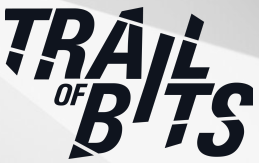

Aave V4

Security Assessment

February 10, 2026

Prepared for:

Adam Schoeman, CISO

Emilio Frangella, SVP of Engineering

Stani Kulechov, CEO

Aave Labs

Prepared by: **Simone Monica, Quan Nguyen, and Nicolas Donboly**

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Kimberly Espinoza, Project Manager
kimberly.espinoza@trailofbits.com

The following engineering director was associated with this project:

Benjamin Samuels, Engineering Director, Blockchain
benjamin.samuels@trailofbits.com

The following consultants were associated with this project:

Simone Monica, Consultant
simone.monica@trailofbits.com

Quan Nguyen, Consultant
quan.nguyen@trailofbits.com

Nicolas Donboly, Consultant
nicolas.donboly@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
October 21, 2025	Pre-project kickoff call
November 6, 2025	Delivery of report draft
November 6, 2025	Report readout meeting
December 19, 2025	Delivery of report draft
January 30, 2026	Delivery of report with fix review appendix
February 10, 2026	Delivery of updated report with fix review appendix

Executive Summary

Engagement Overview

Aave engaged Trail of Bits to review the security of Aave V4, a lending protocol comprising two main components: the Hub and the Spoke. The Hub is the single contract that contains the liquidity and manages the assets and spokes in the system. A Spoke is a contract that handles the borrowing and lending of assets for each reserve supported, where a reserve is associated with a Hub.

A team of three consultants conducted the review from October 21 to November 3, 2025, for a total of six engineer-weeks of effort. Our testing efforts focused on assessing risk isolation within the Hub-Spoke architecture, and identifying issues that could cause financial losses to the users or to the protocol. With full access to source code and documentation, we performed static and dynamic testing of the target, using automated and manual processes.

Observations and Impact

The review identified six findings: two medium-severity issues, two low-severity issues, and three informational findings. The codebase demonstrates satisfactory arithmetic practices through custom libraries that enforce explicit rounding semantics, comprehensive event logging across critical state changes, and a clear architectural separation of concerns between Hub liquidity management, Spoke risk configuration, and user position handling.

One medium-severity finding (TOB-AAVE-1) identifies a denial-of-service (DoS) condition in the deficit reporting mechanism during liquidations. The other medium-severity finding (TOB-AAVE-7) demonstrates the possibility of users being incorrectly liquidated.

Two findings warrant attention from a system configuration and validation perspective. The Spoke contract does not adhere to ERC-7201 storage layout best practices for upgradeability (TOB-AAVE-2), which creates potential risks for future upgrades when adding storage variables. Additionally, lowering a Spoke's `riskPremiumThreshold` below the premium already accrued in the system causes premium refresh operations to revert (TOB-AAVE-3), temporarily blocking user flows including repayments that would otherwise decrease risk. An asset validation gap between Hub and Spoke (TOB-AAVE-5) could prevent a Spoke from adding a valid Hub asset if the asset identifier exceeds the Spoke's maximum allowed range.

Recommendations

Based on the findings identified during the security review, Trail of Bits recommends that Aave take the following steps:

- **Remediate the findings disclosed in this report.** The medium-severity DoS issue allows attackers to block timely deficit reporting, while configuration issues can

temporarily prevent user operations during parameter updates. These findings should be addressed through direct fixes or broader refactoring efforts.

- **Implement stateful invariant fuzzing as part of continuous integration.** Integrate tools like Medusa and Echidna to continuously validate protocol invariants across the Hub-Spoke architecture, premium debt system, and share accounting mechanisms. This will provide deeper property-based verification of critical system invariants beyond the current test suite.
- **Add mutation testing to the continuous integration pipeline.** Run tools like Necessist and slither-mutate to identify gaps in test coverage and ensure that the test suite can detect real defects in code. This will improve the effectiveness of the existing test suite at catching bugs.
- **Strengthen testing of administrative parameter changes.** Develop comprehensive test scenarios that validate system behavior when governance modifies market parameters.

