Blackthorn

Security Review For **Aave Labs**

Collaborative Audit Prepared For:	**Aave Labs**
Lead Security Expert(s):	**0x52**
	bughuntoor
	mstpr-brainbot
	pkqs90
	xiaoming90
Date Audited:	**October 6 - October 20, 2025**

Introduction

Aave V4 introduces a new Hub and Spoke architecture. This architecture introduces several new design paradigms to Aave, each with their own set of benefits.

Capital is no longer fragmented across markets on the same chain. Instead, all liquidity flows through Liquidity Hubs, which increases utilization and unlocks better rates for both suppliers and borrowers. Governance also becomes lighter. New Spokes can be added or upgraded without disrupting the rest of the system.

To price risk more accurately, V4 also introduces Risk Premiums. This system ties borrowing rates to the quality of the collateral provided. Users with 'safer' collateral receive better rates, which creates a more efficient market where users' borrow rates better reflect actual risk.

Protocol Contacts

- Adam Schoeman (CISO, Aave Labs)
- Emilio Frangella (SVP of Engineering, Aave Labs)
- Stani Kulechov (CEO, Aave Labs)

Scope

Repository: aave/aave-v4

Audited Commit: dc31f9a4d54c0503093ef6939e6e8a8d2586709d

Final Commit: 91bb988323a635cdf67c1fb36d039306a8a6528b

Files:

- src/hub/AssetInterestRateStrategy.sol
- src/hub/HubConfigurator.sol
- src/hub/Hub.sol
- src/hub/interfaces/IAssetInterestRateStrategy.sol

- src/hub/interfaces/IBasicInterestRateStrategy.sol
- src/hub/interfaces/IHubBase.sol
- src/hub/interfaces/IHubConfigurator.sol
- src/hub/interfaces/IHub.sol
- src/hub/libraries/AssetLogic.sol
- src/hub/libraries/SharesMath.sol
- src/interfaces/IMulticall.sol
- src/interfaces/IRescuable.sol
- src/libraries/math/MathUtils.sol
- src/libraries/math/PercentageMath.sol
- src/libraries/math/WadRayMath.sol
- src/libraries/types/EIP712Types.sol
- src/libraries/types/Roles.sol
- src/misc/UnitPriceFeed.sol
- src/position-manager/interfaces/INativeTokenGateway.sol
- src/position-manager/interfaces/INativeWrapper.sol
- src/position-manager/interfaces/ISignatureGateway.sol
- src/position-manager/NativeTokenGateway.sol
- src/position-manager/SignatureGateway.sol
- src/spoke/AaveOracle.sol
- src/spoke/instances/SpokeInstance.sol
- src/spoke/interfaces/IAaveOracle.sol
- src/spoke/interfaces/IPriceOracle.sol
- src/spoke/interfaces/ISpokeBase.sol

- src/spoke/interfaces/ISpokeConfigurator.sol
- src/spoke/interfaces/ISpoke.sol
- src/spoke/interfaces/ITreasurySpoke.sol
- src/spoke/libraries/KeyValueList.sol
- src/spoke/libraries/LiquidationLogic.sol
- src/spoke/libraries/PositionStatusMap.sol
- src/spoke/SpokeConfigurator.sol
- src/spoke/Spoke.sol
- src/spoke/TreasurySpoke.sol
- src/utils/Multicall.sol
- src/utils/Rescuable.sol

Administrative Note

The security review involved developing an understanding of the protocol's design, attack surfaces, and risk areas, followed by a review of the smart contracts to identify issues or deviations from security best practices. Identified issues were documented and communicated to the protocol team, after which implemented fixes were reviewed to verify that the issues have been remediated appropriately.

Final Commit Hash

91bb988323a635cdf67c1fb36d039306a8a6528b

Findings

Each issue has an assigned severity:

- High issues are directly exploitable security vulnerabilities that need to be fixed.

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Low/Info issues are non-exploitable, informational findings that do not pose a security risk or impact the system's integrity. These issues are typically cosmetic or related to compliance requirements, and are not considered a priority for remediation.

Issues Found

High	Medium	Low/Info
0	2	9

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Security Experts Dedicated to This Review

@0x52

@bughuntoor

@mstpr-brainbot

@pkqs90

@xiaoming90

Issue M-1: Malicious users can take advantage of the rounding direction of a high-value & low-decimal asset

Source: https://github.com/sherlock-audit/2025-10-aave-v4-oct-6th/issues/103

Vulnerability Detail

Executing `updateUserRiskPremium()` function consumes around 63556 gas.

The current gas price at the time of writing this report (18 Oct 2025) on the Base chain is 0.003 Gwei. Assume that the average ETH price is around 3500 USD. In this case, the cost for executing the transaction will be around 0.0007 USD.

```
uint256 gasBefore = gasleft();
spoke1.updateUserRiskPremium(alice);
uint256 gasAfter = gasleft();
console.log("Gas used:", gasBefore - gasAfter);

Result: Gas used: 63556
```

When an `updateUserRiskPremium()` transaction is executed, the premium debt of the hub's asset, the spoke's reserve, and the user position will be slightly inflated by up to 2 wei due to the internal rounding direction. This behavior has been documented in the comment in Line 699:

```
File: Hub.sol
681:   function _applyPremiumDelta(
..SNIP..
699:     // can increase due to precision loss on premium (drawn unchanged)
700:     require(asset.premium() + premiumAmount - assetPremiumBefore <= 2,
↪  InvalidPremiumChange());
701:     uint256 spokePremiumAfter = _getSpokePremium(spoke, assetId);
702:     require(spokePremiumAfter + premiumAmount - spokePremiumBefore <= 2,
↪  InvalidPremiumChange());
```

Note

Note 1: The `updateUserRiskPremium()` will internally trigger the `Spoke._notifyRiskPremiumUpdate() -> Hub.refreshPremium() -> Hub._applyPremiumDelta()`.

Note 2: Up to 2 wei is due to the first variable of the formula being rounded up (`hub.previewRestoreByShares(assetId, oldPremiumShares)`) while the second variable (`oldPremiumOffset`) is rounded down here.

