and all the behavior that differs from the normal ERC20 standard.

Description

The `seize()` function is a function that will be executed only when `Gsm` is an extreme scenario and is a last-resort solution to a critical problem.

This function should be executable no-matter what the state of the contract and should never revert under normal circumstances (unless `_ghoTreasury` is equal to `address(0)` but this scenario has already been reported in another issue).

We need to anyway take in consideration that some `ERC20` tokens could use an implementation that does not adhere to the `ERC20` standard that allows the transfer of `0` tokens.

To avoid a possible revert caused by these "weird" ERC20 tokens, the `seize()` function should execute the `safeTransfer` operation only if the `underlyingBalance` of the contract is greater than zero.

Recommendations

Aave should consider performing the `IERC20(UNDERLYING_ASSET).safeTransfer` operation only if the balance of the contract is greater than zero, to avoid any possible revert caused by a "weird" `ERC20` implementation used by `UNDERLYING_ASSET` .

```
function seize() external notSeized onlyRole(LIQUIDATOR_ROLE) {
    _isSeized = true;

    (, uint256 ghoMinted) = IGhoToken(GHO_TOKEN).getFacilitatorBucket(address(this));
    uint256 underlyingBalance = IERC20(UNDERLYING_ASSET).balanceOf(address(this));
-    IERC20(UNDERLYING_ASSET).safeTransfer(_ghoTreasury, underlyingBalance);
+    if( underlyingBalance > 0 ) {
+       IERC20(UNDERLYING_ASSET).safeTransfer(_ghoTreasury, underlyingBalance);
+    }
    emit Seized(msg.sender, _ghoTreasury, underlyingBalance, ghoMinted);
}
```

Discussion

Security Researcher

The recommendations have been implemented in the commit 7c03c52c0a271120837cd8e2c2750fe12b9f00c4 (included in the PR 369)

[L-06] `Gsm` could be initialized with `admin` as `address(0)` preventing the configurator to operate and setup additional role members like Liquidator and Freezer

Context

- Gsm.sol#L106-L107
- AccessControl.sol#L228-L233

Severity

Impact: Low When the `Gsm` is deployed, there are no funds in the contract. I assume that the contract simply won't be enabled as a facilitator in the `GHO` contract, and it won't be able to mint/burn `GHO`.

If an incorrectly initialized `Gsm` is enabled as facilitator, the impact and severity should be raised.

Likelihood: Low The deployment, initialization (and later enabled as a `GHO` facilitator) should be executed as a Governance proposal. I assume that all these steps are carefully reviewed by the DAO, community and all security party involved that would prevent such misconfiguration.

Description

The current implementation of `Gsm.initialize` allows the caller to specify an arbitrary `admin` address that is configures as the first `ADMIN` and `CONFIGURATOR` of the `Gsm`.

```
  function initialize(
      address admin,
      address ghoTreasury,
      address priceStrategy,
      uint128 exposureCap
    ) external initializer {
  >     _grantRole(DEFAULT_ADMIN_ROLE, admin);
  >     _grantRole(CONFIGURATOR_ROLE, admin);

      // ... additional function code
  }
```

The OpenZeppelin implementation of the `internal` function `_grantRole` called during the `Gsm.initialize` does not perform any sanity check on both the `role` and `user` input parameter, allowing the caller to set up the admin as `address(0)`.

An important role of the `ADMIN` is also to add users/wallets to the `TOKEN_RESCUER_ROLE`, `SWAP_FREEZER_ROLE` and `LIQUIDATOR_ROLE`.

Without an admin (and as a consequence, a `CONFIGURATOR`) correctly setupped all the "authed" functions that administer the `Gsm` cannot be called.

If such `Gsm` is accidentally enabled as a valid `GHO` facilitator, the consequences would be much worse.

Recommendations

Consider adding a sanity check on the `admin` input parameter and revert if it is equal to `address(0)`. For completeness, consider adding also these basic sanity checks to the `Gsm.constructor`.

Note: the sanity check on the input parameter `ghoTreasury` has already been handled in a separate issue.

Discussion

Security Researcher

The recommendations have been implemented in the commit 7c03c52c0a271120837cd8e2c2750fe12b9f00c4 (included in the PR 369) The commit

a5ac4d25a150ec49bfc8a4f54061d5c1e4d0b016 (included in the PR 369) implements the recommended sanity checks for the `Gsm.constructor`

[L-07] `Gsm4626._cumulateYieldInGho` should always mint as many `GHO` as possible of the yield instead of skipping the fee accrual

Context

- Gsm4626.sol#L62-L66

Severity

Impact: Low `Gsm4626` won't be able to mint fees derived from the vault's yield until `GHO` (that manages the facilitators) increases the `Gsm4626` bucket capacity or some users buy assets from the `Gsm` to burn the amount of `GHO` needed by `_cumulateYieldInGho`

Likelihood: Low Unless the bucket level is very low or the user's fill up the capacity by selling the assets, this should not happen often. There are anyway actions that could be taken to "unlock" the situation.

Description

The `Gsm4626._cumulateYieldInGho` is an internal function that mints `GHO` for the treasury if there is an excess of `GHO` for the `Gsm` .

Those `GHO` can be minted as fees only if the **whole amount** can be minted by the facilitator, given the amount of `GHO` already minted and the current facilitator's capacity.

The current implementation `_cumulateYieldInGho` won't perform such action if the whole amount of fee cannot be minted, even if the facilitator has still some capacity left.

Test

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import './TestGhoBase.t.sol';

contract SmartGsm4626 is Gsm {
  constructor(address ghoToken, address underlyingAsset) Gsm(ghoToken, underlyingAsset
    // Intentionally left blank
```

```
  }

  function updatePriceStrategy(address priceStrategy) public override {
    // Cumulates yield based on the current price strategy before updating
    // Note that the accrual can be skipped in case the capacity is maxed out
    // A temporary increase of the bucket capacity facilitates the fee accrual
    _cumulateYieldInGho();

    super.updatePriceStrategy(priceStrategy);
  }

  function distributeFeesToTreasury() public override {
    _cumulateYieldInGho();
    super.distributeFeesToTreasury();
  }

  function _beforeBuyAsset(address, uint128, address) internal override {
    _cumulateYieldInGho();
  }

  function _beforeSellAsset(address, uint128, address) internal override {}

  function _cumulateYieldInGho() internal {
    (uint256 ghoCapacity, uint256 ghoLevel) = IGhoToken(GHO_TOKEN).getFacilitatorBucke
      address(this)
    );
    uint256 ghoAvailableToMint = ghoCapacity > ghoLevel ? ghoCapacity - ghoLevel : 0;
    (uint256 ghoExcess, ) = _getCurrentBacking(ghoLevel);

    // @audit - if we can't mint as much as we would like, mint the max possible amoun
    if (ghoAvailableToMint < ghoExcess) {
      ghoExcess = ghoAvailableToMint;
    }

    if (ghoExcess > 0) {
      _accruedFees += uint128(ghoExcess);
      IGhoToken(GHO_TOKEN).mint(address(this), ghoExcess);
    }
  }
}

contract SGsm4626YieldAsMuchAsPossibleTest is TestGhoBase {
  using PercentageMath for uint256;
  using PercentageMath for uint128;

  function testCANNOTDistributeYieldBecauseOfCapacity() public {
    // The Gsm4626 is a GHO facilitator that can mint UP TO (max) 200 GHO
    IGhoToken(GHO_TOKEN).setFacilitatorBucketCapacity(address(GHO_GSM_4626), 200e18);

    // Gsm4626 with 1:1 price ratio and zero fees
    GHO_GSM_4626.updateFeeStrategy(address(0));

    // Mint 100 USDC worth of shares from the vault
```

```
    _mintShares(USDC_4626_TOKEN, USDC_TOKEN, ALICE, 100e6);

    // sell those shares to acquire GHO
    vm.startPrank(ALICE);
    USDC_4626_TOKEN.approve(address(GHO_GSM_4626), DEFAULT_GSM_USDC_AMOUNT);
    GHO_GSM_4626.sellAsset(DEFAULT_GSM_USDC_AMOUNT, ALICE);
    vm.stopPrank();

    (uint256 excess, uint256 deficit) = GHO_GSM_4626.getCurrentBacking();
    assertEq(excess, 0);
    assertEq(deficit, 0);

    // Let's donate 110 USDC to the vault to generate some yield
    uint256 usdcDontedToVault = 110e6;
    vm.startPrank(FAUCET);
    USDC_TOKEN.mint(FAUCET, usdcDontedToVault);
    USDC_TOKEN.transfer(address(USDC_4626_TOKEN), usdcDontedToVault);
    vm.stopPrank();

    (excess, deficit) = GHO_GSM_4626.getCurrentBacking();
    assertEq(excess, 110e18);
    assertEq(deficit, 0);

    // excess now is 110 GHO and those 110 GHO could be minted as "yield fees" to be g
    // but if you try to execute `updatePriceStrategy` that triggers `_cumulateYieldIn
    // nothing happens because there's not enough capacity for the facilitator to mint
    GHO_GSM_4626.updatePriceStrategy(GHO_GSM_4626.getPriceStrategy());

    // excess will be the same as before
    // and no fees has been accumulated into `_accruedFees`
    (excess, deficit) = GHO_GSM_4626.getCurrentBacking();
    assertEq(excess, 110e18);
    assertEq(deficit, 0);
    assertEq(GHO_GSM_4626.getAccruedFees(), 0);
  }

  function testCanDistributeYield() public {
    // Let's do the same thing but by using the "SmartGsm4626" contract,
    // a modified version of the Gsm4626 that simply check if the excees of GHO is abo
    // and if that's true we mint just what we can instead of skipping it!

    SmartGsm4626 smartGsm4626 = new SmartGsm4626(address(GHO_TOKEN), address(USDC_4626
    smartGsm4626.initialize(
      address(this),
      TREASURY,
      address(GHO_GSM_4626_FIXED_PRICE_STRATEGY),
      DEFAULT_GSM_USDC_EXPOSURE
    );
    smartGsm4626.updateFeeStrategy(address(0));
    smartGsm4626.grantRole(GSM_LIQUIDATOR_ROLE, address(GHO_GSM_LAST_RESORT_LIQUIDATOR
    smartGsm4626.grantRole(GSM_SWAP_FREEZER_ROLE, address(GHO_GSM_SWAP_FREEZER));

    // set the GSM bucket limit as the same as the previus test
```

```
    IGhoToken(GHO_TOKEN).addFacilitator(address(smartGsm4626), 'GSM 4626 Facilitator',

    // perform the same action as before
    // mint 100 USDC to ALICE
    // deposit them to the Vault
    // sell the shares to GSM
    vm.startPrank(FAUCET);
    USDC_TOKEN.mint(FAUCET, DEFAULT_GSM_USDC_AMOUNT);
    USDC_TOKEN.approve(address(USDC_4626_TOKEN), DEFAULT_GSM_USDC_AMOUNT);
    USDC_4626_TOKEN.deposit(DEFAULT_GSM_USDC_AMOUNT, ALICE);
    vm.stopPrank();

    // sell those shares to acquire GHO
    vm.startPrank(ALICE);
    USDC_4626_TOKEN.approve(address(smartGsm4626), DEFAULT_GSM_USDC_AMOUNT);
    smartGsm4626.sellAsset(DEFAULT_GSM_USDC_AMOUNT, ALICE);
    vm.stopPrank();

    (uint256 excess, uint256 deficit) = smartGsm4626.getCurrentBacking();
    assertEq(excess, 0);
    assertEq(deficit, 0);

    // donate 110 USD to the vault
    // Let's donate 110 USDC to the vault to generate some yield
    uint256 usdcDontedToVault = 110e6;
    vm.startPrank(FAUCET);
    USDC_TOKEN.mint(FAUCET, usdcDontedToVault);
    USDC_TOKEN.transfer(address(USDC_4626_TOKEN), usdcDontedToVault);
    vm.stopPrank();

    (excess, deficit) = smartGsm4626.getCurrentBacking();
    assertEq(excess, 110e18);
    assertEq(deficit, 0);

    // and try to trigger the `updatePriceStrategy` to see if fees now are distributed
    smartGsm4626.updatePriceStrategy(smartGsm4626.getPriceStrategy());

    // we were able to mint 100 GHO of the 110 GHO of the excess (because we are limit
    // the other amount of GHO will be minted when possible
    (excess, deficit) = smartGsm4626.getCurrentBacking();
    assertEq(excess, 10e18);
    assertEq(deficit, 0);
    assertEq(smartGsm4626.getAccruedFees(), 100e18);

    // we have maxed out what we could have minted to fill up all the Gsm excess
    (uint256 ghoCapacity, uint256 ghoLevel) = IGhoToken(GHO_TOKEN).getFacilitatorBucke
      address(smartGsm4626)
    );

    assertEq(ghoCapacity, ghoLevel);
  }
}
```

Recommendations

Aave should modify the logic of the `_cumulateYieldInGho` function, allowing the `Gsm4626` to mint as much `GHO` available to be minted instead of skipping the whole executing.

Here's an example of a possible solution

```
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 (ghoAvailableToMint < ghoExcess) {
+    ghoExcess = ghoAvailableToMint;
+  }