Finding Severities and Categories

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	*Count*
High	0
Medium	2
Low	1
Informational	4
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Auditing and Logging	1
Configuration	1
Data Validation	3
Denial of Service	2

Project Goals

The engagement was scoped to provide a security assessment of the Aave V4 protocol. Specifically, we sought to answer the following non-exhaustive list of questions:

- Does the Hub-Spoke architecture properly isolate risk and maintain accounting integrity across the different Spokes?
- Are the liquidation and deficit mechanisms economically sound and resistant to manipulation?
- Are the user risk premium calculations correct, and does that premium correctly get applied to their borrowing costs?
- Does the dynamic reserve configuration snapshot refresh and apply automatically on all health-decreasing actions?
- Does the protocol follow best practices regarding upgradeability to avoid potential issues when upgrading a contract?
- Are correct access control mechanisms in place?
- Are there edge cases or DoS vectors in the liquidation flow that could prevent bad debt resolution?
- Do the share-based accounting mechanisms handle rounding correctly across all operations to prevent value leakage or accumulation exploits?
- Can the multicall and signature-based delegation features be exploited to bypass security checks or manipulate user positions?
- Do supply caps, borrow caps, and asset offboarding procedures prevent protocol insolvency while allowing graceful market transitions?

Project Targets

The engagement involved reviewing and testing the following target.

Aave V4

Repository	https://github.com/aave/aave-v4/
Version	Initial `c89769563f9d742cc8b2e70912febb6487241b40`
	Final `8f8a61a44694199e78d3dab5fab2a9886db1d5e2`
Type	Solidity
Platform	EVM

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following:

- **`Hub`**: This contract contains the full liquidity of the system and allows each Spoke to draw and supply liquidity with the possibility of capping both draw and supply. A privileged address can add an asset to the `Hub` (and consequently be used by a Spoke), and set various other configurations. We reviewed the `Hub` contract to assess if adequate access controls are in place; if possible configuration changes can cause issues; if the asset and spoke shares are updated and stay in synchrony; if the limits imposed to a Spoke are correctly enforced; if the interests are accrued correctly; and if the risk premium is correctly computed and applied.
- **`HubConfigurator`**: This contract is set as the privileged address that can make configuration changes to the Hub. We reviewed whether every function that can change the configuration on the Hub can be called only by the owner, and we checked whether every function can correctly change the associated configuration.
- **`AssetInterestRateStrategy`**: This contract implements the calculation of the interest rates; every `Hub` contract has its own `AssetInterestRateStrategy` contract. We assessed whether the interest rate parameters can be set to only valid values, and whether the interest rates calculation is correct.
- **`Spoke`**: This contract handles the risk configuration and borrowing strategies for reserves and user positions. A reserve is a combination of `Hub` and the associated `assetId`, among other parameters. Each reserve has a dynamic configuration that specifies the `collateralFactor`, `maxLiquidationBonus`, and `liquidationFee`, and it can be updated by the governance; however, it is not immediately applied to users. We checked whether only a privileged account can add and update reserves; whether user positions are correctly updated when executing actions; whether the user's health factor is computed as documented (for example, that only assets supplied by the user as collateral are counted towards the total collateral); and whether the liquidation can be stopped by the liquidated user, or whether the liquidator can liquidate more than what is expected.
- **`SpokeConfigurator`**: This contract is the privileged address on the Spoke that handles administrative functions, such as adding a reserve or changing the reserve's state to frozen or paused. We reviewed this contract to determine if each function has the correct access control and if they allow for the proper adjustment of all possible configurations of each Spoke.
- **Position Managers**: The position managers are authorized entities allowed to manage user positions once the user allows it. The `NativeTokenGateway` and a