Note 3: The `updateUserRiskPremium()` can continue to be executed even after the account's health factor is below 1e18 and the debt continues to increase by 1-2 wei

The following demonstrates the side effects of calling the `updateUserRiskPremium()` function repeatedly. The premium debt increased by 100,000 wei (From 7 to 100007) after running 100,000 iterations of `updateUserRiskPremium`. Note that in this specific example, it only increases by 1 wei in each call, but other examples can increase the debt by 2 wei after each call, as mentioned in the codebase's comment.

```
[PASS] test_repeatUpdateRisk() (gas: 6285631650)
Logs:
  ################### Initial State ###################
  ========== Print Debt Data ==========
  Alice's preimumDebt =  0
  Hub's assetPremium =  0
  Hub's totalAddedAssets =  15000
  Hub's totalAddedShares =  15000
  Alice's healthFactor =  10013333333333333333
  ################### Initial State after 7 days ###################
  ========== Print Debt Data ==========
  Alice's preimumDebt =  7
  Hub's assetPremium =  7
  Hub's totalAddedAssets =  15051
  Hub's totalAddedShares =  15001
  Alice's healthFactor =  9979403361902863597
  ################### End State after calling updateUserRiskPremium() 100,000
  ↪  times ###################
  Number of iterations ran =  100000
  ========== Print Debt Data ==========
  Alice's preimumDebt =  100007
  Hub's assetPremium =  100007
```

```
  Hub's totalAddedAssets =  115051
  Hub's totalAddedShares =  15001
  Alice's healthFactor =  1305507992107847824
```

The protocol supports asset tokens with decimals from 5 to 18.

```
File: Hub.sol
34:   /// @inheritdoc IHub
35:   uint8 public constant MIN_ALLOWED_UNDERLYING_DECIMALS = 5;
```

Assume that BTCX token (e.g., a new derivative asset of BTC) is five (5) decimal precision and worth 100,000 USD. In this case, 1 unit of BTCX is 1e5, and 1 wei is worth 1 USD. BTCX. The token has been added as a supported asset in the hub and a supported reserve asset in the spoke.

Assume that Bob owns a significant share of the total BTCX token supply in the hub, which he supplied via one of the spokes.

Let's say Bob currently owns 5% of the total shares for BTCX assets in the hub.

The value/price of each BTCX share is calculated based on the following formula:

```
totalAddedAssets = asset.liquidity + asset.swept + asset.deficit + (asset.drawn +
↪  asset.premium)
sharePrice = totalAddedAssets / totalAddedShares
```

In this case, Bob will open a small BTCX position in a spoke. For each `updateUserRiskPremium()` function he executed, the `totalAddedAssets` will increase by 1 wei of BTCX, which is equal to 1 USD in value. 1 BTCX (100_000 wei = 1e5) is worth 100,000 USDC, so 1 wei is worth 1 USD.

Since Bob owns 5% of the total shares, he will be entitled to 0.05 USD of the value, while he only spent 0.00067 USD on the gas fee (profitable after taking into account of the gas cost).

Bob repeatedly calls this function to inflate the value of his shares and then withdraws all his shares.

This should ideally be done within a single block or within a few blocks. Otherwise, liquidators might attempt to liquidate the position. Even if liquidation occurs, a

malicious user can create a new position and continue the attack.

If Bob enables his BTCX's supplied shares as collateral in his account, this would also increase its borrowing power since the value of his supplied shares increases. He could proceed to borrow more assets from the protocol.

POC

First, define a new token asset (BTCX) with 5 decimals and worth 100,000 USD in the `Base.t.sol` test file. Then, copy the following test function to the `Spoke.PositionManager.t.sol` test file.

```
function test_exploitRepeatUpdateRisk() public {
  uint256 btcxAssetId = spoke1.getReserve(_btcxReserveId(spoke1)).assetId;
  uint256 btcxReserveId = _btcxReserveId(spoke1); // BTCX's collateral factor = 78%

  // Bob (malicious user) own 100_000
  // Other random users own 900_000
  // Virtual shares = 1_000_000
  // Total shares = 2_000_000
  // Following add 1_000_000 shares to simulate the above setup
  deal(spoke1, btcxReserveId, address(this), 100_000);
  Utils.approve(spoke1, btcxReserveId, address(this), UINT256_MAX);
  Utils.supplyCollateral(spoke1, btcxReserveId, address(this), 100_000,
  ↪  address(this)); // Bob's position
  _openSupplyPosition(spoke1, btcxReserveId, 900_000); // positions of other users

  // Create a dust position for the sole purpose of triggering
  ↪  updateUserRiskPremium() to exploit the rounding error
  // Alice's address is also owned by Bob
  Utils.supplyCollateral(spoke1, _usdxReserveId(spoke1), alice, 10e6, alice); //
  ↪  supplies 10 USDX (worth 10 USD only)
  Utils.borrow(spoke1, btcxReserveId, alice, 7, alice); // Alice's HF is
  ↪  1000000012820512984

  uint256 valueOfBTCXShareBefore = hub1.previewRemoveByShares(btcxAssetId, 100_000);
  console.log("100_000 BTCX is worth (before exploit) = ", valueOfBTCXShareBefore);

  vm.warp(block.timestamp + 7 days); // warp 7 days so that there is debt to be
  ↪  accrue
```

```
  console.log("BTCX balance of hub1 (Before Attack) = ",
  ↪  tokenList.btcx.balanceOf(address(hub1)));

  vm.startPrank(alice);
  uint i = 0;
  for (; i < 100_000; i++) {
    spoke1.updateUserRiskPremium(alice);
  }
  vm.stopPrank();

  console.log("BTCX balance of hub1 (After 100_000 loop) = ",
  ↪  tokenList.btcx.balanceOf(address(hub1)));

  uint256 valueOfBTCXShareAfter = hub1.previewRemoveByShares(btcxAssetId, 100_000);
  uint256 gainInBTCX = valueOfBTCXShareAfter - valueOfBTCXShareBefore;
  console.log("100_000 BTCX is worth (after exploit) = ", valueOfBTCXShareAfter);
  console.log("Gain in BTCX after exploit = ", gainInBTCX); // Gain 5000

  uint256 withdrawBefore = tokenList.btcx.balanceOf(address(this));
  spoke1.withdraw(btcxReserveId, 105_000, address(this));
  console.log("Attacker received (in BTCX)= ",
  ↪  tokenList.btcx.balanceOf(address(this)) - withdrawBefore);

  console.log("BTCX balance of hub1 (After Attack) = ",
  ↪  tokenList.btcx.balanceOf(address(hub1)));
}
```