-  if (ghoAvailableToMint >= ghoExcess && ghoExcess > 0) {
+  if (ghoExcess > 0) {
    _accruedFees += uint128(ghoExcess);
    IGhoToken(GHO_TOKEN).mint(address(this), ghoExcess);
  }
}
```

Discussion

Security Researcher

The recommendations have been implemented in the commit 7c03c52c0a271120837cd8e2c2750fe12b9f00c4 (included in the PR 369)

[L-08] Contracts are using a floating pragma declaration

Context

- FixedFeeStrategy.sol#L2
- GsmRegistry.sol#L2
- SampleLiquidator.sol#L2
- SampleSwapFreezer.sol#L2
- FixedPriceStrategy.sol#L2

- FixedPriceStrategy4626.sol#L2
- Gsm.sol#L2
- Gsm4626.sol

Severity

Impact: Low There are two possible problematic impacts

1. The contracts is compiled with a version of solidity that contains a known bug (or unknown bug currently) and the codebase contains code vulnerable to such an exploit
2. The contract is compiled with a version of solidity that contains features that are not supported by other chains to which these contracts will be deployed. An example would be the EIP-3855 `PUSH0` instruction that could be not supported right now on different chains.

Likelihood: Low The review and deployment of these contracts should be executed as a Governance proposal. I assume that all these steps are carefully reviewed by the DAO, community and all security party involved that would prevent such misconfiguration.

Description

The current implementation of the contracts listed in the Context sections are using a "floating pragma" declaration. By doing this, Aave is not ensuring that the contract will be built and deployed with the same Solidity version that has been used to test it.

Recommendations

Aave should lock the pragma version of the contracts to be sure that the contract will be built and deployed with a specific, known and tested version. Aave should also take in consideration building those contracts with a Solidity version that is compatible with the chain to which those contracts will be deployed to.

Discussion

Security Researcher

The PR commit 7c03c52c0a271120837cd8e2c2750fe12b9f00c4 (included in the PR 369) updates the floating pragma from `pragma solidity ^0.8.0;` to `pragma solidity ^0.8.10;` for the `Contract` files.

Aave has decided to not enforce a "fixed" pragma version because the solidity version (used for the build and deployment) will be enforced by the `foundry.toml` configuration.

It's important to note that external users who decide to use this codebase and do not follow the same approach of Aave could end up building and deploying those contracts with a Solidity version different from `0.8.10`

[L-09] `CONFIGURATOR` can execute `Gsm.backWith` after that the `Gsm` has been seized, locking the `underlying` used to back it with

Context

- Gsm.sol#L229-L253
- Gsm.sol#L183-L186

Severity

Original Impact: High The `underlying` sent by the `CONFIGURATOR` to back the `Gsm` cannot be rescued **Updated Impact:** Low The `underlying` sent by the `CONFIGURATOR` to back the `Gsm` cannot be rescued unless Aave upgrades the contract implementing a new function that allows an authed user to rescue those tokens.

Likelihood: Low The `CONFIGURATOR` should be aware that the `Gsm` has been seized and should not back the `Gsm` with additional `underlying` or `GHO`

Description

The current implementation of `Gsm.backWith` allows a `CONFIGURATOR` to transfer `GHO` or `UNDERLYING` when the `Gsm` is in deficit.

The problem is that when the `Gsm` has been seized, the `backWith` operation should not be allowed to be executed to avoid locking the `underlying` that has been sent within the `backWith` operation.

Those additional `underlying` tokens can't be rescued because

1. Liquidator cannot call again `seize` (it would revert when `Gsm` is in `seized` state)
2. The asset can't be sold/bought because all the swap operations reverts because the `Gsm` is in `seized` state
3. The `rescqueTokens` reverts because `_currentExposure` is greater than the `Gsm` balance of the underlying. The `seize` operation has removed all the underlying without changing the value

of `_currentExposure` and the `backWith` operation (backing the `Gsm` with additional underlying) has **increased** the value of `_currentExposure`

Test

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import './TestGhoBase.t.sol';
import {ERC20 as SolmateERC20} from '../contracts/gho/ERC20.sol';

contract MintableSolmateERC20 is SolmateERC20 {
  constructor(
    string memory _name,
    string memory _symbol,
    uint8 _decimals
  ) SolmateERC20(_name, _symbol, _decimals) {}

  function mint(address account, uint256 amount) external {
    require(amount > 0, 'INVALID_MINT_AMOUNT');
    _mint(account, amount);
  }
}

contract SGsmBackWithTest is TestGhoBase {
  using PercentageMath for uint256;
  using PercentageMath for uint128;

  address internal gsmSignerAddr;
  uint256 internal gsmSignerKey;

  function setUp() public {
    (gsmSignerAddr, gsmSignerKey) = makeAddrAndKey('gsmSigner');
  }

  function testBackWithGHOAfterSeize() public {
    // Alice sell 100 underlying
    // Price from 1:1 GHO goes to 1:0,5 GHO
    // Aave decide to seize the gsm
    (Gsm gsm, MintableSolmateERC20 solmateUnderlying) = setupTest(0.5e18);

    // Because the price of underlying has dropped there's a deficit and the GSM CONFI
    // could call the `backWith` (even if it shouldn't given that the contract has bee
    (uint256 excess, uint256 deficit) = gsm.getCurrentBacking();

    // There should be a deficit of 50 GHO
    assertEq(excess, 0);
    assertEq(deficit, 50e18);

    // The CONFIGURATOR calls backWith to back the Gsm by burning 50 GHO
    // After the `backWith` operation there's no more deficit and the Gsm from 100 GHO
```

```
    address configurator = address(this);
    backWithGHO(gsm, configurator, 50e18);

    (excess, deficit) = gsm.getCurrentBacking();
    assertEq(excess, 0);
    assertEq(deficit, 0);

    // From 100 GHO we now only have 50 GHO
    (, uint256 ghoMinted) = IGhoToken(GHO_TOKEN).getFacilitatorBucket(address(gsm));
    assertEq(ghoMinted, 50e18);

    // It's not a big deal because the CONFIGURATOR has burned half of the GHO token
    // that should have been burned by the Liquidator during `burnAfterSeize`

    ghoFaucet(address(GHO_GSM_LAST_RESORT_LIQUIDATOR), 50e18);
    vm.startPrank(address(GHO_GSM_LAST_RESORT_LIQUIDATOR));
    GHO_TOKEN.approve(address(gsm), 50e18);
    gsm.burnAfterSeize(50e18);

    (, ghoMinted) = IGhoToken(GHO_TOKEN).getFacilitatorBucket(address(gsm));
    assertEq(ghoMinted, 0);

    // One problem that I could see is that the Gsm is still seen as in deficit becaus
    // has not updated the `_currentExposure` (even if it has removed the whole underl
    // and there's no more GHO minted by the Gsm

    (excess, deficit) = gsm.getCurrentBacking();
    assertEq(excess, 50e18);
    assertEq(deficit, 0);
  }

  function testBackWithUnderlyingAfterSeize() public {
    // Alice sell 100 underlying
    // Price from 1:1 GHO goes to 1:0,5 GHO
    // Aave decide to seize the gsm
    (Gsm gsm, MintableSolmateERC20 solmateUnderlying) = setupTest(0.5e18);

    // Because the price of underlying has dropped there's a deficit and the GSM CONFI
    // could call the `backWith` (even if it shouldn't given that the contract has bee
    (uint256 excess, uint256 deficit) = gsm.getCurrentBacking();

    // There should be a deficit of 50 GHO
    assertEq(excess, 0);
    assertEq(deficit, 50e18);

    // The CONFIGURATOR calls backWith to back the Gsm with missing underlying
    // After the `backWith` operation there's no more deficit and the Gsm from 0 under
    address configurator = address(this);
    backWithUnderlying(gsm, configurator, 100e18);

    (excess, deficit) = gsm.getCurrentBacking();
    assertEq(excess, 0);
    assertEq(deficit, 0);
```

```
    assertEq(solmateUnderlying.balanceOf(address(gsm)), 100e18);

    // when the Gsm is seized
    // - seize() can't be called anymore
    // - sell/buy asset can't be called anymore
    // - rescueTokens() can be called but if `asset == underlying` it will revert beca
    gsm.grantRole(GSM_TOKEN_RESCUER_ROLE, address(this));

    // try to rescque 1 wei of the underlying. it will revert because of an underflow
    vm.expectRevert(stdError.arithmeticError);
    gsm.rescueTokens(address(solmateUnderlying), TREASURY, 1);

    // the current exposure is 200 (100 sold by ALICE, 100 backed from configutator)
    // even if during the seize, those 100 from ALICE has been transferred to the trea
    assertEq(gsm.getAvailableLiquidity(), 200e18);

    // note liquidity == _currentExposure
    assertLt(solmateUnderlying.balanceOf(address(gsm)), gsm.getAvailableLiquidity());

    // the end result is that those tokens can't be rescqued anymore
  }

  function setupTest(uint256 newFixedPrice) internal returns (Gsm, MintableSolmateERC2
    // Create a token with 18 decimals
    MintableSolmateERC20 solmateUnderlying = new MintableSolmateERC20(
      'GsmUnderlying',
      'GsmUnderlying',
      18
    );

    // Deploy a Gsm with a 18 decimal underlying
    // The price strategy now is 1:1 -> 1 underlying === 1 GHO
    Gsm gsm = setUpGsm(TREASURY, address(solmateUnderlying), 100_000_000e18, 100_000_0

    // remove the price strategy just to make things easier
    gsm.updateFeeStrategy(address(0));

    // Alice sell 100 token to get 100 GHO
    sellUnderlying(gsm, ALICE, 100e18);

    // Alice should have received 100 GHO
    // Gsm should have 100 underlying
    assertEq(GHO_TOKEN.balanceOf(address(ALICE)), 100e18);
    assertEq(solmateUnderlying.balanceOf(address(gsm)), 100e18);

    // The price strategy has not changed so there should be no excess or deficit
    (uint256 excess, uint256 deficit) = gsm.getCurrentBacking();
    assertEq(excess, 0);
    assertEq(deficit, 0);

    // Update the fixed price based on the new underlying market value
    FixedPriceStrategy newFixedPriceStrategy = new FixedPriceStrategy(
      newFixedPrice,
```

```
    address(solmateUnderlying),
    18
  );
  gsm.updatePriceStrategy(address(newFixedPriceStrategy));

  // Aave decide to seize and stop the operation
  vm.prank(address(GHO_GSM_LAST_RESORT_LIQUIDATOR));
  gsm.seize();
  assertEq(solmateUnderlying.balanceOf(address(gsm)), 0);

  return (gsm, solmateUnderlying);
}

////////////////////////////////////////////////////
/////// Utility functions
////////////////////////////////////////////////////

function setUpGsm(
  address treasury,
  address underlying,
  uint128 exposureCap,
  uint128 facilitatorCapacity
) internal returns (Gsm) {
  FixedPriceStrategy fixedPriceStrategy = new FixedPriceStrategy(
    DEFAULT_FIXED_PRICE,
    underlying,
    18
  );

  Gsm gsm = new Gsm(address(GHO_TOKEN), underlying);
  gsm.initialize(address(this), treasury, address(fixedPriceStrategy), exposureCap);

  // set the fee strategy
  gsm.updateFeeStrategy(address(GHO_GSM_FIXED_FEE_STRATEGY));

  // setup roles
  gsm.grantRole(GSM_LIQUIDATOR_ROLE, address(GHO_GSM_LAST_RESORT_LIQUIDATOR));
  gsm.grantRole(GSM_SWAP_FREEZER_ROLE, address(GHO_GSM_SWAP_FREEZER));

  // setup the new gsm as a facilitator
  IGhoToken(address(GHO_TOKEN)).addFacilitator(
    address(gsm),
    'GSM Facilitator Tester',
    facilitatorCapacity
  );

  return gsm;
}

function sellUnderlying(Gsm gsm, address seller, uint256 amountToSell) internal {
  address gsmUnderlying = gsm.UNDERLYING_ASSET();

  vm.prank(FAUCET);
```

```
    MintableSolmateERC20(gsmUnderlying).mint(seller, amountToSell);

    vm.startPrank(seller);
    MintableSolmateERC20(gsmUnderlying).approve(address(gsm), amountToSell);

    gsm.sellAsset(uint128(amountToSell), seller);
    vm.stopPrank();
  }

  function backWithUnderlying(Gsm gsm, address backer, uint256 amountToBack) internal
    address gsmUnderlying = gsm.UNDERLYING_ASSET();

    vm.prank(FAUCET);
    MintableSolmateERC20(gsmUnderlying).mint(backer, amountToBack);
    vm.startPrank(backer);
    MintableSolmateERC20(gsmUnderlying).approve(address(gsm), amountToBack);

    gsm.backWith(address(gsmUnderlying), uint128(amountToBack));
    vm.stopPrank();
  }

  function backWithGHO(Gsm gsm, address backer, uint256 amountToBack) internal {
    ghoFaucet(backer, amountToBack);
    vm.startPrank(backer);
    GHO_TOKEN.approve(address(gsm), amountToBack);
    gsm.backWith(address(GHO_TOKEN), uint128(amountToBack));
    vm.stopPrank();
  }
}
```

Recommendations

Aave should allow the execution of the `backWith` operation **only** when the `Gsm` is **not** in the `seized` state.

```
/// @inheritdoc IGsm
function backWith(address asset, uint128 amount)
    external
+   notSeized
    onlyRole(CONFIGURATOR_ROLE) {
    // function's code
}
```

Discussion

Security Researcher

The recommendations have been implemented correctly in the commit 7c03c52c0a271120837cd8e2c2750fe12b9f00c4 (included in the PR 369) and `backWith` cannot be executed once the `Gsm` has been seized.

Aave

We consider this one low severity, given that `backWith` is an authenticated function and contract is upgradeable anyway.

Security Researcher

The Impact of the issue has been lowered to **Low** because the contract can be upgraded and the funds recovered.

[L-10] Some "gifted" underlying will be locked forever in `Gsm` if gifted after `seize()`

Context

- Gsm.sol#L184-L185

Severity

Original Impact: High Part of the "gifted" underlying tokens will be locked in the contract forever
Updated Impact: Low Part of the "gifted" underlying tokens will be locked in the contract forever unless Aave upgrades the contract implementing a new function that allows an authed user to rescue those tokens.

Likelihood: Low The `Gsm` must be seized and someone must "gift" underlying to the `Gsm`

Description

When a Liquidator seizes a `Gsm` it will transfer all the underlying to the Treasury without updating the `_currentExposure` state variable.

After a seize event, users can't swap the asset (buy/sell) and the `seize` function can't be called anymore.

