certora

# Security Assessment

aave

# Remote GSM

July-2025

*Prepared for:*
**Aave DAO**

*Code developed by:*

## Table of contents

# Project Summary

## Project Scope

| Project Name | Repository (link) | Latest Commit Hash | Platform |
|---|---|---|---|
| Remote GSM | https://github.com/aave-dao/gho-origin | 6ea6b0 | EVM |

## Project Overview

This document describes the security review of **Aave Remote GSM** code using manual code review. The work was undertaken from **July 7th** to **July 15th, 2025**

The following contracts are considered in scope for this review:

- `src/contracts/facilitators/gsm/GhoReserve.sol`
- `src/contracts/facilitators/gsm/Gsm.sol`
- `src/contracts/facilitators/gsm/Gsm4626.sol`
- `src/contracts/facilitators/gsm/OwnableFacilitator.sol`
- `src/contracts/facilitators/gsm/interfaces/*`

The team performed a manual audit of all the solidity contracts. Issues discovered during the review are listed in the following pages.

## Protocol Overview

GHO is a decentralised, overcollateralised stablecoin that is fully backed, transparent, and native to the Aave Protocol. In addition to the collateral backing GHO loans, GHO Stability Modules (GSMs) have been deployed to keep GHO pegged to various tokens (specifically, stablecoins and ERC4626 vaults using stablecoins). This gives additional guarantees of GHO's stable value, ensuring the financial security of GHO to holders.

The **Aave Remote GSM** will bring GSMs to other networks, enabling lower gas usage for buying/selling GHO against other stable assets and easy access to GHO. This removes the necessity to bridge GHO from Ethereum Mainnet as is the case currently.

The new remote GSM is built using a new facilitator on the mainnet to mint the GSM’s funds (in GHO), and a GHO reserve to hold the GHO on the relevant L2, as GHO cannot be (originally) minted/burned outside the mainnet. Minor adjustments have been made to the GSMs (and GSM4626s) to pull and return funds to the GHO reserve, instead of directly minting and burning from GHO’s ERC20 contract.

## Findings Summary

The table below summarizes the findings of the review, including type and severity details.

| Severity | Discovered | Confirmed | Fixed |
|---|---|---|---|
| Critical | - | - | - |
| High | 1 | 1 | 1 |
| Medium | 1 | 1 | 1 |
| Low | 1 | 1 | 1 |
| Informational | - | - | - |
| **Total** | 3 | 3 | 3 |

## Severity Matrix

| | | | | |
|---|---|---|---|---|
| **Impact** | High | Medium | High | Critical |
| | Medium | Low | Medium | High |
| | Low | Low | Low | Medium |
| | | Low | Medium | High |
| | | **Likelihood** | | |

# Detailed Findings

## Audit Goals

1. **Correct Integration of GhoReserve**
   a. Ensure that all facilitator interactions with `GhoReserve` via `use()` and `restore()` maintain accurate system-wide GHO accounting. This includes enforcing entity-specific limits and keeping `.used` balances aligned with actual GHO flows, both during asset swaps and in edge cases like fee distribution.
   b. Ensure all facilitator interactions use the `GhoReserve`, and there are no lingering calls to `mint()` and `burn()`.

2. **Facilitator Lifecycle Safety**
   a. Ensure that facilitator onboarding to the `GhoReserve`, and removal via `addEntity()` and `removeEntity()`, are executed cleanly, without leaving behind stale usage or credit, and follow a predictable and safe sequence of operations.

3. **Controlled Minting via OwnableFacilitator**
   a. Ensure that minting and burning of GHO through the OwnableFacilitator is properly access-controlled, consistent with intended ownership flows, and does not introduce accounting discrepancies. Confirm that only authorized entities can trigger supply changes, and that the facilitator's logic aligns with expected system behavior across deployment and operational scenarios.