`SignatureGateway`. `NativeTokenGateway` contracts simplify the process of handling the native token by wrapping/unwrapping it for the user. `SignatureGateway` consumes EIP-712 typed signatures to execute Spoke actions on behalf of the users. We reviewed if they can manage a user's position only after the user has granted permission; if they can execute any malicious operations; and if EIP-712-typed signatures adhere to the standard.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- We assumed that third-party dependencies work as expected.
- The Chainlink price feeds have limited validation, and the Aave team acknowledges this; as a result, during the review, we assumed that they report a correct price.
- We did not review interactions with non-standard ERC-20 Tokens (e.g., fee-on-transfer, rebasing). The Aave team stated that the protocol does not support them.
- We trusted that governance sets and adjusts parameters correctly. We considered unsafe or disruptive parameter changes to be out of scope under this trust assumption.
- We did not validate the `AccessManager` or roles configuration across all restricted entrypoints.
- The stateful fuzzing suite is limited and not comprehensive of all the possible actions in the system.

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in-house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We used the following tools in the automated testing phase of this project:

Tool	Description	Policy
Echidna	A smart contract fuzzer that can rapidly test security properties via malicious, coverage-guided test case generation.	Local: 2 hours
Medusa	A cross-platform go-ethereum-based fuzzer providing parallelized fuzz testing of smart contracts, heavily inspired by Echidna.	Local: 2 hours
Necessist	A mutation-based tool for finding bugs in tests.	Appendix D
Slither-Mutate	A mutation testing tool for Solidity smart contracts.	Appendix D
Slither	A static analysis framework that can statically verify algebraic relationships between Solidity variables.	Appendix E

Areas of Focus

Our automated testing and verification work focused on the following system properties:

- **Share accounting integrity across the Hub-Spoke architecture**, including consistency of added shares, drawn shares, and premium shares between Hubs and Spokes, as well as underlying asset balance reconciliation
- **Exchange rate and interest accrual correctness**, verifying that the share-to-asset exchange rate is monotonically non-decreasing over time and that the drawn index properly tracks interest accumulation
- **Premium debt system mechanics**, validating that premium shares, offsets, and realized premium are correctly tracked and updated across supply, borrow, repay operations

- **Test suite quality and effectiveness**, using mutation testing to assess whether the existing test suite adequately detects bugs and can catch real defects in the contract code

Test Results

The results of this focused testing are detailed below.

Stateful invariant testing

Trail of Bits developed a stateful fuzzing suite that implements 51 invariants, covering global system properties and operation-specific constraints across the Hub-Spoke architecture. The fuzzing suite is limited to the supply, withdraw, borrow and repay actions; see appendix F for possible improvements.

Summary of Invariants

The table below summarizes the number and type of invariants we ran for each component. We ran the fuzzer both locally and on the cloud.

Component	Invariant Type	Total Number
Global	System invariants	10
`Spoke.Supply`	Functional invariants	9
`Spoke.Withdraw`	Functional invariants	11
`Spoke.Borrow`	Functional invariants	10
`Spoke.Repay`	Functional invariants	11
Total Invariants		**51**

Summary of Failed Invariants

Throughout the engagement, the following five invariants failed:

- AAVE-INV-19: User `healthFactor` does not go below `HEALTH_FACTOR_LIQUIDATION_THRESHOLD` when withdrawing. (`Spoke.Withdraw`)
- AAVE-INV-30: User `healthFactor` does not go below `HEALTH_FACTOR_LIQUIDATION_THRESHOLD` when borrowing. (`Spoke.Borrow`)

- AAVE-INV-38: User `healthFactor` does not decrease when repaying. (`Spoke.Repay`)
- AAVE-INV-39: User `healthFactor` does not go below `HEALTH_FACTOR_LIQUIDATION_THRESHOLD` when repaying. (`Spoke.Repay`)

Global System Invariants

These invariants verify system-wide properties across the Hub-Spoke architecture. The tests check that the sum of added shares across all Spokes equals Hub added shares; that Hub underlying balances match tracked liquidity minus swept amounts; that exchange rates and drawn indices are non-decreasing over time; that premium shares satisfy the premium offset requirement; that deficit tracking is consistent across Spokes and Hub; and that users cannot have debt without collateral or borrow when below the liquidation threshold.