The only way to be able to rescue the underlying is by calling the `rescueTokens` function that limits the amount of rescuable underlying to `IERC20(underlying).balanceOf(address(this)) - _currentExposure`.

Let's assume that:

- `Gsm` has been seized, all the underlying has been transferred to the treasury. This means that `IERC20(underlying).balanceOf(address(this))` returns `0`.
- `_currentExposure` is `100e18` (100 tokens)
- Someone accidentally sends `300e18` underlying to the seized `Gsm`

The `TOKEN_RESCUER` will be able to only rescue a max of `300e18 - 100e18 = 200e18` tokens from the `Gsm`. The remaining amount of 100 underlying tokens will always remain locked inside the `Gsm`. If the Rescuer tries to recover the remaining amount, the `rescueTokens` function will revert with the error `INSUFFICIENT_EXOGENOUS_ASSET_TO_RESCUE`.

Test

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import './TestGhoBase.t.sol';
import {ERC20 as SolmateERC20} from '../contracts/gho/ERC20.sol';

contract MintableSolmateERC20 is SolmateERC20 {
  constructor(
    string memory _name,
    string memory _symbol,
    uint8 _decimals
  ) SolmateERC20(_name, _symbol, _decimals) {}

  function mint(address account, uint256 amount) external {
    require(amount > 0, 'INVALID_MINT_AMOUNT');
    _mint(account, amount);
  }
}

contract SGiftedTokenLockedAfterSeizeTest is TestGhoBase {
  using PercentageMath for uint256;
  using PercentageMath for uint128;

  address internal gsmSignerAddr;
  uint256 internal gsmSignerKey;

  function setUp() public {
    (gsmSignerAddr, gsmSignerKey) = makeAddrAndKey('gsmSigner');
  }

  function testGiftedTokenAfterSeizeAreLocked() public {
    // Alice sell 100 underlying
    // Price from 1:1 GHO goes to 1:0,5 GHO
    // Aave decide to seize the gsm
    (Gsm gsm, MintableSolmateERC20 solmateUnderlying) = setupTest(0.5e18);
```

```
  // Because the price of underlying has dropped there's a deficit and the GSM CONFI
  // could call the `backWith` (even if it shouldn't given that the contract has bee
  (uint256 excess, uint256 deficit) = gsm.getCurrentBacking();

  // There should be a deficit of 50 GHO
  assertEq(excess, 0);
  assertEq(deficit, 50e18);

  // someone by mistakes donates 100 underlying to the Gsm
  vm.prank(FAUCET);
  MintableSolmateERC20(solmateUnderlying).mint(BOB, 300e18);
  vm.prank(BOB);
  MintableSolmateERC20(solmateUnderlying).transfer(address(gsm), 300e18);

  // At this point only 200 of those underlying can be rescqued because
  // `_currentExposure` is still equal to 100e18
  // and `rescqueTokens` allows to only restore until balance-_currentExposure do no
  gsm.grantRole(GSM_TOKEN_RESCUER_ROLE, address(this));

  // rescque 200e18 of underlying
  gsm.rescueTokens(address(solmateUnderlying), TREASURY, 200e18);

  // if you try to rescque just 1 wei more it will revert because of `INSUFFICIENT_E
  vm.expectRevert(bytes('INSUFFICIENT_EXOGENOUS_ASSET_TO_RESCUE'));
  gsm.rescueTokens(address(solmateUnderlying), TREASURY, 1);
}

function setupTest(uint256 newFixedPrice) internal returns (Gsm, MintableSolmateERC2
  // Create a token with 18 decimals
  MintableSolmateERC20 solmateUnderlying = new MintableSolmateERC20(
    'GsmUnderlying',
    'GsmUnderlying',
    18
  );

  // Deploy a Gsm with a 18 decimal underlying
  // The price strategy now is 1:1 -> 1 underlying === 1 GHO
  Gsm gsm = setUpGsm(TREASURY, address(solmateUnderlying), 100_000_000e18, 100_000_0

  // remove the price strategy just to make things easier
  gsm.updateFeeStrategy(address(0));

  // Alice sell 100 token to get 100 GHO
  sellUnderlying(gsm, ALICE, 100e18);

  // Alice should have received 100 GHO
  // Gsm should have 100 underlying
  assertEq(GHO_TOKEN.balanceOf(address(ALICE)), 100e18);
  assertEq(solmateUnderlying.balanceOf(address(gsm)), 100e18);

  // The price strategy has not changed so there should be no excess or deficit
  (uint256 excess, uint256 deficit) = gsm.getCurrentBacking();
```

```
    assertEq(excess, 0);
    assertEq(deficit, 0);

    // Update the fixed price based on the new underlying market value
    FixedPriceStrategy newFixedPriceStrategy = new FixedPriceStrategy(
      newFixedPrice,
      address(solmateUnderlying),
      18
    );
    gsm.updatePriceStrategy(address(newFixedPriceStrategy));

    // Aave decide to seize and stop the operation
    vm.prank(address(GHO_GSM_LAST_RESORT_LIQUIDATOR));
    gsm.seize();
    assertEq(solmateUnderlying.balanceOf(address(gsm)), 0);

    return (gsm, solmateUnderlying);
  }

  ////////////////////////////////////////////////////
  /////// Utility functions
  ////////////////////////////////////////////////////

  function setUpGsm(
    address treasury,
    address underlying,
    uint128 exposureCap,
    uint128 facilitatorCapacity
  ) internal returns (Gsm) {
    FixedPriceStrategy fixedPriceStrategy = new FixedPriceStrategy(
      DEFAULT_FIXED_PRICE,
      underlying,
      18
    );

    Gsm gsm = new Gsm(address(GHO_TOKEN), underlying);
    gsm.initialize(address(this), treasury, address(fixedPriceStrategy), exposureCap);

    // set the fee strategy
    gsm.updateFeeStrategy(address(GHO_GSM_FIXED_FEE_STRATEGY));

    // setup roles
    gsm.grantRole(GSM_LIQUIDATOR_ROLE, address(GHO_GSM_LAST_RESORT_LIQUIDATOR));
    gsm.grantRole(GSM_SWAP_FREEZER_ROLE, address(GHO_GSM_SWAP_FREEZER));

    // setup the new gsm as a facilitator
    IGhoToken(address(GHO_TOKEN)).addFacilitator(
      address(gsm),
      'GSM Facilitator Tester',
      facilitatorCapacity
    );

    return gsm;
```

```
  }

  function sellUnderlying(Gsm gsm, address seller, uint256 amountToSell) internal {
    address gsmUnderlying = gsm.UNDERLYING_ASSET();

    vm.prank(FAUCET);
    MintableSolmateERC20(gsmUnderlying).mint(seller, amountToSell);

    vm.startPrank(seller);
    MintableSolmateERC20(gsmUnderlying).approve(address(gsm), amountToSell);

    gsm.sellAsset(uint128(amountToSell), seller);
    vm.stopPrank();
  }

  function backWithUnderlying(Gsm gsm, address backer, uint256 amountToBack) internal
    address gsmUnderlying = gsm.UNDERLYING_ASSET();

    vm.prank(FAUCET);
    MintableSolmateERC20(gsmUnderlying).mint(backer, amountToBack);
    vm.startPrank(backer);
    MintableSolmateERC20(gsmUnderlying).approve(address(gsm), amountToBack);

    gsm.backWith(address(gsmUnderlying), uint128(amountToBack));
    vm.stopPrank();
  }

  function backWithGHO(Gsm gsm, address backer, uint256 amountToBack) internal {
    ghoFaucet(backer, amountToBack);
    vm.startPrank(backer);
    GHO_TOKEN.approve(address(gsm), amountToBack);
    gsm.backWith(address(GHO_TOKEN), uint128(amountToBack));
    vm.stopPrank();
  }
}
```

Recommendations

Aave should set the value of `_currentExposure` (the amount of underlying token owned by the `Gsm`) to `0` when the `seize()` function is executed.

By doing so, the Rescuer will be able to rescue the whole underlying amount after that the `Gsm` has been seized.

Discussion

Security Researcher

The recommendations have been implemented in the commit 7c03c52c0a271120837cd8e2c2750fe12b9f00c4 (included in the PR 369)

Aave

We consider this one low severity, given that rescue of tokens is not the main purpose of the contract.

Security Researcher

The Impact of the issue has been lowered to **Low** because the contract can be upgraded and the funds recovered.

[L-11] `Gsm.buyAsset` will stop working as soon as the `priceStrategy` is updated to a one with a higher (of just `1 wei`) `PRICE_RATIO`

Context

- Gsm.sol#L396-L409
- GhoToken.sol#L54 (where the revert happens)

Severity

Impact: Low User will not be able to buy the asset from the `Gsm`. The Impact is Low because the user is not going to lose funds and could use other services to exchange `GHO` for the `underlying`.

Likelihood: Low The depegs of a well known and liquid stable coin can be seen as a low likelihood event. The described issue will only happen if the GSM Configurator decides to update the priceStrategy of the GSM to a one with a higher `PRICE_RATIO`

Description

When the user tries to buy an `amount` of underlying, it will need to send `Y` amount of `GHO` for the purchase and some `ghoFee` depending on the `feeStrategy` used by the `Gsm`.

The `totalGHONeeded - ghoFee` transferred to the `Gsm` to make the purchase will be burned. When the `GHO.burn` is executed, the `GHO` contract executes the following function

```
function burn(uint256 amount) external {
    require(amount > 0, 'INVALID_BURN_AMOUNT');
```

```
    Facilitator storage f = _facilitators[msg.sender];
    uint256 currentBucketLevel = f.bucketLevel;
>>    uint256 newBucketLevel = currentBucketLevel - amount;
    f.bucketLevel = uint128(newBucketLevel);

    _burn(msg.sender, amount);

    emit FacilitatorBucketLevelUpdated(msg.sender, currentBucketLevel, newBucketLevel)
}
```

If the amount of `GHO` burned is higher compared to the amount of `GHO` minted by the facilitator (in this case, it's represented by the `f.bucketLevel` variable), the function will revert because of an underflow exception.

This is what happens when the `Gsm priceStrategy` changes to a `priceStrategy` with a higher `PRICE_RATIO` and the user ties to purchase of `underlying` enough big to trigger such underflow.

Let's make an example:

The `Gsm` is initialized with - `TokenA` as an underlying with `18 decimals` - `priceStrategy` with `PRICE_RATIO = 1e18` - `feeStrategy` with `buyFee` and `sellFee` equal to 10%

1. Alice sell `100e18` of `TokenA`
2. `Gsm` will **mint** `100 GHO`, `90 GHO` will be transferred to Alice, `10 GHO` remain in the contract (to be later distributed to the treasury)
3. The `TokenA` market value increases of `1 wei` and Aave decides to update the `Gsm` to a new `priceStrategy` with `PRICE_RATIO = 1e18 + 1`
4. Bob wants to purchase `100e18` of `TokenA` knowing that he will need to pay a little bit more of `GHO` because of the increase in price.
5. Bob prepare all the `GHO` needed to execute the transaction and call `Gsm.buyAsset(100e18)`
6. The `Gsm` calculates the amount of `GHO` needed by executing `(uint256 ghoSold, uint256 grossAmount, uint256 fee) = _calculateGhoAmountForBuyAsset(amount);`
7. By using the new `PRICE_RATIO` we know that the user will have to transfer a total of `110,00000000000000011 GHO`. `100,0000000000000001 GHO` will be burned by the protocol (in exchange for the `100e18` of `TokenA`) and `10,00000000000000001 GHO` will be taken as fee (fee is not burned, stays in the contract to be later transferred to the treasury)
8. At this point, `Gsm._buyAsset` tries to execute `IGhoToken(GHO_TOKEN).burn(100,0000000000000001);` that will revert because of underflow. `Gsm` had only minted `100e18 GHO` during sell operation done by Alice at the very beginning

Test

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
```

```
import './TestGhoBase.t.sol';
import {ERC20 as SolmateERC20} from '../contracts/gho/ERC20.sol';

contract MintableSolmateERC20 is SolmateERC20 {
  constructor(
    string memory _name,
    string memory _symbol,
    uint8 _decimals
  ) SolmateERC20(_name, _symbol, _decimals) {}

  function mint(address account, uint256 amount) external {
    require(amount > 0, 'INVALID_MINT_AMOUNT');
    _mint(account, amount);
  }
}

contract SGsmPriceIncreaseBreakBuyTokenTest is TestGhoBase {
  using PercentageMath for uint256;
  using PercentageMath for uint128;

  address internal gsmSignerAddr;
  uint256 internal gsmSignerKey;

  function setUp() public {
    (gsmSignerAddr, gsmSignerKey) = makeAddrAndKey('gsmSigner');
  }

  function testBuyRevertWhenPriceRatioIncrease() public {
    // Create a token with 18 decimals
    MintableSolmateERC20 solmateUnderlying = new MintableSolmateERC20(
      'GsmUnderlying',
      'GsmUnderlying',
      18
    );

    Gsm gsm = setUpGsm(TREASURY, address(solmateUnderlying), 100_000_000e18, 100_000_0

    // Price Strategy of 1:1
    uint256 startPriceRatio = 1e18;
    FixedPriceStrategy fixedPriceStrategy = new FixedPriceStrategy(
      startPriceRatio,
      address(solmateUnderlying),
      18
    );
    gsm.updatePriceStrategy(address(fixedPriceStrategy));

    // User 1 sell 100 token and will receive 90 GHO (100 GHO minted, 10 GHO of fee)
    uint256 amountToSell = 100e18;

    // Alice tries to sell 100 underlying
    vm.prank(FAUCET);
    solmateUnderlying.mint(ALICE, amountToSell);
```

```
    vm.startPrank(ALICE);
    solmateUnderlying.approve(address(gsm), amountToSell);

    gsm.sellAsset(uint128(amountToSell), ALICE);
    vm.stopPrank();

    assertEq(GHO_TOKEN.balanceOf(ALICE), 90e18);
    assertEq(GHO_TOKEN.balanceOf(address(gsm)), 10e18);
    assertEq(solmateUnderlying.balanceOf(address(gsm)), 100e18);

    // The PRICE_RATIO increase of 1 wei
    fixedPriceStrategy = new FixedPriceStrategy(
      startPriceRatio + 1,
      address(solmateUnderlying),
      18
    );
    gsm.updatePriceStrategy(address(fixedPriceStrategy));

    // BOB want to buy 100 token and will need to pay a little bit more compared to be
    uint256 buyAmount = 100e18;
    uint256 ghoForAssetAmount = fixedPriceStrategy.getAssetPriceInGho(buyAmount);
    uint256 buyFee = ghoForAssetAmount.percentMul(DEFAULT_GSM_BUY_FEE);

    ghoFaucet(BOB, ghoForAssetAmount + buyFee);
    vm.startPrank(BOB);
    GHO_TOKEN.approve(address(gsm), ghoForAssetAmount + buyFee);

    // the buyAsset operation reverts because `Gsm` tries to burn more token that have
    // To be precise the one that is reverting is GHO when it tries to remove the burn
    // and the operation reverts because of an underflow
    vm.expectRevert(stdError.arithmeticError);
    gsm.buyAsset(uint128(buyAmount), BOB);
    vm.stopPrank();
  }

  ////////////////////////////////////////////////////
  /////// Utility functions
  ////////////////////////////////////////////////////

  function setUpGsm(
    address treasury,
    address underlying,
    uint128 exposureCap,
    uint128 facilitatorCapacity
  ) internal returns (Gsm) {
    FixedPriceStrategy fixedPriceStrategy = new FixedPriceStrategy(
      DEFAULT_FIXED_PRICE,
      underlying,
      18
    );

    Gsm gsm = new Gsm(address(GHO_TOKEN), underlying);
```

```
      gsm.initialize(address(this), treasury, address(fixedPriceStrategy), exposureCap);

      // set the fee strategy
      gsm.updateFeeStrategy(address(GHO_GSM_FIXED_FEE_STRATEGY));

      // setup roles
      gsm.grantRole(GSM_LIQUIDATOR_ROLE, address(GHO_GSM_LAST_RESORT_LIQUIDATOR));
      gsm.grantRole(GSM_SWAP_FREEZER_ROLE, address(GHO_GSM_SWAP_FREEZER));

      // setup the new gsm as a facilitator
      IGhoToken(address(GHO_TOKEN)).addFacilitator(
        address(gsm),
        'GSM Facilitator Tester',
        facilitatorCapacity
      );

      return gsm;
    }
  }
```

