Pashov Audit Group

TokenLogic Security Review

January 27th 2026 - January 29th 2026

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of 40+ freelance security researchers, who are well proven in the space - most have earned over $100k in public contest rewards, are multi-time champions or have truly excelled in audits with us. We only work with proven and motivated talent.

With over 300 security audits completed — uncovering and helping patch thousands of vulnerabilities — the group strives to create the absolute very best audit journey possible. While 100% security is never possible to guarantee, we do guarantee you our team's best efforts for your project.

Check out our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We 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.

3. Risk Classification

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

Impact

• **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users
• **Medium** - leads to a moderate material loss of assets in the protocol or moderately harms a group of users
• **Low** - leads to a minor material loss of assets in the protocol or harms a small group of users

Likelihood

• **High** - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost
• **Medium** - only a conditionally incentivized attack vector, but still relatively likely
• **Low** - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive

4. About TokenLogic

Aave Proposals in two PRs introduce a full migration from existing GSM contracts to new RemoteGSM contracts on Ethereum—seizing assets, deploying updated GHO facilitator/reserve components, registering new automation keepers, and updating interfaces while transferring all liquidity and configurations to the upgraded system. They also launch GHO on the Plasma network through a coordinated multi-step bridge deployment, configure CCIP rate limiters, deploy GHO as a lending asset with tailored risk parameters, establish new GSM and e-Mode setups, and assign governance roles for the new cross-chain GHO ecosystem.

5. Executive Summary

A time-boxed security review of the **bgd-labs/aave-proposals-v3** and **bgd-labs/aave-proposals-v3** repositories was done by Pashov Audit Group, during which **JCN, Klaus, Said, 0x37, silver_eth, 0xl33** engaged to review **TokenLogic**. A total of **5** issues were uncovered.

Protocol Summary	
Project Name	TokenLogic
Protocol Type	Contract migration
Timeline	January 27th 2026 - January 29th 2026

Review commit hashes:

- 07c6ca8cc440a0b9fd2f17fc41a8b3c0f2e57366 (bgd-labs/aave-proposals-v3)
- 616f44ba08f2be8a88af2c4f2a807cf3670660e2 (bgd-labs/aave-proposals-v3)

Fixes review commit hash:

- f8a05d6e7afa067018bc35bcb00a9e790457602f (bgd-labs/aave-proposals-v3)

Scope

`AaveV3Ethereum_GSMMigration_20251113.sol`
`AaveV3Ethereum_LaunchGHOOnPlasmaSetACIAsEmissionsManagerForRewards_20250930_Part1.sol`
`AaveV3Ethereum_LaunchGHOOnPlasmaSetACIAsEmissionsManagerForRewards_20250930_Part2.sol`
`AaveV3Plasma_LaunchGHOOnPlasmaSetACIAsEmissionsManagerForRewards_20250930_Part1.sol`
`AaveV3Plasma_LaunchGHOOnPlasmaSetACIAsEmissionsManagerForRewards_20250930_Part2.sol`
`IGhoDirectFacilitator.sol` `IGhoReserve.sol` `IGsm.sol`

6. Findings

Findings count

Severity	Amount
Medium	1
Low	4
Total findings	**5**

Summary of findings

ID	Title	Severity	Status
[M-01]	Migration payload will fail given current contract state	Medium	**Resolved**
[L-01]	Draw limit does not align with exposure cap or consider yield accrued	Low	**Resolved**
[L-02]	GSM migration shortfall coverage depends on collector GHO balance	Low	Acknowledged
[L-03]	The migration payload can be blocked via token donation	Low	**Resolved**
[L-04]	GSM includes ERC4626 fees in price calculation using `FixedPriceStrategy4626`	Low	Acknowledged

Medium findings

[M-01] Migration payload will fail given current contract state

Severity

Impact: Medium

Likelihood: Medium

Description

The draw limit for the `NEW_GSM_USDT` is hardcoded at `50_000_000` GHO. However, the exposure cap for the old `GSM_USDT` was recently updated and the current exposure of that GSM now sits near the cap, close to `55 Mil` at the time of writing this report.