The test shows that increasing the premium debt by 100,000 wei (1e5) consumes 5173606869 gas, which costs around 55 USD based on the gas cost in the Base chain mentioned at the start of the report.

```
forge test --match-test test_exploitRepeatUpdateRisk -vvv

[PASS] test_exploitRepeatUpdateRisk() (gas: 5173606869)
Logs:
  100_000 BTCX is worth (before exploit) =  100000
  BTCX balance of hub1 (Before Attack) =  999993
```

```
BTCX balance of hub1 (After 100_000 loop) =  999993
100_000 BTCX is worth (after exploit) =  105000
Gain in BTCX after exploit =  5000
Attacker received (in BTCX)=  105000
BTCX balance of hub1 (After Attack) =  894993
```

After the attack, Bob gained 5000 wei of BTCX, worth 5000 USD, while spending only 55 USD on the gas fee, making this a profitable attack.

Impact

If a high-value, low-decimal asset is supported, a malicious user (or a group of majority shareholders) could potentially increase the value of their supplied shares or collateral profitably, even after accounting for gas costs on certain chains.

Malicious users with inflated share value will exit promptly, potentially leaving the remaining shareholders with shares not backed by any assets unless the entire debt in the malicious position is eventually repaid. However, this will not happen, as there is no incentive for the malicious user to repay.

Furthermore, as shown in the test result above, the health factor of the malicious position drops from 1.001 to 0.13 after the attack, indicating that bad debt and deficits will be created. Someone (likely the umbrella spoke) will have to incur a loss by covering the deficit.

Note

Impact (High) - Funds are directly or nearly directly at risk

Likelihood (Medium) - Might occur under specific conditions when a particular ERC20 token (High value, Low Decimal token) is used on the platform

Severity (High) based on V4 Threat Model - Vulnerability Severity Matrix

Code Snippet

https://github.com/sherlock-audit/2025-10-aave-v4-oct-6th/blob/d792069ed8a08ef5226d77a1299c9d34aa2e6a7b/aave__aave-v4/src/hub/Hub.sol#L699

Tool Used

Manual Review

Recommendation

Due to the current rounding direction, an additional 1 or 2 wei of assets is added to the debt, and the intention is to always favor the protocol by having the users (borrower) repay more debts instead of the other way round.

However, the issue here is that another group of protocol users (lenders) indirectly benefits from the extra 1 or 2 wei of assets given to them, which opens up the opportunity for some lenders to exploit this to their advantage.

One possible solution is to discard the excess premium debt dust (1-2 wei) resulting from rounding errors to ensure that neither the borrower nor the lenders benefit from the rounding direction.

Discussion

Aave Labs

Premium accounting is now calculated and stored with full precision (token decimals augmented with a Ray, i.e. 27 decimals) to avoid rounding issues from Ray math. As a result, a user's premium debt does not immediately change after any action that can be executed (such as `updateUserRiskPremium`), eliminating any impact that one user's actions could have on other users' positions in the system. PR: https://github.com/aave/aave-v4/pull/991 & https://github.com/aave/aave-v4/pull/907

Lucas | Sherlock

After internal discussion, this is being downgraded to Medium. Fix is confirmed with PRs #991 and #907.

Issue M-2: Significant loss of yield/gain due to dead shares if a high-value & low-decimal asset is supported

Source: https://github.com/sherlock-audit/2025-10-aave-v4-oct-6th/issues/111

This issue has been acknowledged by the team but won't be fixed at this time.

Vulnerability Detail

The codebase uses `VIRTUAL_ASSETS` and `VIRTUAL_SHARES` of `1e6` (1_000_000)

```
File: SharesMath.sol
11: library SharesMath {
12:   using MathUtils for uint256;
13:
14:   uint256 internal constant VIRTUAL_ASSETS = 1e6;
15:   uint256 internal constant VIRTUAL_SHARES = 1e6;
```

The formula for converting between shares and assets is as follows:

$$shares = assets \times \frac{totalShares + 1000000}{totalAssets + 1000000}$$

$$assets = shares \times \frac{totalAssets + 1000000}{totalShares + 1000000}$$

From the above formula, this means that there are 1,000,000 dead shares, and any yield/gain earned by these dead shares will be inaccessible and lost.

The protocol supports asset tokens with decimals from 5 to 18.

```
File: Hub.sol
34:   /// @inheritdoc IHub
35:   uint8 public constant MIN_ALLOWED_UNDERLYING_DECIMALS = 5;
```

It was observed that the smaller the decimals precision and higher value of a supported asset, the greater the loss.

Assume that the protocol adds a new asset that has 5 decimals precision (1e5) and is a high-value asset worth 100,000 USD per unit (e.g., a derivative of BTC). Let's call this asset BTCX in this example. In this case, 1 wei of BTCX token is worth 1 USD.

At the start, there are already 1,000,000 wei of dead BTCX shares. If the total assets supplied by all users are worth 100,000 USD (= 100,000 wei of BTCX), this means that if the yield (from gain in accrued debt or premium debt) is 1000 USD, 909 USD of them will be lost to the dead shares, which is close to 90% of the total yield.

```
* no of dead shares == VIRTUAL_ASSETS == 1e6 == 1_000_000

dead shares gain = (gain in asset token) * [(shares of dead shares) / (totalShares
↪  + VIRTUAL_SHARES)]
dead shares gain = 1000 USD * (1_000_000/1_100_000) = 909.0909091
```

For simplicity's sake, assume that 1 share is worth close to 1 asset. The loss only reduced to 10% when the total shares of users increased to 9,000,000 wei of BTCX shares and the total locked value of users supplied BTCX is 9 million USD. Even at a 30 million USD locked value, 3% will be lost.