Recommendations

With the current logic of `Gsm` and `GHO` there's not a direct way (`Gsm` functions to be executed) to handle this scenario that breaks the buy operation.

Aave should extensively detail how they plan to act during scenarios like this (price change) that are pretty common in DeFi.

Discussion

Aave

Acknowledged The FixedPriceStrategy is not designed to "float" with the prevailing exchange price, but is instead used to "fix" a rate of exchange to provide price stability. In the event that prices are outside of bounds deemed acceptable (i.e., by Risk Providers), the DAO can elect to freeze swaps. In the event a price is permanently dislocated, the next step would likely be to seize assets in the GSM and there would be a need for Governance to evaluate next steps. Arbitrage opportunities created as a result of using a fixed price ratio relative to small fluctuations in price are by design.

[I-01] Consider renaming `Gsm _ghoTreasury` as `_gsmTreasury` to be consistent with the new role of the treasury

Context

- Gsm.sol#L58

Description

Unlike in the previous iteration where the `_ghoTreasury` was only used to receive the `GHO _accruedFees`, the current implementation of the `Gsm` uses the `_ghoTreasury` as the receiver of both the fees and the seized `underlying` asset (transferred when the `seize()` function is executed).

To be consistent with the new role of the `_ghoTreasury`, Aave should consider renaming the state variable to something more specific and explicit like `_gsmTreasury`.

Recommendations

Aave should consider renaming the state variable to something more specific and explicit, like `_gsmTreasury`. If Aave decides to do so, it should perform all the following actions:

- rename the `_gsmTreasury`
- rename the `initialize` input parameter `ghoTreasury`
- rename the `updateGhoTreasury` function (and the input parameter) in both `Gsm` and `IGsm`
- rename the `GhoTreasuryUpdated` event

Discussion

Aave

Acknowledged Naming this variable after the GHO treasury is keeping it consistent with other GHO facilitators and the GHO facilitator interface.

[I-02] Consider to explicitly initializing the `feeStrategy` during the execution of `Gsm.initialize`

Context

- Gsm.sol#L100-L111

Description

The current implementation of `Gsm.initialize` does not initialize the `feeStrategy` of the `Gsm` . While this should not be a security concern (when `feeStrategy` is equal to `address(0)` it acts like if the user has to pay no fee), explicitly initializing the `feeStrategy` brings some benefit

1. The deployer avoids forgetting to properly set up a `feeStrategy` if it's needed by the `Gsm`
2. `dApps` or monitoring tools can monitor the correct initialization of the `Gsm` and an event emission for each state variable
3. After the executing the `initialize` function, all the state variables have been correctly initialized with an explicit value

Recommendations

Consider to explicitly initializing the `feeStrategy` during the execution of the `Gsm.initialize` function.

Discussion

Aave

Acknowledged Acknowledged, this is an item to ensure is appropriately configured when deploying a GSM which will go through DAO assessment and review.

[I-03] Consider emitting an explicit event for the `Gsm.initialize` function

Context

- Gsm.sol#L100-L111

Description

The current implementation of `Gsm.initialize` is indirectly emitting an event for the following state variable:

- `_exposureCap`
- `_priceStrategy`
- `admin`

The omission of the sanity check and event emission for `_ghoTreasury` and the omission of an explicit initialization of `_feeStrategy` has been already addresses in two other issues.

While it's true that the `initialize` emits implicit events, Aave should consider emitting a `GsmInitialized` passing to it all the `initialize` input parameters to better track this crucial event from dApps and monitoring system and to distinguish this event from the events that are normally emitted by the execution of the various state variable setters.

Recommendations

Aave should consider emitting a `GsmInitialized` event to better track the proper initialization of the `Gsm`.

Discussion

Aave

Acknowledged

[I-04] `Gsm.burnAfterSeize` should revert early if `amount` is equal to zero for a better DX

Context

- Gsm.sol#L215-L226

Description

The current implementation of `Gsm.burnAfterSeize` allows the liquidator to execute the function passing `amount == 0`.

This is not a security concern because the execution will revert anyway because `GHO.burn()` does not allow burning a zero amount of `GHO` token.

Recommendations

To offer a better DX, save gas cost and revert with a more meaningful error, `Gsm.burnAfterSeize` should revert as soon as possible if `amount == 0`

```
  function burnAfterSeize(uint256 amount) external onlyRole(LIQUIDATOR_ROLE) {
+    require(amount > 0, 'INVALID_AMOUNT');

    // [...]
  }
```

Discussion

Security Researcher

The recommendations have been correctly implemented in the commit 7c03c52c0a271120837cd8e2c2750fe12b9f00c4 (included in the PR 369)

[I-05] `Gsm.rescueTokens` should revert when `amount == 0` to avoid wasting gas and emitting spam events

Context

- Gsm.sol#L174-L189

Description

The `Gsm.rescueTokens` allows the Rescuer to execute the function when the input parameter `amount` is equal to `0`.

The consequence is that no token will be transferred (rescued) to the Treasury and a "useless" and spammy event `TokensRescued(token, to, 0)` will be emitted.

While the gas wasted is not a big problem, critical events like `TokenRescued` should not be emitted when not necessary.

Test

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import './TestGhoBase.t.sol';
import {ERC20 as SolmateERC20} from '../contracts/gho/ERC20.sol';

contract MintableSolmateERC20 is SolmateERC20 {
```

```
  constructor(
    string memory _name,
    string memory _symbol,
    uint8 _decimals
  ) SolmateERC20(_name, _symbol, _decimals) {}

  function mint(address account, uint256 amount) external {
    require(amount > 0, 'INVALID_MINT_AMOUNT');
    _mint(account, amount);
  }
}

contract SRescueZeroTokenTest is TestGhoBase {
  using PercentageMath for uint256;
  using PercentageMath for uint128;

  address internal gsmSignerAddr;
  uint256 internal gsmSignerKey;

  function setUp() public {
    (gsmSignerAddr, gsmSignerKey) = makeAddrAndKey('gsmSigner');
  }

  function testRescueZeroAmountOfTokens() public {
    // Alice sell 100 underlying
    (Gsm gsm, MintableSolmateERC20 solmateUnderlying) = setupTest();

    // someone by mistakes donates 100 underlying to the Gsm
    vm.prank(FAUCET);
    MintableSolmateERC20(solmateUnderlying).mint(BOB, 100e18);
    vm.prank(BOB);
    MintableSolmateERC20(solmateUnderlying).transfer(address(gsm), 100e18);

    // someone by mistakes donates 100 underlying to the Gsm
    vm.prank(FAUCET);
    USDC_TOKEN.mint(BOB, 100e6);
    vm.prank(BOB);
    USDC_TOKEN.transfer(address(gsm), 100e6);

    // someone by mistakes donates 100 GHO to the Gsm
    ghoFaucet(address(gsm), 100e18);

    gsm.grantRole(GSM_TOKEN_RESCUER_ROLE, address(this));

    // Can you rescue 0 GHO?
    vm.expectEmit(true, true, true, true, address(gsm));
    emit TokensRescued(address(GHO_TOKEN), TREASURY, 0);
    gsm.rescueTokens(address(GHO_TOKEN), TREASURY, 0);

    // Can you rescue 0 underlying?
    vm.expectEmit(true, true, true, true, address(gsm));
    emit TokensRescued(address(solmateUnderlying), TREASURY, 0);
    gsm.rescueTokens(address(solmateUnderlying), TREASURY, 0);
```

```
  // Can you rescue 0 USDC (token that is not the Gsm underlying)
  vm.expectEmit(true, true, true, true, address(gsm));
  emit TokensRescued(address(USDC_TOKEN), TREASURY, 0);
  gsm.rescueTokens(address(USDC_TOKEN), TREASURY, 0);
}

function setupTest() internal returns (Gsm, MintableSolmateERC20) {
  // Create a token with 18 decimals
  MintableSolmateERC20 solmateUnderlying = new MintableSolmateERC20(
    'GsmUnderlying',
    'GsmUnderlying',
    18
  );

  // Deploy a Gsm with a 18 decimal underlying
  // The price strategy now is 1:1 -> 1 underlying === 1 GHO
  Gsm gsm = setUpGsm(TREASURY, address(solmateUnderlying), 100_000_000e18, 100_000_0

  // remove the price strategy just to make things easier
  gsm.updateFeeStrategy(address(0));

  // Alice sell 100 token to get 100 GHO
  sellUnderlying(gsm, ALICE, 100e18);

  // Alice should have received 100 GHO
  // Gsm should have 100 underlying
  assertEq(GHO_TOKEN.balanceOf(address(ALICE)), 100e18);
  assertEq(solmateUnderlying.balanceOf(address(gsm)), 100e18);

  // The price strategy has not changed so there should be no excess or deficit
  (uint256 excess, uint256 deficit) = gsm.getCurrentBacking();
  assertEq(excess, 0);
  assertEq(deficit, 0);

  return (gsm, solmateUnderlying);
}

//////////////////////////////////////////////////////
/////// Utility functions
//////////////////////////////////////////////////////

function setUpGsm(
  address treasury,
  address underlying,
  uint128 exposureCap,
  uint128 facilitatorCapacity
) internal returns (Gsm) {
  FixedPriceStrategy fixedPriceStrategy = new FixedPriceStrategy(
    DEFAULT_FIXED_PRICE,
    underlying,
    18
  );
```

```
    Gsm gsm = new Gsm(address(GHO_TOKEN), underlying);
    gsm.initialize(address(this), treasury, address(fixedPriceStrategy), exposureCap);

    // set the fee strategy
    gsm.updateFeeStrategy(address(GHO_GSM_FIXED_FEE_STRATEGY));

    // setup roles
    gsm.grantRole(GSM_LIQUIDATOR_ROLE, address(GHO_GSM_LAST_RESORT_LIQUIDATOR));
    gsm.grantRole(GSM_SWAP_FREEZER_ROLE, address(GHO_GSM_SWAP_FREEZER));

    // setup the new gsm as a facilitator
    IGhoToken(address(GHO_TOKEN)).addFacilitator(
      address(gsm),
      'GSM Facilitator Tester',
      facilitatorCapacity
    );

    return gsm;
  }

  function sellUnderlying(Gsm gsm, address seller, uint256 amountToSell) internal {
    address gsmUnderlying = gsm.UNDERLYING_ASSET();

    vm.prank(FAUCET);
    MintableSolmateERC20(gsmUnderlying).mint(seller, amountToSell);

    vm.startPrank(seller);
    MintableSolmateERC20(gsmUnderlying).approve(address(gsm), amountToSell);

    gsm.sellAsset(uint128(amountToSell), seller);
    vm.stopPrank();
  }

  function backWithUnderlying(Gsm gsm, address backer, uint256 amountToBack) internal
    address gsmUnderlying = gsm.UNDERLYING_ASSET();

    vm.prank(FAUCET);
    MintableSolmateERC20(gsmUnderlying).mint(backer, amountToBack);
    vm.startPrank(backer);
    MintableSolmateERC20(gsmUnderlying).approve(address(gsm), amountToBack);

    gsm.backWith(address(gsmUnderlying), uint128(amountToBack));
    vm.stopPrank();
  }

  function backWithGHO(Gsm gsm, address backer, uint256 amountToBack) internal {
    ghoFaucet(backer, amountToBack);
    vm.startPrank(backer);
    GHO_TOKEN.approve(address(gsm), amountToBack);
    gsm.backWith(address(GHO_TOKEN), uint128(amountToBack));
    vm.stopPrank();
```