This translates to approximately `63 Mil` GHO.

This can be validated by supplying the current exposure of the old GSM to the `getGhoAmountForSellAsset` API of the new GSM.

As a result, selling the `55 Mil` underlying into the new GSM will result in the GSM attempting to draw over its configured limit, causing the migration payload to revert.

Recommendations

Consider calculating the necessary draw limits dynamically in order to take into account sudden changes in the exposure cap for the old GSMs.

Low findings

[L-01] Draw limit does not align with exposure cap or consider yield accrued

Description

The draw limit for the `NEW_GSM_USDC` is hardcoded at `100_000_000 M` GHO. This draw limit should at least align with the exposure cap, which is the maximum amount of underlying that can be sold into the GSM to purchase up to this GHO limit. The exposure cap for the old GSM is `87 M`. However, when selling this amount of underlying into the new GSM, the amount of GHO to draw will be approximately `100_976_907`. **This can be validated by supplying `87000000000000` to the `getGhoAmountForSellAsset` API of the new GSM.**

As a result, the effective exposure cap for the `NEW_GSM_USDC` will be lower than the configured cap. In this case, it would be approximately `8_6100_000`, about `900_000` underlying less than the actual cap.

More importantly, the `NEW_GSM_USDC` is a `Gsm4626` and therefore the underlying assets will accrue yield as time passes. Therefore, the draw limit should also consider yield that accrues while the exposure cap is hit. However, as we have seen, the draw limit will not even accommodate the actual cap and while the effective exposure cap is hit, no yield can be collected by the treasury:

```
  function distributeFeesToTreasury() public override(Gsm, IGhoFacilitator) {
    _cumulateYieldInGho();
    super.distributeFeesToTreasury();
  }

...

  function _cumulateYieldInGho() internal {
    (uint256 ghoLimit, uint256 ghoUsed) = _getUsage();
    uint256 ghoAvailableToMint = ghoLimit > ghoUsed ? ghoLimit - ghoUsed : 0;
    (uint256 ghoExcess, ) = _getCurrentBacking(ghoUsed);
    if (ghoExcess > 0 && ghoAvailableToMint > 0) {
      ghoExcess = ghoExcess > ghoAvailableToMint ? ghoAvailableToMint : ghoExcess;
      _accruedFees += uint128(ghoExcess);
      IGhoReserve(_ghoReserve).use(ghoExcess);
    }
  }
```

It is important to note the only way normal user operations can reduce the `_currentExposure` is via `buyAsset` calls, where the user restores GHO in exchange for underlying. However, for `Gsm4626` contracts, there is a `_beforeBuyAsset` hook that invokes `_cumulateYieldInGho` before the exposure is updated:

```
function _buyAsset(
  address originator,
  uint256 minAmount,
  address receiver
) internal returns (uint256, uint256) {
  (
    uint256 assetAmount,
    uint256 ghoSold,
    uint256 grossAmount,
    uint256 fee
  ) = _calculateGhoAmountForBuyAsset(minAmount);

  _beforeBuyAsset(originator, assetAmount, receiver);
```

```
function _beforeBuyAsset(address, uint256, address) internal override {
  _cumulateYieldInGho();
}
```

Therefore, once the effective exposure cap is hit and yield has accrued, the GSM will be bricked until governance mints more GHO to the reserve and increases the GSM's draw limit accordingly.

Recommendation

Consider increasing the draw limit to include some headroom for yield accrued while the exposure cap is hit.

For example, the legacy `GSM_USDT` is very close to its exposure cap, which translates to about `63_475_884` GHO at the time of writing this report. The GSM's bucket capacity is currently `65_000_000` GHO (can be verified here by supplying the GSM's address: `0x535b2f7C20B9C83d70e519cf9991578eF9816B7B`). This extra headroom allows about `1_524_116` GHO in yield to accrue and be collected while the cap is hit.