Impact

If assets with low precision and high value are supported, a significant amount of yield/gain will be lost due to a large number of dead shares that occupy a substantial portion of the total shares.

Code Snippet

https://github.com/sherlock-audit/2025-10-aave-v4-oct-6th/blob/d792069ed8a08ef5226d77a1299c9d34aa2e6a7b/aave__aave-v4/src/hub/libraries/SharesMath.sol#L14

Tool Used

Manual Review

Recommendation

The issue is caused by a large number of dead shares (1e6) and having a 1:1 ratio (VIRTUAL_ASSETS = 1e6 and VIRTUAL_SHARES = 1e6).

Using dead shares is a common mitigation for inflation attack. However, Uniswap only uses a dead share of 1000 (1e3) compared to 1e6 here, which reduce the impact of the loss yield.

In addition, using virtual shares (dead shares) coupled with virtual assets is also a common mitigation for inflation attack. This approach is found in Openzepplin's ERC4626 library and also Morpho's SharesMathLib. However, the impact of this issue is less significant on them because their VIRTUAL_ASSETS is `1` instead of `1e6`. The VIRTUAL_SHARES scale up the number of shares by 1e6. Therefore, when the user supplies and mints the shares, a large number of shares will be minted to the users. Since the users own a significant portion of the total shares, less yield/gain will go to the dead shares.

Consider reducing the VIRTUAL_ASSETS/VIRTUAL_SHARES and/or adopt the approach of Openzepplin's ERC4626 library (VIRTUAL_SHARES=1e6, VIRTUAL_ASSET=1)

Discussion

Aave Labs

Taking into account the range of asset decimals that will be supported, using 1e6 virtual shares/assets is considered appropriate to mitigate potential risks from inflation techniques. Any interest accrued on these virtual shares is deemed acceptable to be locked in the system. Note that these funds are not lost, but simply not withdrawable by anyone, which also acts as a cushion for suppliers and provides confidence that they will not be the last ones withdrawing.

Lucas | Sherlock

The issue has been acknowledged by the team.

Issue L-1: Potential Out-of-Gas revert in `_calculateAndPotentiallyRefreshUserAccountData()` function

Source: https://github.com/sherlock-audit/2025-10-aave-v4-oct-6th/issues/101

Vulnerability Detail

The Spoke can support up to 65536 reserves/assets. It was observed that there are no restrictions on the number of assets that a user can borrow or supply.

The _calculateAndPotentiallyRefreshUserAccountData() function refreshes user data by looping through all the supported reserves in the spoke here and iterating over all of the user's collateral here.

However, the issue is that if there are a large number of reserves supported, coupled with user borrowing or supplying a large number of different assets, the loop will even go out of gas, and a revert will occur.

Impact

Any function that depends on `_calculateAndPotentiallyRefreshUserAccountData()` function will be affected and DOSed.

Code Snippet

https://github.com/sherlock-audit/2025-10-aave-v4-oct-6th/blob/d792069ed8a08ef5226d77a1299c9d34aa2e6a7b/aave__aave-v4/src/spoke/Spoke.sol#L678

Tool Used

Manual Review

Recommendation

Considering restricting the number of reserves supported and the number of assets that a user can borrow or supply.

Alternatively, consider documenting this risk to ensure that the spoke owner is aware of it and takes it into account when determining the appropriate number of reserves to configure for the spoke.

Discussion

Aave Labs

This limitation is now documented and must be considered by the Spoke owner when configuring reserves in the Spoke. Additionally, the SpokeConfigurator contract can enforce a maximum allowed number of reserves, acting as a safeguard when configuring the Spoke through it. PR: https://github.com/aave/aave-v4/pull/974

Lucas | Sherlock

Fix is confirmed with PR#974

Issue L-2: KeyValueList does not support edge case `key == 2**32-1, value == 0`

Source: https://github.com/sherlock-audit/2025-10-aave-v4-oct-6th/issues/102

Vulnerability Detail

The `KeyValueList` is a helper data structure to sort key in ascending order while value in descending order. The top 32 bits stores the key.

But when we're adding the kv pair, we allow key to be `2**32-1`. Then when we pack it, we reverse the bits. This means if we are adding a `key == 2**32-1, value == 0` kv pair, the actual data we are storing is 0.

This is buggy because we also use `data == 0` to check if the key is initialized or not, so we cannot differentiate between the two cases.

https://github.com/sherlock-audit/2025-10-aave-v4-oct-6th/blob/main/aave__aave-v4/src/spoke/libraries/KeyValueList.sol#L39

```
uint256 internal constant _KEY_BITS = 32;
uint256 internal constant _MAX_KEY = (1 << _KEY_BITS) - 1;

function add(List memory self, uint256 idx, uint256 key, uint256 value) internal
↪  pure {
  require(key <= _MAX_KEY && value <= _MAX_VALUE, MaxDataSizeExceeded());
  self._inner[idx] = pack(key, value);
}

function pack(uint256 key, uint256 value) internal pure returns (uint256) {
  return ((_MAX_KEY - key) << _KEY_SHIFT) | value;
}

function unpackKey(uint256 data) internal pure returns (uint256) {
  return _MAX_KEY - (data >> _KEY_SHIFT);
}
```

```
  function unpack(uint256 data) internal pure returns (uint256, uint256) {
@>  if (data == 0) return (0, 0);
    return (unpackKey(data), unpackValue(data));
  }
```

Impact

No real impact, because the KeyValueList is only used to store `(key=collateralRisk, value=userCollateralValue)` kv pairs. It is impossible for collateralRisk to hit `2**32-1`. However, it's still nice to fix.

Recommendation

```
  function add(List memory self, uint256 idx, uint256 key, uint256 value) internal
  ↪  pure {
-   require(key <= _MAX_KEY && value <= _MAX_VALUE, MaxDataSizeExceeded());
+   require(key < _MAX_KEY && value <= _MAX_VALUE, MaxDataSizeExceeded());
    self._inner[idx] = pack(key, value);
  }
```

Discussion

Aave Labs

Key collision has been prevented by adopting the recommendation. PR: https://github.com/aave/aave-v4/pull/901

Lucas | Sherlock

Fix is confirmed with PR#901.