```
  }
}
```

Recommendations

Aave should revert the `rescueTokens` when the input parameter `amount` is equal to `0`

Discussion

Security Researcher

The recommendations have been implemented in the commit 7c03c52c0a271120837cd8e2c2750fe12b9f00c4 (included in the PR 369)

[I-06] `Gsm` functions executable behind an auth role should explicitly pass the `msg.sender` in the event emission

Context

- Gsm.sol#L188
- Gsm.sol#L283
- Gsm.sol#L497
- Gsm.sol#L507
- Gsm.sol#L517

Description

The behavior of reporting the `msg.sender` in the event emitted inside an authed function is already implemented in some of the `Gsm` functions like

- `setSwapFreeze`
- `backWith`
- `seize`
- `burnAfterSeize`
- `backWith`

A role can be granted to multiple users or smart contracts, for this reason, the `msg.sender` should be always included in the event emission to better monitor the execution of such crucial functions from dApps and monitoring tools.

The following functions are not following this best practice:

- `rescueTokens`
- `updateGhoTreasury`
- `updatePriceStrategy`
- `updateFeeStrategy`
- `updateExposureCap`

Recommendations

Aave should consider including the `msg.sender` when an event is emitted inside these functions:

- `rescueTokens`
- `updateGhoTreasury`
- `updatePriceStrategy`
- `updateFeeStrategy`
- `updateExposureCap`

Discussion

Aave

Acknowledged Acknowledged, we intend to keep the design as-is for event emissions; the originator of a transaction can be pulled via existing transaction data to supplement event data.

[I-07] Consider refactoring `Gsm.setSwapFreeze` to make the code more clean

Context

- Gsm.sol#L192-L202

Description

The `Gsm.setSwapFreeze` could be refactored to have a more clean and readable code. Here's an example of the possible refactoring suggested:

```
  function setSwapFreeze(bool enable) external onlyRole(SWAP_FREEZER_ROLE) {
    if (enable) {
      require(!_isFrozen, 'GSM_ALREADY_FROZEN');
-      _isFrozen = true;
-      emit SwapFreeze(msg.sender, true);
    } else {
      require(_isFrozen, 'GSM_ALREADY_UNFROZEN');
-      _isFrozen = false;
-      emit SwapFreeze(msg.sender, false);
    }
+    _isFrozen = enable;
+    emit SwapFreeze(msg.sender, enable);
  }
```

Recommendations

Consider refactoring the `Gsm.setSwapFreeze` as explained in the above description.

Discussion

Security Researcher

The recommendations have been implemented in the commit 7c03c52c0a271120837cd8e2c2750fe12b9f00c4 (included in the PR 369)

[I-08] `SampleLiquidator` is not implementing the `triggerBurnAfterSeize` function that should be callable by the Liquidator Role

Context

- SampleLiquidator.sol

Description

Inside the `Gsm` contract, the `LIQUIDATOR_ROLE` group is allowed to perform two specific actions:

- `seize()`
- `burnAfterSeize()`

The `SampleLiquidator` contract that is described as "Minimal Last Resort Liquidator that can serve as sample contract" implements only `triggerSeize` that will trigger the `gsm.seize()` function.

Recommendations

Aave should also implement in `SampleLiquidator` a function that internally calls the `gsm.burnAfterSeize` to make the `SampleLiquidator` feature complete.

Here's an example of a possible implementation of such a function:

```
// TODO: add the proper natspec of the function
function burnAfterSeize(address gsm, uint256 amount) external onlyOwner {
    IGsm(gsm).burnAfterSeize(amount);
}
```

Discussion

Security Researcher

The recommendations have been implemented in the commit 7c03c52c0a271120837cd8e2c2750fe12b9f00c4 (included in the PR 369)

[I-09] Consider validating the `underlyingAssetDecimals` constructor parameter of both the `FixedPriceStrategy` and `FixedPriceStrategy4626` contracts

Context

- FixedPriceStrategy.sol#L35-L36
- FixedPriceStrategy4626.sol#L36-L37

Description

Both the `FixedPriceStrategy` and `FixedPriceStrategy4626` contracts are blindly assuming correct that the `underlyingAssetDecimals` input parameter represents the **real** number of decimals used by the underlying asset `underlyingAsset` .

The contracts could adopt the same strategy used by OpenZeppelin ERC4626 implementation that tries to validate, when possible, that the input parameter that represents the underlying decimals correctly match the real decimal of the asset

Recommendations

Aave should consider to further validating the input parameter `underlyingAssetDecimals` to ensure that the input value does indeed match the number of decimals adopted by the `underlyingAsset`

Here's a possible example of the check to be implemented:

```
  constructor(uint256 priceRatio, address underlyingAsset, uint8 underlyingAssetDecima
    PRICE_RATIO = priceRatio;
    UNDERLYING_ASSET = underlyingAsset;
-    UNDERLYING_ASSET_DECIMALS = underlyingAssetDecimals;
-    _underlyingAssetUnits = 10 ** underlyingAssetDecimals;

+    (bool success, uint8 assetDecimals) = _tryGetAssetDecimals(IERC20(underlyingAsset
+    UNDERLYING_ASSET_DECIMALS = success ? assetDecimals : underlyingAssetDecimals;
+    _underlyingAssetUnits = 10 ** UNDERLYING_ASSET_DECIMALS;
  }

  /**
   * @dev Attempts to fetch the asset decimals. A return value of false indicates that
   */
  function _tryGetAssetDecimals(IERC20 asset_) private view returns (bool, uint8) {
    (bool success, bytes memory encodedDecimals) = address(asset_).staticcall(
      abi.encodeCall(IERC20Metadata.decimals, ())
    );
    if (success && encodedDecimals.length >= 32) {
      uint256 returnedDecimals = abi.decode(encodedDecimals, (uint256));
      if (returnedDecimals <= type(uint8).max) {
        return (true, uint8(returnedDecimals));
      }
    }
    return (false, 0);
  }
```

Discussion

Aave

Acknowledged Acknowledged, this is an item to ensure is appropriately configured when deploying a `GSM` which will go through DAO assessment and review.

[I-10] Consider tracking the value change of `_currentExposure` with a specific event

Context

- Gsm.sol#L64
- Gsm.sol#L248
- Gsm.sol#L402
- Gsm.sol#L430

Description

The internal state variable `_currentExposure` plays a crucial role inside the `Gsm` contract, and it influences vital operations like `sellAsset`, `sellAssetWithSig`, `buyAsset` and `buyAssetWithSig`.

Aave could implement the emission of a specific event that tracks the changes made to `_currentExposure` to later monitor it via dApps or other monitoring systems.

Recommendations

Consider implementing an internal function that updates the value of `_currentExposure` and emits an ad-hoc event to track the "old value" and "new value" of the state variable.

Discussion

Aave

Acknowledged

[I-11] `Gsm.getGhoAmountForBuyAsset` and `Gsm.getGhoAmountForSellAsset` should accept at max `uint128` amounts given that the `buyAsset/sellAsset` can accept at max `type(uint128).max` underlying

Context

- Gsm.sol#L293
- Gsm.sol#L303

Description

The input parameter `assetAmount` of both `Gsm.getGhoAmountForBuyAsset` and `Gsm.getGhoAmountForSellAsset` is declared as a `uint256`.

The value returned by these functions is useful to let the user/smart contract understand how much `GHO` they need to have or will receive (and how much `GHO` will be paid in fee for the operations) when they will perform a buy or sell operation.

Because `buyAsset`, `buyAssetWithSig`, `sellAsset` and `sellAssetWithSig` functions can only accept at max `type(uint128).max` of `assetAmount`, the value returned by `Gsm.getGhoAmountForBuyAsset` and `Gsm.getGhoAmountForSellAsset` could potentially be wrong and could cause problems to the end user or smart contract that rely on those functions.

Recommendations

Aave should update both `getGhoAmountForBuyAsset` and `getGhoAmountForSellAsset` in the `Gsm` contract and `IGsm` interface and declare `assetAmount` as a `uint128`.

Discussion

Security Researcher

The commit 7c03c52c0a271120837cd8e2c2750fe12b9f00c4 (included in the PR 369) harmonizes all the input parameters to the type `uint256`.

Now the methods are coherent between the input and the output types, it's also true that the `buyAsset*` and `sellAsset*` functions will revert if the user tries to buy or sell more than `type(uint128).max` amount of assets because the check against `_currentExposure` and `_exposureCap` will revert.

Aave

This is the intended behaviour.

[I-12] Consider differentiating between the sold asset and the gifted asset inside the

`Gsm.seize` logic

Context

- Gsm.sol#L210-L211

Description

The `Gsm.seize()` function retrieves the `UNDERLYING_ASSET` balance of the contract by executing

```
uint256 underlyingBalance = IERC20(UNDERLYING_ASSET).balanceOf(address(this));
```

And transfer it to the `_ghoTreasury`. After the transfer, it emits the event `Seized(msg.sender, _ghoTreasury, underlyingBalance, ghoMinted)`.

Inside the `underlyingBalance` there could be a mix of "gifted tokens" and assets that have been sold by the users to receive `GHO`.

Aave should consider to:

- Transfer to the `_ghoTreasury` only the `_currentExposure` and to a `recipient` (like it's already done in the `rescueTokens`) the remaining amount (that would represent the "gifted tokens")
- Emit the same event but differentiate the total balance between the assets (sold by the users) that have been seized and the amount of "gifted tokens" that are rescued

Recommendations

Aave should consider differentiating the whole `UNDERLYING_ASSET` balance between "gifted tokens" and "seized tokens" to handle the seize procedure in a more clear way.

Discussion

Aave

Acknowledged

[I-13] Avoid distributing the fees to the treasury when `_accruedFees` is equal to `0`

Context

- Gsm.sol#L271-L276

Description

The `distributeFeesToTreasury` is a function that distributes `_accruedFees` amount of `GHO` tokens to the `_ghoTreasury` address and resets the state variable to zero.

The function is present in both `Gsm` and `Gsm4626` and can be executed by anyone. On `Gsm4626`, before distributing those fees, the contract executes the internal function `_cumulateYieldInGho` that could increase the `_accruedFees` state variable if there is an excess of underlying produced by the ERC4626 yield.

If there is no fee to be distributed (`_accruedFees == 0`), `distributeFeesToTreasury` should avoid updating the `_accruedFees` variable, execute the transfer and emit the event.

By avoiding executing that part of the function's logic, Aave will save gas and avoid emitting a "useless" event.

Important Note: I'm avoiding to handle the case where `_ghoTreasury` is equal to `address(0)` because that issue has been already reported on a different finding.

Recommendations

Aave should consider executing the `distributeFeesToTreasury` only if there are fees to be distributed to the treasury

```
  function distributeFeesToTreasury() public virtual override {
    uint256 accruedFees = _accruedFees;
-    _accruedFees = 0;
-    IERC20(GHO_TOKEN).transfer(_ghoTreasury, accruedFees);
-    emit FeesDistributedToTreasury(_ghoTreasury, GHO_TOKEN, accruedFees);