[L-02] GSM migration shortfall coverage depends on collector GHO balance

Description

In `AaveV3Ethereum_GSMMigration_20251113.sol`, the `_fund()` function sells seized `stataTokens` into new GSMs and uses the acquired GHO to burn old GSM debt. If the GHO acquired from selling is less than the old GSMs' outstanding debt, the shortfall is covered by transferring GHO from the `Collector`.

```
// ...
uint256 acquiredGho = amountGhoUsdc + amountGhoUsdt;
uint256 mintedGho = ghoUsdcNeeded + ghoUsdtNeeded;

if (mintedGho > acquiredGho) {
    AaveV3Ethereum.COLLECTOR.transfer(
        IERC20(GhoEthereum.GHO_TOKEN),
        address(this),
        mintedGho - acquiredGho    // shortfall
    );
}
// ...
```

The shortfall is covered by transferring GHO from the `Collector`. This creates an implicit dependency on the `Collector` holding sufficient GHO at execution time. If the Collector's GHO balance is insufficient, the `COLLECTOR.transfer` reverts.

Document the minimum `Collector` GHO balance requirement as a pre-condition for migration execution. Alternatively, mint a small buffer of additional GHO through the `DirectFacilitator` to cover expected fees, avoiding dependence on the `Collector`'s balance at execution time.

[L-03] The migration payload can be blocked via token donation

Description

The legacy `GSM_USDT` contract is currently operating very close to its exposure cap. At the time of writing this report, the contract state is as follows:

- Current exposure: 54,999,977 underlying.
- Exposure cap: 55,000,000 underlying.

The migration payload `AaveV3Ethereum_GSMMigration_20251113` determines how much underlying to migrate by reading the entire token balance of the old GSM:

```
    uint256 balanceUsdt = IERC20(AaveV3EthereumAssets.USDT_STATA_TOKEN).balanceOf(
      GhoEthereum.GSM_USDT
    );
```

It then attempts to sell this full amount into the new GSM:

```
    (, uint256 amountGhoUsdt) = IGsm(NEW_GSM_USDT).sellAsset(balanceUsdt, address(this));
```

However, the new GSM enforces its exposure cap during `sellAsset`:

```
    require(_currentExposure + assetAmount <= _exposureCap,
'EXOGENOUS_ASSET_EXPOSURE_TOO_HIGH');
```

Since the payload uses the raw token balance as the sell amount, an attacker can increase this balance before execution by transferring (donating) underlying tokens to the old `GSM_USDT`.

If this donation causes the balance of the old GSM to go above the exposure cap, the sellAsset call reverts, causing the entire migration process to fail.

At the time of writing this report, an attacker would need to donate at least `23e6` underlying to the contract to cause this impact.

Note: This exploit is currently shadowed by the fact that the migration payload will revert due to the hardcoded draw limit. However, this report still illustrates a vulnerability that can affect the migration even if the draw limits were configured properly. Recent on-chain activities suggest that the conditions for this exploit have a high likelihood of being reached in practice: Approximately 12 seconds after the exposure cap was updated, an MEV bot began selling into the old GSM and has since increased the exposure very close to the limit.

Recommendation

Consider using the `_currentExposure's` of the old GSMs as the sell amounts instead of the entire underlying balance of the contracts.

[L-04] GSM includes ERC4626 fees in price calculation using `FixedPriceStrategy4626`

Description

When using Gsm4626, the price strategy used is FixedPriceStrategy4626. Since Gsm4626 inherits from the Gsm contract, functions defined in Gsm will also use FixedPriceStrategy4626.

`FixedPriceStrategy4626.getAssetPriceInGho` and `FixedPriceStrategy4626.getGhoPriceInAsset` include or exclude ERC4626's deposit/withdraw fees depending on the `roundup` parameter. In `getAssetPriceInGho`, `ERC4626.previewMint` returns an amount minus the ERC4626 deposit fee, while `ERC4626.convertToAssets` returns an amount before deducting fees. `ERC4626.previewWithdraw` returns an amount minus the withdrawal fee, while `ERC4626.convertToShares` returns an amount before deducting fees.