Issue L-3: Bad debt reporting can be blocked or delayed by front-running

Source: https://github.com/sherlock-audit/2025-10-aave-v4-oct-6th/issues/104

This issue has been acknowledged by the team but won't be fixed at this time.

Vulnerability Detail

A position's bad debt will only be reported to the hub if the `isUserInDeficit` boolean is `True`.

```
File: Spoke.sol
354:     if (isUserInDeficit) {
355:       _reportDeficit(user);
356:     } else {
357:       // new risk premium only needs to be propagated if no deficit exists
358:       _notifyRiskPremiumUpdate(user,
 ↪  _calculateUserAccountData(user).riskPremium);
359:     }
```

The `isUserInDeficit` boolean will only be true if the `_evaluateDeficit()` function return `True`.

```
File: LiquidationLogic.sol
215:     return
216:       _evaluateDeficit({
217:         isCollateralPositionEmpty: isCollateralPositionEmpty,
218:         isDebtPositionEmpty: isDebtPositionEmpty,
219:         activeCollateralCount: params.activeCollateralCount,
220:         borrowedCount: params.borrowedCount
221:       });
```

It was observed that as long as the number of collateral assets (`activeCollateralCount`) that remain in the account is larger than one (1), which is equivalent to not the last collateral asset, the `_evaluateDeficit()` function will always return `False`.

```
File: LiquidationLogic.sol
473:   function _evaluateDeficit(
474:     bool isCollateralPositionEmpty,
475:     bool isDebtPositionEmpty,
476:     uint256 activeCollateralCount,
477:     uint256 borrowedCount
478:   ) internal pure returns (bool) {
479:     if (!isCollateralPositionEmpty || activeCollateralCount > 1) {
480:       return false;
481:     }
482:     return !isDebtPositionEmpty || borrowedCount > 1;
483:   }
```

Assume that there is a position that suffers a massive bad debt and the account enables two collateral assets (USDT and USDC).

A malicious user can block or grief the protocol by preventing their bad positions from being reported as bad debt via the `_reportDeficit()` by front-running. When the liquidator attempts to liquidate against its last collateral asset (USDT), a malicious user can front-run the transaction by depositing 1 wei of another collateral asset (USDC).

The `Spoke.supply()` does not enforce the minimum supply amount, thus 1 wei of collateral will be accepted.

When the liquidation transaction is executed, the `activeCollateralCount` will be two (2), and the `_reportDeficit()` will be skipped. The system will consider a collateral to be active as long as there is 1 wei of collateral assets.

Impact

If bad debt reporting is blocked or delayed, it might cause bad debt to accumulate in the protocol. Bad debt has to be reported in a timely manner so that it can be eliminated from the system via the `eliminateDeficit()` function.

Code Snippet

https://github.com/sherlock-audit/2025-10-aave-v4-oct-6th/blob/d792069ed8a08ef5226d77a1299c9d34aa2e6a7b/aave__aave-v4/src/spoke/libraries/LiquidationLogic.sol#L473

Tool Used

Manual Review

Recommendation

Consider reporting the deficit as long as the value of the collateral is worth less than the value of the debt, regardless of the number of `activeCollateralCount`. Otherwise, consider documenting this behavior.

Discussion

Aave Labs

The deficit reporting flow operates as intended, accounting for deficits when the liquidated collateral is effectively exhausted, no more collateral assets in the position, and either outstanding debt remains in the liquidated debt asset or multiple debt assets persist in the position. While deficit could also be considered when there are more than one collateral asset in the position and only a single debt asset exists, this scenario is intentionally excluded due to the additional complexity and gas overhead required to support it. As a resolution procedure, a user position can be liquidated until only a single collateral asset remains, at which point the deficit is fully reported and the position is cleared. Although such liquidations are not economically profitable, they allow the protocol to progressively resolve the position and maintain accurate accounting over time.

Lucas | Sherlock

The issue has been acknowledged by the team.

Issue L-4: Assets with `assetId` above `65536` cannot be added to spoke

Source: https://github.com/sherlock-audit/2025-10-aave-v4-oct-6th/issues/105

This issue has been acknowledged by the team but won't be fixed at this time.

Vulnerability Detail

Per Line 104 below, assets with `assetId` above `65536` will be rejected. Thus, if there are any assets with an asset ID above 65536, they cannot be added to the spoke.

```
File: Spoke.sol
096:   function addReserve(
097:     address hub,
098:     uint256 assetId,
099:     address priceSource,
100:     ReserveConfig calldata config,
101:     DynamicReserveConfig calldata dynamicConfig
102:   ) external restricted returns (uint256) {
103:     require(hub != address(0), InvalidAddress());
104:     require(assetId <= MAX_ALLOWED_ASSET_ID, InvalidAssetId()); // @audit-info
↪  MAX_ALLOWED_ASSET_ID = type(uint16).max = 2^16 = 65536
105:     require(!_reserveExists[hub][assetId], ReserveExists());
106:
107:     _validateReserveConfig(config);
108:     _validateDynamicReserveConfig(dynamicConfig);
109:     uint256 reserveId = _reserveCount++;
110:     uint16 dynamicConfigKey; // 0 as first key to use
111:
112:     (address underlying, uint8 decimals) =
↪  IHubBase(hub).getAssetUnderlyingAndDecimals(assetId);
```

Upon reviewing the implementation of the `Hub.addAsset()` function, it was found that there is no restriction in the code to prevent adding more than 65536 assets to the hub. Thus, theoretically, it is possible for the hub to contain more than 65536 assets.

```
File: Hub.sol
53:   function addAsset(
54:     address underlying,
55:     uint8 decimals,
56:     address feeReceiver,
57:     address irStrategy,
58:     bytes calldata irData
59:   ) external restricted returns (uint256) {
60:     require(
61:       underlying != address(0) && feeReceiver != address(0) && irStrategy !=
↪  address(0),
62:       InvalidAddress()
63:     );
64:     require(
65:       MIN_ALLOWED_UNDERLYING_DECIMALS <= decimals && decimals <=
↪  MAX_ALLOWED_UNDERLYING_DECIMALS,
66:       InvalidAssetDecimals()
67:     );
68:
69:     uint256 assetId = _assetCount++;
..SNIP..
81:     _assets[assetId] = Asset({
82:       liquidity: 0,
83:       deficit: 0,
```

Impact

Hub's assets with ID above 65536 cannot be added to the spoke.