+   if( accruedFees > 0 ) {
+       _accruedFees = 0;
+       IERC20(GHO_TOKEN).transfer(_ghoTreasury, accruedFees);
+       emit FeesDistributedToTreasury(_ghoTreasury, GHO_TOKEN, accruedFees);
+   }
  }
```

Discussion

Security Researcher

The recommendations have been implemented in the commit 7c03c52c0a271120837cd8e2c2750fe12b9f00c4 (included in the PR 369)

[I-14] Aave should extensively document how they plan to act based on the underlying price fluctuation

Description

The `Gsm` contract (and `Gsm4626`) is initialized with a `priceStrategy` to handle the "swap" (sell/buy) of `underlying <> GHO` .

Currently, there is not a clear explanation about how Aave is planning to act when the market value between `underlying` and `GHO` changes.

Recommendations

Aave should carefully and extensively document how they plan to act when the `PRICE_RATIO` (used by the price strategy) changes (up or down).

Aave should also document, with explicit and real life examples, in which scenario the following "authed" function will be executed

- `setSwapFreeze`
- `seize`
- `burnAfterSeize`
- `backWith`

Discussion

Aave

Acknowledged. Documentation pertaining to the safety mechanisms available in the GSM and their use-cases under various conditions will be prepared.

[I-15] `Gsm.getAssetAmountForBuyAsset` and `Gsm.getAssetAmountForSellAsset` use `uint128` for both inputs and return values

Context

- Gsm.sol#L312-L323
- Gsm.sol#L326-L337

Description

The input parameter `ghoAmount` of both `Gsm.getAssetAmountForBuyAsset` and `Gsm.getAssetAmountForSellAsset` is declared as a `uint256`.

All the variables in `Gsm` (like `_exposureCap`, `_currentExposure`, `_accruedFees`) and the limits of a Facilitator are expressed with the `uint128` type. The same is true for the input parameters of the actions can be executed on the contract.

By accepting (and returning) `uint256` these functions could return values that could not be later accepted by the `Gsm` contract and would lead to a revert.

Recommendations

Update both the `inputs` and `returns` value types of both the `Gsm.getAssetAmountForBuyAsset` and `Gsm.getAssetAmountForSellAsset` to be sure that the returned values are compatible with the `Gsm` (and `GHO`) executable functions.

Discussion

Security Researcher

The commit 7c03c52c0a271120837cd8e2c2750fe12b9f00c4 (included in the PR 369) harmonizes all the input parameters to the type `uint256`.

Now the methods are coherent between the input and the output types, it's also true that the `buyAsset*` and `sellAsset*` functions will revert if the user tries to buy or sell more than `type(uint128).max` amount of assets because the check against `_currentExposure` and `_exposureCap` will revert.

Aave

This is the intended behaviour.

[I-16] The `FixedPriceStrategy4626` does not adhere to the `IGsmPriceStrategy` interface

Context

- FixedPriceStrategy4626.sol#L17

Description

Both the `Gsm4626` and `FixedPriceStrategy4626` will be deployed with `UNDERLYING_ASSET` equal to the vault's share token and not the vault's underlying.

The `IGsmPriceStrategy` interface states that `PRICE_RATIO()` (the "magic" getter of the `PRICE_RATIO` immutable variable) should

> @notice Returns the price ratio from underlying asset to GHO @dev e.g. A ratio of 2e18 means 2 GHO per 1 underlying asset @return The price ratio from underlying asset to GHO (expressed in WAD)

Following the `IGsmPriceStrategy` documentation, the `FixedPriceStrategy4626.PRICE_RATIO` should represent how many `GHO` one vault's share is worth. This is not true for the current implementation of the `FixedPriceStrategy4626` contract because the `PRICE_RATIO` represents how many `GHO` one vault's underlying is worth.

The same comply problem can be found for the `UNDERLYING_ASSET_DECIMALS` and `_underlyingAssetUnits` immutable variable. By following the `IGsmPriceStrategy` documentations, those variables should represent the asset decimals and asset units of the `Gsm` `UNDERLYING_ASSET` but in `FixedPriceStrategy4626` they instead represent the asset decimals and units of the Vault's underlying and not the vault's share (that is used as the `Gsm` and `FixedPriceStrategy` underlying).

These discrepancies could create both confusions and problems for the users and integrators that will rely on those values without fully understanding that `FixedPriceStrategy4626` does not fully behave as the `IGsmPriceStrategy` describe.

Recommendations

Aave should:

1. Override the `PRICE_RATIO()` function in `FixedPriceStrategy4626` to be able to override the natspec documentation of the function to reflect the difference between the "standard" behavior compared to what the `IGsmPriceStrategy` states
2. Add internal comments and natspec documentation to the `FixedPriceStrategy4626` `PRICE_RATIO` to better document this difference and remove any possible doubts
3. Properly document the discrepancies of both the `UNDERLYING_ASSET_DECIMALS` and `_underlyingAssetUnits` that are not representing such values for the `UNDERLYING_ASSET` but for the Vault's underlying.

Discussion

Security Researcher

The recommendations have been implemented in the commit 7c03c52c0a271120837cd8e2c2750fe12b9f00c4 (included in the PR 369)

[I-17] `Gsm4626` that uses `ERC4626` vaults with fees will experience `GHO` excess or deficit based on the fee change

Context

- Gsm4626.sol

Description

The ERC-4626: Tokenized Vaults standard states that the vault can include fees.

The concept of "fee" for such standard is defined as following:

> An amount of assets or shares charged to the user by the Vault. Fees can exists for deposits, yield, AUM, withdrawals, or anything else prescribed by the Vault.

If the Vault fee can be changed dynamically during the time, the `Gsm4626` will incur in `GHO` **excess** or `GHO` deficit **without** that the price of the Vault's underlying token has changed (and as a consequence, the `Gsm` price strategy).

The `Gsm4626` will see an excess/deficit because the `PRICE_RATIO` of the `FixedPriceStrategy4626` is related to the Vault's underlying and not the Vault's share. Because of this, when the fee on the vault's operations change (increase or the crease) the value returned by `FixedPriceStrategy4626.getAssetPriceInGho(...)` will return a greater (if fee has decreased) or lower (if fee has increased) value.

The `getAssetPriceInGho` function is used in many places inside `Gsm` and is also used implicitly by the `Gsm4626._cumulateYieldInGho` internal function.

Supporting ERC4626 with dynamic fees will lead to problematic behaviors like

- Minting `GHO` because of ERC4626 Yields (`_cumulateYieldInGho`) when there has been no real yield (the value of the underlying has not changed, only the vault's fee has been decreased)

- Being able to execute `backWith` to cover the `Gsm4626` deficit when there has been no real deficit (the underlying price has not changed)

1. Vault fees decrease: the `Gsm4626` thinks that there has been an excess in `GHO` and will be able to trigger `_cumulateYieldInGho()`. The `Gsm` has not accumulated any real yield because the underlying amount in the vault has not increased and the `FixedPriceStrategy4626` has not increased its value.
2. Vault fees increase: the `Gsm4626` things that there have been a deficit in `GHO` and will be able to execute `backWith` to cover the deficit. The `Gsm` in reality, has not really incurred in any deficit because the there has been no real loss and the `FixedPriceStrategy4626` has not decreased its value.

Test

`MockERC4626WithFees.sol`

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import {Math} from '@openzeppelin/contracts/utils/math/Math.sol';
import {ERC4626} from '@openzeppelin/contracts/token/ERC20/extensions/ERC4626.sol';
import {ERC20} from '@openzeppelin/contracts/token/ERC20/ERC20.sol';
import {IERC20} from '@openzeppelin/contracts/token/ERC20/IERC20.sol';

contract MockERC4626WithFees is ERC4626 {
  // Example taken from https://docs.openzeppelin.com/contracts/4.x/erc4626
  using Math for uint256;

  address private immutable _feeRecipient;
  uint256 private _feeBps;

  constructor(
    string memory _name,
    string memory _symbol,
    address _asset,
    address feeRecipient,
    uint256 feeBps
  ) ERC4626(IERC20(_asset)) ERC20(_name, _symbol) {
    _feeRecipient = feeRecipient;
    _feeBps = feeBps;
  }

  /** @dev See {IERC4626-previewDeposit}. */
  function previewDeposit(uint256 assets) public view virtual override returns (uint25
    uint256 fee = _feeOnTotal(assets, _entryFeeBasePoint());
    return super.previewDeposit(assets - fee);
  }

  /** @dev See {IERC4626-previewMint}. */
```

```
function previewMint(uint256 shares) public view virtual override returns (uint256)
  uint256 assets = super.previewMint(shares);
  return assets + _feeOnRaw(assets, _entryFeeBasePoint());
}

/** @dev See {IERC4626-previewWithdraw}. */
function previewWithdraw(uint256 assets) public view virtual override returns (uint2
  uint256 fee = _feeOnRaw(assets, _exitFeeBasePoint());
  return super.previewWithdraw(assets + fee);
}

/** @dev See {IERC4626-previewRedeem}. */
function previewRedeem(uint256 shares) public view virtual override returns (uint256
  uint256 assets = super.previewRedeem(shares);
  return assets - _feeOnTotal(assets, _exitFeeBasePoint());
}

/** @dev See {IERC4626-_deposit}. */
function _deposit(
  address caller,
  address receiver,
  uint256 assets,
  uint256 shares
) internal virtual override {
  uint256 fee = _feeOnTotal(assets, _entryFeeBasePoint());
  address recipient = _entryFeeRecipient();

  super._deposit(caller, receiver, assets, shares);

  if (fee > 0 && recipient != address(this)) {
    IERC20(asset()).transfer(recipient, fee);
  }
}

/** @dev See {IERC4626-_deposit}. */
function _withdraw(
  address caller,
  address receiver,
  address owner,
  uint256 assets,
  uint256 shares
) internal virtual override {
  uint256 fee = _feeOnRaw(assets, _exitFeeBasePoint());
  address recipient = _exitFeeRecipient();

  super._withdraw(caller, receiver, owner, assets, shares);

  if (fee > 0 && recipient != address(this)) {
    IERC20(asset()).transfer(recipient, fee);
  }
}

function setFeeBasePoint(uint256 newFeeBps) external {
```

```
    _feeBps = newFeeBps;
  }

  function _entryFeeBasePoint() internal view virtual returns (uint256) {
    return _feeBps;
  }

  function _entryFeeRecipient() internal view virtual returns (address) {
    return _feeRecipient;
  }

  function _exitFeeBasePoint() internal view virtual returns (uint256) {
    return _feeBps;
  }

  function _exitFeeRecipient() internal view virtual returns (address) {
    return _feeRecipient;
  }

  function _feeOnRaw(uint256 assets, uint256 feeBasePoint) private pure returns (uint2
    return assets.mulDiv(feeBasePoint, 1e5, Math.Rounding.Up);
  }

  function _feeOnTotal(uint256 assets, uint256 feeBasePoint) private pure returns (uin
    return assets.mulDiv(feeBasePoint, feeBasePoint + 1e5, Math.Rounding.Up);
  }
}
```

SGsm4626WithFee.t.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import './TestGhoBase.t.sol';

//
// ========= IMPORTANT NOTE =========
//
// 1) you need to import into the /mock folder the `MockERC4626WithFees.sol` contract
// 2) you need to import the `MockERC4626WithFees` into `TestGhoBase.t.sol` like shown
//    `import {MockERC4626WithFees} from './mocks/MockERC4626WithFees.sol';`

contract SGsm4626WithFeeTest is TestGhoBase {
  using PercentageMath for uint256;
  using PercentageMath for uint128;

  function testVaultFeeChangeInTimeGeneraticExcessOrDeficit() public {
    address vaultFeeRecipient = makeAddr('vaultFeeRecipient');

    // this is the underlying used by the VAULT
    TestnetERC20 vaultUnderlying = new TestnetERC20(
```

```
  'VaultUnderlying',
  'VaultUnderlying',
  18,
  FAUCET
);

// The Gsm contract can be initialized with an empty treasury
(Gsm4626 gsm, MockERC4626WithFees vault) = setUpGsm4626(
  TREASURY,
  address(vaultUnderlying),
  100_000_000e18,
  100_000_000e18,
  vaultFeeRecipient
);

// Ok now we want to show case that supporting a Gsm4626 that can change the fees
// underlying asset does not change in price, the Gsm could "produce" excess or de

// Let's test it out
(uint256 excess, uint256 deficit) = gsm.getCurrentBacking();

// at the very start there's no excess or deficit
assertEq(excess, 0);
assertEq(deficit, 0);

// OP: sell vault shares to Gsm

// 1) mint 100 vault underlying to ALICE
vm.prank(FAUCET);
vaultUnderlying.mint(ALICE, 100e18);

// 2) ALICE approve vault
vm.prank(ALICE);
vaultUnderlying.approve(address(vault), 100e18);

// 3) ALICE deposit the underlying to mint shares
vm.prank(ALICE);
uint256 sharesMinted = vault.deposit(100e18, ALICE);

// 4) ALICE approve the Gsm to sell the vault's shares
vm.prank(ALICE);
vault.approve(address(gsm), sharesMinted);

// 4) ALICE sell the shares to the Gsm to receive GHO
vm.prank(ALICE);
gsm.sellAsset(uint128(sharesMinted), ALICE);

(excess, deficit) = gsm.getCurrentBacking();

// Alice has minted some GHO by selling asset. excess and deficit should be anyway
assertEq(excess, 0);
assertEq(deficit, 0);
```

```
    // Vault's owner change the fee setting them to 15%
    // This will generate a DEFICIT because `FixedPriceStrategy4626.getAssetPriceInGho
    // will return a smaller value for the `_currentExposure` (compared to mintedGHO)
    vault.setFeeBasePoint(15_000);
    (excess, deficit) = gsm.getCurrentBacking();
    assertEq(excess, 0);
    assertGt(deficit, 0);

    // Vault's owner change the fee setting them to 5%
    // This will generate an EXCESS because `FixedPriceStrategy4626.getAssetPriceInGho
    // will return a bigger value for the `_currentExposure` (compared to mintedGHO)
    vault.setFeeBasePoint(5_000);
    (excess, deficit) = gsm.getCurrentBacking();
    assertGt(excess, 0);
    assertEq(deficit, 0);
  }

  // ////////////////////////////////////////////////////
  // /////// Utility functions
  // ////////////////////////////////////////////////////

  function setUpGsm4626(
    address treasury,
    address underlying,
    uint128 exposureCap,
    uint128 facilitatorCapacity,
    address vaultFeeRecipient
  ) internal returns (Gsm4626, MockERC4626WithFees) {
    // Deploy the vault
    MockERC4626WithFees vault = new MockERC4626WithFees(
      'mockERC4626',
      'mockERC4626',
      underlying,
      vaultFeeRecipient,
      10_000 // 10% fee
    );

    FixedPriceStrategy4626 fixedPriceStrategy = new FixedPriceStrategy4626(
      1e18,
      address(vault),
      18
    );

    Gsm4626 gsm = new Gsm4626(address(GHO_TOKEN), address(vault));
    gsm.initialize(address(this), treasury, address(fixedPriceStrategy), exposureCap);

    // set zero fee
    gsm.updateFeeStrategy(address(0));

    // setup roles
    gsm.grantRole(GSM_LIQUIDATOR_ROLE, address(GHO_GSM_LAST_RESORT_LIQUIDATOR));
    gsm.grantRole(GSM_SWAP_FREEZER_ROLE, address(GHO_GSM_SWAP_FREEZER));
```