For details on ERC4626 functions and whether deposit/withdrawal fees are included, refer to the EIP-4626 document.

```
  function getAssetPriceInGho(uint256 assetAmount, bool roundUp) external view returns
(uint256) {
    // conversion from 4626 shares to 4626 assets
    uint256 vaultAssets = roundUp
@>      ? IERC4626(UNDERLYING_ASSET).previewMint(assetAmount) // round up
@>      : IERC4626(UNDERLYING_ASSET).convertToAssets(assetAmount); // round down
    return
      vaultAssets.mulDiv(
        PRICE_RATIO,
        _underlyingAssetUnits,
        roundUp ? Math.Rounding.Up : Math.Rounding.Down
      );
  }

  function getGhoPriceInAsset(uint256 ghoAmount, bool roundUp) external view returns (uint256) {
    uint256 vaultAssets = ghoAmount.mulDiv(
      _underlyingAssetUnits,
      PRICE_RATIO,
      roundUp ? Math.Rounding.Up : Math.Rounding.Down
    );
    // conversion of 4626 assets to 4626 shares
    return
      roundUp
@>      ? IERC4626(UNDERLYING_ASSET).previewWithdraw(vaultAssets) // round up
@>      : IERC4626(UNDERLYING_ASSET).convertToShares(vaultAssets); // round down
  }
}
```

The problem is that Gsm calls the price strategy function without considering the ERC4626 fees. The underlying asset registered in Gsm4626 is an ERC4626 share token. Since deposits and withdrawals are not made directly to ERC4626 on Gsm but are only used in the form of share tokens, Gsm's price calculation does not need to include ERC4626 deposit and withdrawal fees. However, Gsm is unintentionally applying ERC4626 deposit and withdrawal fees in its calculations.

For example, in `Gsm._calculateGhoAmountForBuyAsset`, it calls `FixedPriceStrategy4626.getAssetPriceInGho` with `roundup` set to `true`. This uses `ERC4626.previewMint`, meaning it returns an amount deducting the ERC4626 deposit fee. In `Gsm._calculateGhoAmountForSellAsset`, it calls `FixedPriceStrategy4626.getGhoPriceInAsset` with `roundup` set to `true`. This uses `ERC4626.previewWithdraw`, returning an amount deducting the ERC4626 withdrawal fee.

```
  function _calculateGhoAmountForBuyAsset(
    uint256 assetAmount
  ) internal view returns (uint256, uint256, uint256, uint256) {
    bool withFee = _feeStrategy != address(0);
    // pick the highest GHO amount possible for given asset amount
@>  uint256 grossAmount = IGsmPriceStrategy(PRICE_STRATEGY).getAssetPriceInGho(assetAmount,
true);
    ...
  }

  function _calculateGhoAmountForSellAsset(
    uint256 assetAmount
  ) internal view returns (uint256, uint256, uint256, uint256) {
    ...
    // pick the highest asset amount possible for given GHO amount
@>  uint256 finalAssetAmount = IGsmPriceStrategy(PRICE_STRATEGY).getGhoPriceInAsset(
      finalGrossAmount,
      true
    );
    uint256 finalFee = finalGrossAmount - ghoBought;
    return (finalAssetAmount, finalGrossAmount - finalFee, finalGrossAmount, finalFee);
  }
```

Recommendations

The problem occurs when the Gsm contract inherited by Gsm4626 uses FixedPriceStrategy4626. You can take the following approach.

- Override all functions in the Gsm contract that use `PRICE_STRATEGY` in Gsm4626 and modify them to be compatible with FixedPriceStrategy4626.
- Or, in FixedPriceStrategy4626, return values that do not apply ERC4626 deposit and withdrawal fees.