Code Snippet

https://github.com/sherlock-audit/2025-10-aave-v4-oct-6th/blob/d792069ed8a08ef5226d77a1299c9d34aa2e6a7b/aave__aave-v4/src/spoke/Spoke.sol#L104

Tool Used

Manual Review

Recommendation

Consider removing the restriction.

Discussion

Aave Labs

The Spoke restricts asset identifiers to values below 2^16 - 1 (uint16 maximum value), so it cannot be connected to Hub assets with identifiers exceeding this limit. Given that the restriction is enforced per (Hub, assetId) pair and that a Spoke can connect to multiple Hubs concurrently, this does not impose a practical limitation, as a Hub exceeding this limit can deploy an additional Hub to list further assets.

Lucas | Sherlock

The issue has been acknowledged by the team.

Issue L-5: Cap does not prevent malicious spoke from stealing funds from users

Source: https://github.com/sherlock-audit/2025-10-aave-v4-oct-6th/issues/106

Vulnerability Detail

The various caps (e.g., add cap or debt cap) have been put in place to contain the risk of a malicious spoke.

Assume that a new spoke with extra custom logic is added to the hub, and it is granted an `addCap` of 500,000 USDC.

```
File: Hub.sol
723:   /// @dev Spoke with maximum cap have unlimited add capacity.
724:   function _validateAdd(
725:     Asset storage asset,
726:     SpokeData storage spoke,
727:     uint256 assetId,
728:     uint256 amount,
729:     address from
730:   ) internal view {
731:     require(from != address(this), InvalidAddress());
732:     require(amount > 0, InvalidAmount());
733:     require(spoke.active, SpokeNotActive());
734:     uint256 addCap = spoke.addCap;
735:     require(
736:       addCap == MAX_ALLOWED_SPOKE_CAP ||
737:         addCap * MathUtils.uncheckedExp(10, asset.decimals) >=
738:         previewAddByShares(assetId, spoke.addedShares) + amount,
739:       AddCapExceeded(addCap)
740:     );
741:   }
```

However, the existing caps in the protocol do not prevent malicious spokes from stealing assets from users who have granted allowance to the Hub.

A malicious spoke can call the `Hub.add()` function with the `from` parameter set to Bob (victim), who has granted max allowance to the Hub. Malicious spoke will set the `amount` to 500,000 USDC and `from` to Bob's address. This will pull 500,000 USDC from Bob's wallet to the hub, and mint 500,000 USD worth of USDC shares to the malicious spoke.

At this point, the `addCap` of 500,000 USDC has been reached, which prevents the spoke from supplying more USDC to the hub.

```
File: Hub.sol
187:   /// @inheritdoc IHubBase
188:   function add(uint256 assetId, uint256 amount, address from) external returns
↪  (uint256) {
189:     Asset storage asset = _assets[assetId];
190:     SpokeData storage spoke = _spokes[assetId][msg.sender];
191:
192:     asset.accrue(_spokes, assetId);
193:     _validateAdd(asset, spoke, assetId, amount, from);
194:
195:     uint128 shares = previewAddByAssets(assetId, amount).toUint128();
196:     require(shares > 0, InvalidShares());
197:     asset.addedShares += shares;
198:     spoke.addedShares += shares;
199:     asset.liquidity += amount.toUint128();
200:
201:     asset.updateDrawnRate(assetId);
202:
203:     IERC20(asset.underlying).safeTransferFrom(from, address(this), amount);
204:
205:     emit Add(assetId, msg.sender, shares, amount);
206:
207:     return shares;
208:   }
209:
```

The malicious spoke could workaround this restriction by calling the `Hub.remove()` function to withdraw all 500,000 USDC to the attacker's wallet. In this case, the malicious spoke's USDC limit will be reset to zero because `spoke.addedShares` will be zero.

It can then proceed with extracting more funds from Bob's wallet or move on to the next

victim. As a result, any users who have granted USDC allowance to the Hub will have their USDC asset drained.

```
File: Hub.sol
210:   /// @inheritdoc IHubBase
211:   function remove(uint256 assetId, uint256 amount, address to) external
 ↪  returns (uint256) {
212:     Asset storage asset = _assets[assetId];
213:     SpokeData storage spoke = _spokes[assetId][msg.sender];
214:
215:     asset.accrue(_spokes, assetId);
216:     _validateRemove(spoke, assetId, amount, to);
217:     uint256 liquidity = asset.liquidity;
218:     require(amount <= liquidity, InsufficientLiquidity(liquidity));
219:
220:     uint128 shares = previewRemoveByAssets(assetId, amount).toUint128(); //
 ↪  non zero since we round up
221:     asset.addedShares -= shares;
222:     spoke.addedShares -= shares;
223:     asset.liquidity = liquidity.uncheckedSub(amount).toUint128();
224:
225:     asset.updateDrawnRate(assetId);
226:
227:     IERC20(asset.underlying).safeTransfer(to, amount);
228:
229:     emit Remove(assetId, msg.sender, shares, amount);
230:
231:     return shares;
232:   }
```

Impact

If a malicious spoke is granted a draw limit in a hub, funds of users who grant allowance to the hub will be stolen.

Code Snippet

https://github.com/sherlock-audit/2025-10-aave-v4-oct-6th/blob/d792069ed8a08ef5226d77a1299c9d34aa2e6a7b/aave__aave-v4/src/hub/Hub.sol#L735

Tool Used

Manual Review

Recommendation

If any spoke with custom logic/code has to be added to a hub, the governance must ensure that the new spoke does not contain any malicious logic as the existing draw or debt limit does not prevent against this risk.

Alternatively, consider having a rate-limiting mechanism to restrict the amount of funds that can flow between a hub and spoke to reduce the risk.