```
      // setup the new gsm as a facilitator
      IGhoToken(address(GHO_TOKEN)).addFacilitator(
        address(gsm),
        'GSM Facilitator Tester',
        facilitatorCapacity
      );

      return (gsm, vault);
    }
  }
```

Recommendations

Aave should not add support to ERC4626 Vault that have fees or that have fees that could be updated during the lifecycle of the vault.

If Aave plan to do so, they should modify the `Gsm4626` and `FixedPriceStrategy4626` to properly handle this edge scenario.

Discussion

Aave

Acknowledged, when a GSM is deployed, it must undergo assessment and review by the DAO to ensure appropriate configuration as well as underlying asset compatibility with the functionality of the GSM.

[I-18] `USDC` and `USDT` could lock `Gsm` underlying if the `Gsm` is blacklisted

Context

- Gsm.sol

Description

`ERC20` tokens like `USDT` and `USDC` have blacklisting logic inside of them that could prevent the execution of `transfer` and `transferFrom` if (depending on the operation) the `msg.sender`, `sender` or `receiver` has been added to such blacklist.

If the `Gsm` itself is added to the blacklist, all the operations that involve the transfer of the `Gsm` `UNDERLYING_TOKEN` will revert.

Here is the full list of operations that would revert:

- `sellAsset`
- `sellAssetWithSig`
- `buyAsset`
- `buyAsssetWithSig`
- `rescueToken`
- `seize`
- `backWith`

This would mean that users won't be able to purchase or sell `GHO` and the `Gsm` authed users would not be able to rescue the underlying, seize the `Gsm` or back it if needed.

Recommendations

Unfortunately, there's not any possible recommendations that could solve such a scenario.

Aave should be aware of this possibility and warn the users that such scenario could happen and which are the consequences if the `Gsm` is blacklisted on the underlying token.

Discussion

Aave

Acknowledged, when a GSM is deployed, it must undergo assessment and review by the DAO to ensure appropriate configuration as well as underlying asset compatibility with the functionality of the GSM.

[I-19] Consider refactoring `FixedPriceStrategy4626` by inheriting `FixedPriceStrategy`

Context

- FixedPriceStrategy.sol
- FixedPriceStrategy4626.sol

Description

The main difference between these two Price Strategy contracts is that `FixedPriceStrategy4626` needs to perform conversions to/from shares/underlying before returning the underlying calculation.

The core mechanism of those contracts are the same, and they share the same storage layout. The `FixedPriceStrategy4626` contract could be refactored to directly inherit from `FixedPriceStrategy` and just `override` the functions `getAssetPriceInGho` and `getGhoPriceInAsset` to apply the custom conversion from/to shares before executing the custom logic.

Recommendations

Aave should consider refactoring the `FixedPriceStrategy4626` to inherit directly from `FixedPriceStrategy`.

Option 1

This is the "base" version that just inherits from `FixedPriceStrategy` and fully overrides the needed functions.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import {IERC4626} from '@openzeppelin/contracts/interfaces/IERC4626.sol';
import {IGsmPriceStrategy} from './interfaces/IGsmPriceStrategy.sol';
import {FixedPriceStrategy} from './FixedPriceStrategy.sol';

/**
 * @title FixedPriceStrategy4626
 * @author Aave
 * @notice Price strategy involving a fixed-rate conversion from an ERC4626 asset to GI
 */
contract FixedPriceStrategy4626 is FixedPriceStrategy {
  /**
   * @dev Constructor
   * @param priceRatio The price ratio from underlying asset to GHO (expressed in WAD)
   * @param underlyingAsset The address of the underlying asset
   * @param underlyingAssetDecimals The number of decimals of the underlying asset
   */
  constructor(
    uint256 priceRatio,
    address underlyingAsset,
    uint8 underlyingAssetDecimals
  ) FixedPriceStrategy(priceRatio, underlyingAsset, underlyingAssetDecimals) {
    // Intentionally left blank
  }

  /// @inheritdoc IGsmPriceStrategy
  function getAssetPriceInGho(uint256 assetAmount) external view override returns (uin
    // conversion from 4626 shares to 4626 assets (rounding down)
    uint256 vaultAssets = IERC4626(UNDERLYING_ASSET).previewRedeem(assetAmount);
```

```
    return (vaultAssets * PRICE_RATIO) / _underlyingAssetUnits;
  }

  /// @inheritdoc IGsmPriceStrategy
  function getGhoPriceInAsset(uint256 ghoAmount) external view override returns (uint2
    if (PRICE_RATIO == 0) return 0;
    uint256 vaultAssets = (ghoAmount * _underlyingAssetUnits) / PRICE_RATIO;
    // conversion from 4626 assets to 4626 shares (rounding down)
    return IERC4626(UNDERLYING_ASSET).previewDeposit(vaultAssets);
  }
}
```

The `FixedPriceStrategy` needs to be also modified because both `getAssetPriceInGho` and `getGhoPriceInAsset` must be declared as `virtual` to be `overrided` by `FixedPriceStrategy4626`

Option 2

The second option is similar to `Option 1` but further inherit from `FixedPriceStrategy` by executing the `getAssetPriceInGho` and `getGhoPriceInAsset` functions directly using the `FixedPriceStrategy` implementation.

There are two options here depending on Aave choice

1. Declare `FixedPriceStrategy.getAssetPriceInGho` and `FixedPriceStrategy.getGhoPriceInAsset` public to be callable from `FixedPriceStrategy4626`
2. If you want to avoid declaring those functions as `public` , Aave can create an `internal` version for those function, move the logic over there and then both `FixedPriceStrategy` and `FixedPriceStrategy4626` will execute the `internal` version of them

For simplicity, I will just showcase how to implement the first variant

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import {IERC4626} from '@openzeppelin/contracts/interfaces/IERC4626.sol';
import {IGsmPriceStrategy} from './interfaces/IGsmPriceStrategy.sol';
import {FixedPriceStrategy} from './FixedPriceStrategy.sol';

/**
 * @title FixedPriceStrategy4626
 * @author Aave
 * @notice Price strategy involving a fixed-rate conversion from an ERC4626 asset to G
 */
contract FixedPriceStrategy4626 is FixedPriceStrategy {
  /**
   * @dev Constructor
```

```
   * @param priceRatio The price ratio from underlying asset to GHO (expressed in WAD)
   * @param underlyingAsset The address of the underlying asset
   * @param underlyingAssetDecimals The number of decimals of the underlying asset
   */
  constructor(
    uint256 priceRatio,
    address underlyingAsset,
    uint8 underlyingAssetDecimals
  ) FixedPriceStrategy(priceRatio, underlyingAsset, underlyingAssetDecimals) {
    // left empty on purpose
  }

  /// @inheritdoc IGsmPriceStrategy
  function getAssetPriceInGho(uint256 assetAmount) public view override returns (uint2
    // conversion from 4626 shares to 4626 assets (rounding down)
    uint256 vaultAssets = IERC4626(UNDERLYING_ASSET).previewRedeem(assetAmount);
    return super.getAssetPriceInGho(vaultAssets);
  }

  /// @inheritdoc IGsmPriceStrategy
  function getGhoPriceInAsset(uint256 ghoAmount) public view override returns (uint256
    if (PRICE_RATIO == 0) return 0;
    uint256 vaultAssets = super.getGhoPriceInAsset(ghoAmount);
    // conversion from 4626 assets to 4626 shares (rounding down)
    return IERC4626(UNDERLYING_ASSET).previewDeposit(vaultAssets);
  }
}
```

Discussion

Aave

Acknowledged

[I-20] General Natspec documentation improvements, and style suggestions

Description

The issue tries to group together different natspec documentation improvements and style suggestion that Aave could adopt to improve the DX and readability of the code for other developers or security researchers.

The suggestions made try to follow both the Solidity Natspec Format and the Solidity Style Guide published on the Solidity Documentation website.

- FixedFeeStrategy.sol#L15-L16: `_buyFee` and `_sellFee` are `immutable` variables that should be declared in uppercase
- FixedFeeStrategy.sol#L15-L16: `_buyFee` and `_sellFee` variables should be documented with natspec
- IGsm.sol: `event BuyAsset`, `event SellAsset`, `event SwapFreeze`, `event Seized`, `event BurnAfterSeize`, `event BackingProvided`, `event PriceStrategyUpdated`, `event FeeStrategyUpdated`, `event ExposureCapUpdated`, `event TokensRescued` could switch from `@dev` to `@notice` natspec tag
- GsmRegistry.sol#L16: `_gsmList` is missing natspec documentation
- GsmRegistry.sol#L23: The `constructor` input parameter `owner` is shadowing the `owner()` magic getter of `Ownable`. Change the input variable name to `newOwner` to avoid the warning.
- IGsmRegistry.sol: `event GsmAdded` and `event GsmRemoved` could switch from `@dev` to `@notice` natspec tag
- IGsmRegistry.sol#L14: `event GsmAdded` should declare the input parameter `gsmAddress` as `indexed`
- IGsmRegistry.sol#L20: `event GsmRemoved` should declare the input parameter `gsmAddress` as `indexed`
- SampleLiquidator.sol#L10: The `@notice` natspec of the `SampleLiquidator` contract is probably referring to the "old" contract name (used in a previus version of the contract).
- FixedPriceStrategy.sol#L24: `_underlyingAssetUnits` is an `immutable` variable that should be declared in uppercase
- FixedPriceStrategy.sol#L24: `_underlyingAssetUnits` variable should be documented with natspec
- FixedPriceStrategy4626#L25: `_underlyingAssetUnits` is an `immutable` variable that should be declared in uppercase
- FixedPriceStrategy4626#L25: `_underlyingAssetUnits` variable should be documented with natspec
- FixedPriceStrategy4626.sol#L41: consider renaming the `assetAmount` parameter to `sharesAmount`. The `previewRedeem` of the `ERC4626` contract takes as input a share amount and not an asset amount.
- Gsm4626.sol#L6: The `IGsmPriceStrategy` import statement can be removed because never used inside the contract's code
- Gsm.sol#L58-L65: `_ghoTreasury`, `_priceStrategy`, `_isFrozen`, `_isSeized`, `_feeStrategy`, `_exposureCap`, `_currentExposure` and `_accruedFees` variables should be documented with natspec
- Gsm.sol#L79: The revert error message "GSM_SEIZED_SWAPS_DISABLED" used by `notSeized` is incorrect because the modified is invoked not only by the buy/sell operations but also by the `seize()` function that does not involve any swap operation. Consider to use less specific revert message to clear any possible confusion.

Recommendations

Aave should consider following the changes suggested in the description to provide an improved DX for both the developers, integrators and security researchers.

Discussion

Security Researcher

The following recommendations have been implemented in the commit 7c03c52c0a271120837cd8e2c2750fe12b9f00c4 (included in the PR 369)

- `GsmRegistry.constructor` has renamed the parameter `owner` to `newOwner`
- Events in `IGsmRegistry` has correctly declared as `indexed` the needed parameters
- `SampleLiquidator` natspec has been updated
- The natspec in `FixedPriceStrategy` and `FixedPriceStrategy4626` for the `PRICE_RATIO` and `_underlyingAssetUnits` has been updated
- `Gsm.notFrozen` and `Gsm.notSeized` revert error message has been updated

The remaining points have not been fixed yet.

Aave

Intent is to acknowledge the rest.

[I-21] Open questions, suggestions and discussions

Description

This is not a security issue report, but more a place where I would like to share some discussion and suggestions.

1. Add a `canSwap` getter on the `Gsm` that returns `true` if the user can execute `buyAsset`, `buyAssetWithSig`, `sellAsset`, `sellAssetWithSig`. The function will return true if `frozen == false && seized == false`
2. Add a `canBuyAsset(uint256 assetAmount)` utility function. The function returns `true` only if `canSwap() == true && _currentExposure >= amount`. Bonus: the action should also take in consideration the facilitator current `GHO` capacity.
3. Add a `canSellAsset(uint256 assetAmount)` utility function. The function returns `true` only if `canSwap() == true && _currentExposure + amount <= _exposureCap`. Bonus: the action

should also take in consideration the facilitator current `GHO` capacity.

4. Consider emitting an `Event` when `Gsm4626._cumulateYieldInGho` accrued yield to monitor it
5. Consider emitting an `Event` when `Gsm._accruedFees` is updated to monitor it
6. Should an "active" (not frozen, not seized, with liquidity and everything working properly) `Gsm` be removable by executing `GsmRegistry.removeGsm` ? If yes, which are the possible scenarios to allow such an operation?
7. During the pre-audit discussion, Aave said that it will be possible to have different `Gsm` with the same `underlying` tokens but different `feeStrategies` . In that case, Aave should consider creating a "router" utility to redirect the user to the more favorable `Gsm` to execute the buy/sell operation. (note that buy/sell with signature can be performed only on a specific `Gsm`)
8. How should all utility getters used to later execute sell/buy operations behave when the `Gsm` is frozen? For example, `getAssetAmountForBuyAsset` , `getAssetAmountForSellAsset` , `getAvailableUnderlyingExposure` , `getAvailableLiquidity` and so on, should return zero if the swap operations are frozen?
9. `backWith` could be simplified by emitting the `BackingProvided` in one single place

```
if (asset == GHO_TOKEN) {
  IGhoToken(GHO_TOKEN).transferFrom(msg.sender, address(this), amount);
  IGhoToken(GHO_TOKEN).burn(amount);

-  emit BackingProvided(msg.sender, GHO_TOKEN, amount, amount, deficit - amount);
} else {
  _currentExposure += amount;
  IERC20(UNDERLYING_ASSET).safeTransferFrom(msg.sender, address(this), amount);

-  emit BackingProvided(msg.sender, UNDERLYING_ASSET, amount, ghoToBack, deficit - gho
}

+// 1) asset can only be `GHO_TOKEN` or `UNDERLYING_ASSET` otherwise the function woul
+// 2) `amount` will always be the amount of asset used to back the Gsm
+// 3) when `asset == GHO_TOKEN` `amount === ghoToBack`

+emit BackingProvided(msg.sender, asset, amount, ghoToBack, deficit - ghoToBack);
```

10. Add `returns` values to the `sellAsset` , `sellAssetWithSig` , `buyAsset` , `buyAssetWithSig` to let the integrator know how many `GHO` / `ASSET` the `receiver` has received when he has executed a sell / buy operation
11. Given that the same `UNDERLYING` can be used by multiple `Gsm` , Aave should consider building utility contracts that help the user/integrator to understand which `Gsm` is better to use depending on the capacity (how many `GHO` can be minted by the facilitator), the `Gsm` fees and the available liquidity/exposure. Such utility should also consider splitting the buy/sell operation over multiple `Gsm` if one `Gsm` cannot satisfy the whole amount of asset to be bought/sold

Discussion

Security Researcher

The commit 7c03c52c0a271120837cd8e2c2750fe12b9f00c4 (included in the PR 369) only handles the first point of the issue

Aave

Intent is to acknowledge the rest.

[G-01] Consider calling `_beforeBuyAsset` and `_beforeSellAsset` after performing sanity/base checks to save gas

Context

- Gsm.sol#L397-L400
- Gsm.sol#L427-L431

Description

The hooks `_beforeBuyAsset` and `_beforeSellAsset` are executes as the very first instruction of the `_buyAsset` and `_sellAsset` internal functions.

Those same functions, after executing the hook, are performing some very basic requirements checks that could be moved before the hooks to save some gas in case of revert.