4. **Protocol Invariants and Trust Boundaries**
   a. Validate that core invariants of the GHO system, such as supply controls, facilitator permissions, and usage caps, are upheld consistently across contracts. Ensure that trust boundaries (e.g., owner-only functions, facilitator roles, and reserve authority) are clearly enforced and not implicitly bypassed through indirect flows or edge cases.

## Coverage and Conclusions

1. **Correct Integration of GhoReserve**
    a. Facilitator interactions with the `GhoReserve` contract were reviewed to ensure correct use of the `use()` and `restore()` functions, with a focus on enforcing entity credit limits and maintaining accurate `.used` accounting in line with GHO flows. This included both user-facing flows and internal flows (e.g. fee distribution), which were analyzed to confirm they do not unintentionally inflate usage or disrupt Reserve accounting. Simulations validated that entity limits are respected and reserve usage remains in sync with actual GHO movement.
    b. All facilitator interactions which used to directly interact with the GHO token were reviewed, and all uses of `` `mint()` `` and `` `burn()` `` are now routed through the `` `GhoReserve` `` contract (via `` `use()` `` and `` `restore()` ``).

2. **Facilitator Lifecycle Safety**
    a. The onboarding (`addEntity`) and offboarding (`removeEntity`) flows in the `GhoReserve` were analyzed to ensure facilitators cannot retain access or leave behind stale credit or usage records. Removal conditions were tested under various edge cases, including sequencing of limit resets and usage cleanup. The flows follow a predictable and safe structure, preventing misuse or inconsistent states during facilitator deactivation.

3. **Controlled Minting via OwnableFacilitator**
    a. The `OwnableFacilitator` contract was examined to confirm that only the designated owner is authorized to mint or burn GHO. Minting actions are traceable, permissioned, and restricted to mainnet deployment, in line with system design. The `burn()` behavior was inspected to ensure it aligns with accounting expectations and does not introduce inconsistencies in Reserve tracking. Access control via `Ownable` is correctly enforced across all sensitive functions.

4. **Protocol Invariants and Trust Boundaries**
    a. Core invariants, including facilitator credit caps, reserve usage alignment, and trusted actor roles were validated across the system. Each privileged function was reviewed for appropriate access control, and ownership boundaries were tested for correct enforcement. No unintended flows or indirect interactions were found that

bypass facilitator or reserve limits. Invariants held under both normal and adversarial conditions, with attention to cross-contract trust assumptions.

## High Severity Issues

### H-01 Incomplete Deactivation of GSM Modules in GhoReserve

| Severity: **High** | Impact: **High** | Likelihood: **Medium** |
|---|---|---|
| Files: contracts/facilitators/ GHOReserve.sol | Status: Fixed | |

**Description:**

The `removeEntity()` function in the `GhoReserve` contract is intended to fully revoke a GSM's (GHO Stabilizer Module's) access to the Reserve. It removes the entity from the internal `_entities` set, ensuring that governance can formally decommission any GSM.

```javascript
function removeEntity(address entity) external onlyOwner {
  require(_ghoUsage[entity].used == 0, 'ENTITY_GHO_USED_NOT_ZERO');
  require(_entities.remove(entity), 'ENTITY_NOT_REMOVED');

  emit EntityRemoved(entity);
}
```

However, removal **does not clear** the entity's entry in the `_ghoUsage` mapping. Critically, the `use()` and `restore()` functions do **not** verify whether the caller is still a recognized entity in `_entities`. This leads to a major gap: a GSM that has been "removed" by governance can **continue to interact with the Reserve** if it still has a remaining quota under its old `.limit` or `.used` value.

**Impact:**

This violates a key protocol invariant: only active, approved facilitators should be able to mint or return GHO. In its current form, any user can call `use()` or `restore()` on behalf of a GSM that has been removed but still has a non-zero `.limit` balance. This allows unintended or malicious interactions that affect GHO supply, effectively bypassing governance deactivation.