Discussion

Aave Labs

Approvals are now granted by users to the Spoke rather than to the Hub, allowing Spokes to perform token transfers from users to the Hub. The Hub then validates that it holds sufficient liquidity before executing the corresponding action. This approach improves Spoke isolation and limits the scope of token allowances. PR: https://github.com/aave/aave-v4/pull/955

Lucas | Sherlock

Fixed is confirmed with PR#955.

Issue L-6: Invariant broken: exchange rate may decrease in certain edge case

Source: https://github.com/sherlock-audit/2025-10-aave-v4-oct-6th/issues/109

Vulnerability Detail

According to this doc https://docs.craft.do/editor/d/55fd6b0d-5d65-45f2-9cda-2522b781c9a1/949d0a70-7b27-474e-b57a-ad0ef958da2f?s=X4GYzcyYXWFNQgXxCXEhHtoLZ6zHvBeUZUnwJoF4XRCu, there is an invariant

> Add exchange rate (total assets / total shares) cannot decrease (remains constant or increases).
>
> - if no time passes, it stays constant
> - it increases due to interest accumulation, premium debt settlement and donations (from actions' rounding).
> - even if there is deficit (bad debt)
> - monotonicity should hold between actions, and previews

However, in certain edge cases this invariant may not hold.

The `totalAddedAssets() = asset.liquidity + asset.swept + asset.deficit + asset.totalOwed()`, `totalAddedShares() = asset.addedShares + asset.unrealizedFeeShares()`. If we can craft a case where at two timestamps T1 and T2, where the `totalAddedAssets()` is equal, but `totalAddedShares()` increases, then we have our edge case. This can be acheived by using rounding.

For simplicity, let's assume liquidityFee = 100%, and there is no premium shares. For `totalAddedAssets()`, we only to look at `asset.totalOwed()`, and assume other fields remain constant (which would be `asset.liquidity + asset.swept + asset.deficit`). For `totalAddedShares()`, we only look at `asset.unrealizedFeeShares()`, and assume the other field `asset.addedShares` is constant.

Now, we know `asset.totalOwed() = asset.drawnShares.rayMulUp(asset.getDrawnIndex())`, and `asset.unrealizedFeeShares() = asset.drawnShares.rayMulDown(indexDelta)`. Assume initially `asset.drawnShares = 1e18, asset.getDrawnIndex() = 1e27`. Then at the two timestamps we have:

- T1: `asset.drawnShares = 1e18, asset.getDrawnIndex() = 1e27+1`. Then `asset.totalOwed() = roundUp(1e18 * (1e27 + 1) / 1e27) = 1e18 + 1, asset.unrealizedFeeShares() = roundDown(1e18 * 1 / 1e27) = 0`
- T2: `asset.drawnShares = 1e18, asset.getDrawnIndex() = 1e27+1e9`. Then `asset.totalOwed() = roundUp(1e18 * (1e27 + 1e9) / 1e27) = 1e18 + 1, asset.unrealizedFeeShares() = roundDown(1e18 * 1e9 / 1e27) = 1`

Note we don't accrue interest between T1 to T2, or else at T2 the indexDelta will not be 1e9. We show that at T2, the nominator is the same as T1, but the denominator increases by 1, which means the exchange rate at T2 is smaller than T1, breaking the original invariant.

https://github.com/sherlock-audit/2025-10-aave-v4-oct-6th/blob/main/aave__aave-v4/src/hub/libraries/AssetLogic.sol

```
function drawn(IHub.Asset storage asset) internal view returns (uint256) {
  return asset.drawnShares.rayMulUp(asset.getDrawnIndex());
}
function totalOwed(IHub.Asset storage asset) internal view returns (uint256) {
  return asset.drawn() + asset.premium();
}
function totalAddedAssets(IHub.Asset storage asset) internal view returns (uint256)
↪  {
  return asset.liquidity + asset.swept + asset.deficit + asset.totalOwed();
}
function totalAddedShares(IHub.Asset storage asset) internal view returns (uint256)
↪  {
  return asset.addedShares + asset.unrealizedFeeShares();
}
function unrealizedFeeShares(IHub.Asset storage asset) internal view returns
↪  (uint256) {
  return asset.getFeeShares(asset.getDrawnIndex().uncheckedSub(asset.drawnIndex));
}
function getFeeShares(
  IHub.Asset storage asset,
  uint256 indexDelta
) internal view returns (uint256) {
  if (indexDelta == 0) return 0;
  uint256 liquidityFee = asset.liquidityFee;
```

```
  if (liquidityFee == 0) return 0;

  // @dev we do not simplify further to avoid overestimating the liquidity growth
  uint256 feesAmount = (asset.drawnShares.rayMulDown(indexDelta) +
    asset.premiumShares.rayMulDown(indexDelta)).percentMulDown(liquidityFee);

  return feesAmount.toSharesDown(asset.totalAddedAssets() - feesAmount,
  ↪  asset.addedShares);
}
```

Impact

This should be an edge case, and shouldn't be able to be weaponized.

Recommendation

Non-trivial to fix. If we want to fix, maybe we need to change the way interest is accrued, and fix the `totalAddedShares()`.

Discussion

Aave Labs

Minting of fee shares has been moved to a separate, restricted method and no longer occurs on every accrual. This change prevents unrealized fee shares from being included in totalAddedShares prior to execution, which previously caused inconsistencies in the supply share price between the preview and accrual logic. PR: https://github.com/aave/aave-v4/pull/933

Lucas | Sherlock

Fix confirmed with PR#933.

Issue L-7: Freezing/pausing an asset does not free/pause future spokes from using this asset.

Source: https://github.com/sherlock-audit/2025-10-aave-v4-oct-6th/issues/110

This issue has been acknowledged by the team but won't be fixed at this time.

Vulnerability Detail

When we freeze/pause an asset in HubConfigurator, it iterates all spokes, and updates the SpokeData stored in `_spokes[assetId][spoke]`. But if new spokes are added to an assetId, the new spoke can still freely be active, or have non-zero caps.