```
function _buyAsset(address originator, uint128 amount, address receiver) internal {
>>    _beforeBuyAsset(originator, amount, receiver);
>>    require(amount > 0, 'INVALID_AMOUNT');
>>    require(_currentExposure >= amount, 'INSUFFICIENT_AVAILABLE_EXOGENOUS_ASSET_LIQU

    // remaining function logic code...
}

function _sellAsset(address originator, uint128 amount, address receiver) internal {
>>    _beforeSellAsset(originator, amount, receiver);
>>    require(amount > 0, 'INVALID_AMOUNT');
>>    _currentExposure += amount;
>>    require(_currentExposure <= _exposureCap, 'EXOGENOUS_ASSET_EXPOSURE_TOO_HIGH');
```

```
    // remaining function logic code...
}
```

To validate the suggestion, let's look at `Gsm4626` that implements the `_beforeBuyAsset` hook

```
  /// @inheritdoc Gsm
  function _beforeBuyAsset(address, uint128, address) internal override {
    _cumulateYieldInGho();
  }

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

If the `_buyAsset` function is going to revert, all the gas spent on the execution of `_cumulateYieldInGho` would be wasted.

Recommendations

Aave should consider executing the `_beforeBuyAsset` and `_beforeSellAsset` hooks only after the `require` statements of `_buyAsset` and `_sellAsset` .

Important Security Note: the following suggestion is secure **if and only if** the hook's logic does not modify the state variables used by the `require` statement.

```
  function _buyAsset(address originator, uint128 amount, address receiver) internal {
-     _beforeBuyAsset(originator, amount, receiver);

    require(amount > 0, 'INVALID_AMOUNT');
    require(_currentExposure >= amount, 'INSUFFICIENT_AVAILABLE_EXOGENOUS_ASSET_LIQUID

+   _beforeBuyAsset(originator, amount, receiver);

    // remaining function logic code...
  }

  function _sellAsset(address originator, uint128 amount, address receiver) internal {
-     _beforeSellAsset(originator, amount, receiver);
```

```
    require(amount > 0, 'INVALID_AMOUNT');
    _currentExposure += amount;
    require(_currentExposure <= _exposureCap, 'EXOGENOUS_ASSET_EXPOSURE_TOO_HIGH');

+   _beforeSellAsset(originator, amount, receiver);

    // remaining function logic code...
}
```

Discussion

Aave

Acknowledged Acknowledged, the hooks are placed intentionally at the start of the logic, given they could potentially modify state variables subsequently used in checks.

[G-02] In `Gsm._sellAsset_` `_currentExposure` could be updated after the `require` to save gas on revert

Context

- Gsm.sol#L430-L431

Description

To save gas on `revert` the `_currentExposure` update instruction could be moved after the `require` statement.

```
function _sellAsset(address originator, uint128 amount, address receiver) internal {
    _beforeSellAsset(originator, amount, receiver);

    require(amount > 0, 'INVALID_AMOUNT');

-    _currentExposure += amount;
-    require(_currentExposure <= _exposureCap, 'EXOGENOUS_ASSET_EXPOSURE_TOO_HIGH');

+   require(_currentExposure + amount <= _exposureCap, 'EXOGENOUS_ASSET_EXPOSURE_TOO_H
+   _currentExposure += amount;
```

```
    // remaining function logic code...
}
```

Recommendations

Consider updating the `_currentExposure` state variable only after the `require` statement has been executed to save gas in case of revert.

Discussion

Security Researcher

The recommendations have been implemented in the commit 7c03c52c0a271120837cd8e2c2750fe12b9f00c4 (included in the PR 369)

[G-03] Consider saving `_ghoTreasury` in a local variable to avoid an additional `SLOAD` during `Gsm.seize()` execution

Context

- Gsm.sol#L210C43-L211

Description

The `seize()` function is executing twice an `SLOAD` to load the `_ghoTreasury` state variable value.

The function's gas usage can be improved by doing just one `SLOAD` and save the value in a local variable.

Recommendations

Consider saving `_ghoTreasury` in a local variable to save gas

```
  function seize() external notSeized onlyRole(LIQUIDATOR_ROLE) {
    _isSeized = true;

    (, uint256 ghoMinted) = IGhoToken(GHO_TOKEN).getFacilitatorBucket(address(this));
    uint256 underlyingBalance = IERC20(UNDERLYING_ASSET).balanceOf(address(this));
```

```
-     IERC20(UNDERLYING_ASSET).safeTransfer(_ghoTreasury, underlyingBalance);
-     emit Seized(msg.sender, _ghoTreasury, underlyingBalance, ghoMinted);

+     address ghoTreasury = _ghoTreasury;
+     IERC20(UNDERLYING_ASSET).safeTransfer(ghoTreasury, underlyingBalance);
+     emit Seized(msg.sender, ghoTreasury, underlyingBalance, ghoMinted);
  }
```

Discussion

Aave

Acknowledged Gas savings from this change are marginal, and the seize function is expected to be executed extremely infrequently.

[Fix Review Period] Issues and Suggestions on the changes made during the fix period

1. `Gsm _priceStrategy` is now immutable and is initialized during the contract's `constructor`. The previous checks `IGsmPriceStrategy(priceStrategy).UNDERLYING_ASSET() == UNDERLYING_ASSET` and `IGsmPriceStrategy(priceStrategy).PRICE_RATIO() != 0` are not performed anymore. Aave should add those checks into the contract's constructor to prevent deploying a contract with incorrect input parameters.
2. Aave should consider changing the `bool roundUp` parameter passed to the Price Strategy (both the "normal" one and the `4626` one) `getAssetPriceInGho` and `getGhoPriceInAsset` from `bool` to `Math.Rounding`. By doing this, the parameters become explicit and are easier to read from the sources that will call those functions.
3. `Gsm.seize` and `Gsm.burnAfterSeize` are now returning values. Aave should update `SampleLiquidator` to "bubble up" those returned values that could be helpful when `SampleLiquidator` is executed.
4. The behavior of `sellAsset*` and `buyAsset*` has changed. When users sell assets they could end up selling less assets for the same amount of `GHO` and when they buy assets they could end up receiving more assets compared to what has been specified as the function's input parameter. This behavior could produce side effects that could end up reverting the transaction based on both the `underlying ERC20` token used by the `GSM` or how the caller is implemented.
    i. When user calls `sellAsset*`, the `GSM` will execute `IERC20(UNDERLYING_ASSET).safeTransferFrom(originator, address(this), assetAmount);` where `assetAmount` could be lower compared to `maxAmount` specified by the user. The user/smart contract has pre-approved `maxAmount` and if `maxAmount < assetAmount` the allowance that the user/smart contract has given to the `GSM` will not be

fully consumed. Tokens like `USDT` will revert if the user/smart contract set the allowance `> 0` when the allowance is not equal to `0`. If this scenario happens (not all the allowance has been consumed) the transaction will revert.

ii. In general, such behavior (selling less tokens, receiving more tokens) should be well documented to warn the user of such unexpected behavior. Aave should also document which are the configurations (given `amount`, `underlying` decimals and `buy/sell` fees) that this scenario could happen. Aave does not know which checks/requirements the smart contracts/integrators will put in place after that the `sell/buy` asset operation has been executed.

5. `SampleLiquidator` could end up with stuck `GHO` if `triggerBurnAfterSeize` is executed with `amount > gsmFacilitatorBucketLevel`

Aave has added a new function to the `SampleLiquidator` that allows the owner to trigger the `gsm.burnAfterSeize`

```
function triggerBurnAfterSeize(address gsm, uint256 amount) external onlyOwner {
  IERC20 ghoToken = IERC20(IGsm(gsm).GHO_TOKEN());
  ghoToken.transferFrom(msg.sender, address(this), amount);
  ghoToken.approve(gsm, amount);
  IGsm(gsm).burnAfterSeize(amount);
}
```

`amount` of `GHO` will be pulled from the `msg.sender` (owner) and `burnAfterSeize` will be executed on the `Gsm`

The `Gsm` will pull from the caller (the liquidator contract in this example) less than the `amount` of `GHO` passed if such amount is greater than the amount of `GHO` minted by the facilitator (the `Gsm`) itself

```
function burnAfterSeize(uint256 amount) external onlyRole(LIQUIDATOR_ROLE) returns (ui
    require(_isSeized, 'GSM_NOT_SEIZED');
    require(amount > 0, 'INVALID_AMOUNT');

    (, uint256 ghoMinted) = IGhoToken(GHO_TOKEN).getFacilitatorBucket(address(this));
    if (amount > ghoMinted) {
      amount = ghoMinted;
    }
    IGhoToken(GHO_TOKEN).transferFrom(msg.sender, address(this), amount);
    IGhoToken(GHO_TOKEN).burn(amount);

    emit BurnAfterSeize(msg.sender, amount, (ghoMinted - amount));
    return amount;
}
```

Because of this, if the `amount` passed to `triggerBurnAfterSeize` is greater than the amount of `GHO` minted by the `Gsm`, the difference between those amounts will be stuck in the `Liquidator` contract.

Aave should refactor the `triggerBurnAfterSeize` to check if the `amount` specified as the input parameter is greater of the amount of `GHO` that can be burned. If such amount is greater, they should be limited to `ghoMinted`.

```
function triggerBurnAfterSeize(address gsm, uint256 amount) external onlyOwner {
  IERC20 ghoToken = IERC20(IGsm(gsm).GHO_TOKEN());
  (, uint256 ghoMinted) = IGhoToken(address(ghoToken)).getFacilitatorBucket(address(th
+  if (amount > ghoMinted) {
+    amount = ghoMinted;
+  }

  ghoToken.transferFrom(msg.sender, address(this), amount);
  ghoToken.approve(gsm, amount);
  IGsm(gsm).burnAfterSeize(amount);
}
```

Discussion

Security Researcher

The point 1, 3 and 5 of the report has been addressed by the commit a5ac4d25a150ec49bfc8a4f54061d5c1e4d0b016 (included in the PR 369)

Aave

For 2, we acknowledge it.

For 4, we acknowledge the issue, but also note that the documentation around `buyAsset` and `sellAsset` in `IGsm` makes it explicit that input parameters represent minimum/maximum amounts (and are named as such) and, thus, may deviate (we do not advertise it in our docs as an exact input). We also return the exact amount that was bought/sold for those functions to ensure the amount of an asset bought/sold can be known without guesswork. Integrators need to ensure they adhere to the interface provided, the same as for any other contract.

[Fix Review Period] `OracleSwapFreezer` considerations

Note: The `OracleSwapFreezer` is not part of the original audit scope and has been introduced during the audit period.

1. Consider to "unbundle" `_freezeBound` and `_unfreezeBound` to be stored as `immutable` variables.

These variables can't be changed in the codebase, and there is no reason to "waste" gas by interacting with them as state variables.

Aave should consider removing the `struct Bound` and replacing `_freezeBound` with `_freezeLowerBound + _freezeUpperBound` and `_unfreezeBound` with `_unfreezeLowerBound + _unfreezeUpperBound`.

Aave should also require that if `_allowUnfreeze == false`, both `_unfreezeLowerBound` and `_unfreezeUpperBound` are equal to `0` in order to make the `getUnfreezeBound` returned value coherent with the `_allowUnfreeze` flag. This check can be added directly in the `constructor` or in the `_validateBounds` function.

2. Consider declaring the `_allowUnfreeze` as an `immutable` variable. The current implementation of `OracleSwapFreezer` is never changing the value of such variable.

3. The `OracleSwapFreezer` is not taking in consideration the `seize` state of the `Gsm`

If the `Gsm` has been seized, the swap operations `buyAsset*` and `sellAsset*` cannot already be performed. If the `Gsm` is in the seized state, the `OracleSwapFreezer._getAction` should always return `Action.NONE` without performing any further check.

4. Aave should document the edge case where the value returned by the Price Oracle returns `0`. Which are the scenarios when this event happens? How should the `OracleSwapFreezer` and the `Gsm` react to such a scenario? The current implementation of the `OracleSwapFreezer._isActionAllowed` will return `false` and no action will be performed.
5. The `_isActionAllowed` should be documented in a clear way. The `UNFREEZE` action is taken when the price is **within** the `_unfreezeBound` boundaries, instead the `FREEZE` action is taken when the price is **outside** the `_freezeBound` boundaries. The current natspec `@return` description is only taking in consideration the "within a boundary" scenario. The documentation should also document how the `OracleSwapFreezer` acts (do not perform any action) when the price is **between** those two boundaries (`_freezeBound.lowerBound < price < _unfreezeBound.lowerBound || _unfreezeBound.upperBound < price < _freezeBound.upperBound`).
6. Aave should lock the pragma version used by the contract to the same one used by the other contracts in the project (`0.8.10`)
7. If the `ADDRESS_PROVIDER.getPriceOracle()` is using ChainLink as the main source of asset's price, Aave should take in consideration to further validate the answer returned by the ChainLink Oracle. There are edge cases where the ChainLink Price Oracle could return **stale** prices. In such scenarios, the `OracleSwapFreezer` contract could base their action on an invalid price and as a consequence freeze, unfreeze or do nothing while another action should have been taken.

Discussion

Security Researcher

The recommendations for the points 1, 2, 3, 4, 5 and 6 have been implemented in the commit a5ac4d25a150ec49bfc8a4f54061d5c1e4d0b016 (included in the PR 369)

Aave has acknowledged the point 7 with the following message

> we are intentionally using the Aave Oracle, not the ChainLink oracles directly, and so are implicitly ack'ing the use in the Aave Oracle of `latestAnswer`