This means a removed GSM can continue drawing or repaying GHO from the Reserve, potentially distorting total supply and undermining system-level accounting guarantees.

Additionally, stale facilitator state remains in storage indefinitely, causing unnecessary bloat and making it harder to reason about which facilitators are truly active.

**Recommendation:**

- Add `require(_entities.contains(msg.sender), "NOT_ENTITY")` to both `use()` and `restore()`.
- Add delete `_ghoUsage[entity]` in `removeEntity()` to fully clear access and usage state.

**Customer's response:** Issue fixed in commit d064534.

**Fix Review:** Fix confirmed.

## Medium Severity Issues

| M-01. Outdated _accruedFees May Lead to Incorrect GHO Rescue in Gsm4626 | | |
|---|---|---|
| Severity: **Medium** | Impact: **Medium** | Likelihood: **Medium** |
| Files: contracts/facilitators/Gsm.sol | Status: Fixed | |

**Description:**
The rescueTokens() function protects protocol balances by only allowing withdrawal of GHO tokens not accounted for in _accruedFees:

```
JavaScript
uint256 rescuableBalance = IERC20(token).balanceOf(address(this)) - _accruedFees;
require(rescuableBalance >= amount, 'INSUFFICIENT_GHO_TO_RESCUE');
```

However, in Gsm4626, the _accruedFees value may be stale if _cumulateYieldInGho() hasn't been recently called. This function calculates excess ERC4626 vault yield and mints equivalent GHO via the Reserve, crediting the result to _accruedFees. If yield has accumulated but _cumulateYieldInGho() hasn't been run, _accruedFees will be underreported, meaning protocol-owned GHO may incorrectly appear as "rescuable."

**Impact:**

- Earned GHO may be transferred to arbitrary addresses via rescueTokens() instead of being distributed to the treasury.

**Recommendation:**

In Gsm4626, consider overriding rescueTokens() and invoking _cumulateYieldInGho() when token == GHO_TOKEN:

```
JavaScript
  if (token == GHO_TOKEN) {
  _cumulateYieldInGho();
}
```

**Customer's response:** The issue existed in the original GHO implementation. The documentation was updated in commit 7de25e9 to emphasize the correct order of actions.

**Fix Review:** Fix confirmed.

## Low Severity Issues

### L-01. Front-Running removeEntity() with use()

| Severity: **Low** | Impact: **Low** | Likelihood: **Low** |
|---|---|---|
| Files: contracts/facilitators/ GHOReserve.sol | Status: Fixed | |

**Description:**
The removeEntity() function in GhoReserve includes the following check:

```
JavaScript
require(_ghoUsage[entity].used == 0, 'ENTITY_GHO_USED_NOT_ZERO');
```

This ensures that an entity can only be removed after it has returned all borrowed GHO. However, there's a potential edge case where anyone could front-run the removeEntity() transaction by calling use() on that gsm (entity) just before it is mined. This would increase .used and cause the removal to fail.

That said, use() is guarded by:

```
JavaScript
require(entity.limit >= entity.used + amount, 'LIMIT_EXCEEDED');
```

This only prevents front-running if the entity's limit has already been set to zero. There is no guarantee in the current implementation that this prerequisite step has occurred.

**Impact:**

A malicious actor could front-run removeEntity() with a use() call, causing it to fail and making the offboarding process more fragile.

**Recommendation:**
The contract should enforce that both .used == 0 and .limit == 0 at the time of removal to guarantee safety:

```JavaScript
require(_ghoUsage[entity].limit == 0, 'ENTITY_GHO_LIMIT_NOT_ZERO');
```

This would enforce that governance (or the owner) must explicitly set the entity's limit to zero before calling removeEntity(), eliminating ambiguity and ensuring no further GHO can be drawn during the removal process.

**Customer's response:** Fixed in commit c172c2a.

**Fix Review:** Fix confirmed.

## Disclaimer

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.

## About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects, and incident response.