The same goes for freezing/pausing a spoke.

https://github.com/sherlock-audit/2025-10-aave-v4-oct-6th/blob/main/aave__aave-v4/src/hub/HubConfigurator.sol#L112-L131

```
function freezeAsset(address hub, uint256 assetId) external onlyOwner {
  IHub targetHub = IHub(hub);
  uint256 spokesCount = targetHub.getSpokeCount(assetId);
  for (uint256 i = 0; i < spokesCount; ++i) {
    address spokeAddress = targetHub.getSpokeAddress(assetId, i);
    _updateSpokeCaps(targetHub, assetId, spokeAddress, 0, 0);
  }
}

function pauseAsset(address hub, uint256 assetId) external onlyOwner {
  IHub targetHub = IHub(hub);
  uint256 spokesCount = targetHub.getSpokeCount(assetId);
  for (uint256 i = 0; i < spokesCount; ++i) {
    address spokeAddress = targetHub.getSpokeAddress(assetId, i);
    IHub.SpokeConfig memory config = targetHub.getSpokeConfig(assetId,
    ↪  spokeAddress);
    config.active = false;
```

```
    targetHub.updateSpokeConfig(assetId, spokeAddress, config);
  }
}
```

Impact

The freeze/pause feature does not completely freeze/pause an asset. New spokes can still be active and unpaused.

Recommendation

Document this behavior if it's by design.

Discussion

Aave Labs

Since freezing and pausing are actions applied to each Spoke rather than Hub-wide, this behavior is intended. The functions in the `HubConfigurator` contract are designed to facilitate executing the same state-changing action across all existing Spokes of the Hub in a single operation, without implying that the action should automatically apply to future Spokes.

Lucas | Sherlock

The issue has been acknowledged by the team.

Issue L-8: No option for liquidator to take supply shares

Source: https://github.com/sherlock-audit/2025-10-aave-v4-oct-6th/issues/113

Summary

In Aave V3, liquidators had the option to receive aTokens as their liquidation reward. This was useful in cases where the collateral asset was highly utilized, making it impossible to withdraw the underlying collateral token. However, this feature prevented flash-loan-based liquidations, which represent the majority of liquidations in practice. Still, the ability for someone to liquidate and hold aTokens ensured that the protocol always had a mechanism to handle liquidations even in cases of extremely high collateral utilization.

Recommendation

Introduce an option for liquidator to receive spoke supply shares instead of the underlying asset

Discussion

Aave Labs

Given that liquidations may be triggered during liquidity crunch scenarios, where no liquidity is available to be seized upon debt repayment, an alternative liquidation mechanism was introduced. Instead of transferring the underlying assets directly to the liquidator, the protocol credits the liquidator with additional added shares, which can be withdrawn later once liquidity is replenished. PR: https://github.com/aave/aave-v4/pull/884

Lucas | Sherlock

Fix is confirmed with PR#884.

Issue L-9: It might make sense to first liquidate debt than collateral

Source: https://github.com/sherlock-audit/2025-10-aave-v4-oct-6th/issues/114

Summary

When a liquidation occurs, the collateral is typically liquidated first, followed by the repayment of the debt in other words, the collateral is withdrawn and then the debt is repaid.

However, if the **debt asset and the collateral asset are the same token**, it might make more sense to **repay the debt first** in order to increase the available collateral liquidity before liquidating it.

For example: Assume the collateral liquidity is 10, and a user is about to be liquidated with 90 units of debt to be repaid for 100 units of collateral. In this case, liquidating 100 units of collateral is clearly impossible due to insufficient liquidity.

But if the debt were repaid **first** within the same liquidation transaction, the available liquidity would increase to 100, allowing the collateral to be successfully liquidated afterward.

This could be a valuable improvement, assuming that changing the order of debt repayment and collateral liquidation introduces **no additional risk**, since the involved tokens are **non-reentrant**.

Code Snippet

https://github.com/sherlock-audit/2025-10-aave-v4-oct-6th/blob/d792069ed8a08ef5226d77a1299c9d34aa2e6a7b/aave__aave-v4/src/spoke/libraries/LiquidationLogic.sol#L183-L204

Recommendation

```
bool isCollateralPositionEmpty = _liquidateCollateral(
      collateralReserve,
      collateralPosition,
      LiquidateCollateralParams({
        collateralToLiquidate: collateralToLiquidate,
        collateralToLiquidator: collateralToLiquidator,
        liquidator: params.liquidator
      })
    );

    bool isDebtPositionEmpty = _liquidateDebt(
      debtReserve,
      debtPosition,
      positionStatus,
      LiquidateDebtParams({
        reserveId: params.debtReserveId,
        debtToLiquidate: debtToLiquidate,
        premiumDebt: params.premiumDebt,
        accruedPremium: params.accruedPremium,
        liquidator: params.liquidator
      })
    );
```

reversing the orders to:

```
bool isDebtPositionEmpty = _liquidateDebt(
      debtReserve,
      debtPosition,
      positionStatus,
      LiquidateDebtParams({
        reserveId: params.debtReserveId,
        debtToLiquidate: debtToLiquidate,
        premiumDebt: params.premiumDebt,
        accruedPremium: params.accruedPremium,
        liquidator: params.liquidator
      })
```

```
    );

bool isCollateralPositionEmpty = _liquidateCollateral(
      collateralReserve,
      collateralPosition,
      LiquidateCollateralParams({
        collateralToLiquidate: collateralToLiquidate,
        collateralToLiquidator: collateralToLiquidator,
        liquidator: params.liquidator
      })
    );
```

Discussion

Aave Labs

Liquidating debt before collateral can slightly increase the supply share price (i.e., the price at which collateral is seized) due to potential rounding donations when converting asset amounts to shares during repayment. This may result in a more frictional experience for liquidators, who need to account for this behavior when designing their strategies. Given that liquidity shortfalls can prevent releasing collateral first, the system supports allowing liquidators to receive added shares in the accounting instead of underlying assets. This enables liquidations to proceed even when the system temporarily lacks sufficient liquidity. PR: https://github.com/aave/aave-v4/pull/884

Lucas | Sherlock

Fix is confirmed with PR#884.

Disclaimers

Blackthorn does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.