Property	Result	Fix Status
AAVE-GINV-1: For each asset, sum of `spoke.addedShares` across spokes equals hub `asset.addedShares`	**Passed**	**Passed**
AAVE-GINV-2: Hub underlying balance ≥ `asset.liquidity - asset.swept`	**Passed**	**Passed**
AAVE-GINV-3: `asset.irStrategy != address(0)`	**Passed**	**Passed**
AAVE-GINV-4: `previewRestoreByShares(asset.premiumShares) ≥ asset.premiumOffset`	**Passed**	**Passed**
AAVE-GINV-5: For each asset, sum of `spoke.deficit` across spokes equals `asset.deficit`	**Passed**	**Passed**
AAVE-GINV-6: Exchange rate `(addedAssets+1e6)/(addedShares+1e6)` is non-decreasing over time	**Passed**	**Passed**
AAVE-GINV-7: `asset.drawnIndex` is non-decreasing over time	**Passed**	**Passed**
AAVE-GINV-8: If a spoke has `addedAssets > 0`, then `spoke.drawnShares + spoke.addedShares != 0`	**Passed**	**Passed**

AAVE-GINV-9: If `user.totalCollateralValue == 0`, then `user.totalDebtValue == 0`	Passed	Passed
AAVE-GINV-10: If previous `user.healthFactor < HEALTH_FACTOR_LIQUIDATION_THRESHOLD`, `user.drawnShares` must not increase on any reserve	Passed	Passed

Spoke.Supply Functional Invariants

These invariants validate the supply operation's postconditions. The tests verify that returned shares and amounts are greater than zero; that user supplied shares increase correctly; that underlying tokens transfer from users to the Hub; that Hub liquidity increases by the supplied amount; and that both Hub and Spoke added shares update properly while accounting for unrealized fee shares.

Property	Result	Fix Status
AAVE-INV-1: Returned `shares > 0`	Passed	Passed
AAVE-INV-2: Returned `amount > 0`	Passed	Passed
AAVE-INV-3: Operation must succeed when preconditions hold	Passed	Passed
AAVE-INV-4: `user.suppliedShares(new) == user.suppliedShares(old) + shares`	Passed	Passed
AAVE-INV-5: Hub underlying balance increases by `amount`	Passed	Passed
AAVE-INV-6: User underlying balance decreases by `amount`	Passed	Passed
AAVE-INV-7: `asset.liquidity(new) == asset.liquidity(old) + amount`	Passed	Passed
AAVE-INV-8: `asset.addedShares(new) - unrealizedFeeShares == asset.addedShares(old) + shares`	Passed	Passed
AAVE-INV-9: `spoke.addedShares(new) == spoke.addedShares(old) + shares`	Passed	Passed

Spoke.Withdraw Functional Invariants

These invariants test the `withdraw` operation to ensure proper state transitions. The tests verify that returned shares and amounts are greater than zero; that user-supplied shares decrease correctly; that underlying tokens transfer from the Hub to users; that Hub liquidity decreases by the withdrawn amount; that Hub and Spoke added shares update appropriately accounting for unrealized fee shares; and that the user's health factor does not increase (preventing invalid collateral reduction).

Property	Result	Fix Status
AAVE-INV-10: Returned `shares > 0`	**Passed**	**Passed**
AAVE-INV-11: Returned `amount > 0`	**Passed**	**Passed**
AAVE-INV-12: `user.suppliedShares(new) == user.suppliedShares(old) - shares`	**Passed**	**Passed**
AAVE-INV-13: Hub underlying balance decreases by `amount`	**Passed**	**Passed**
AAVE-INV-14: User underlying balance increases by `amount`	**Passed**	**Passed**
AAVE-INV-15: `asset.liquidity(new) == asset.liquidity(old) - amount`	**Passed**	**Passed**
AAVE-INV-16: `asset.addedShares(new) - unrealizedFeeShares == asset.addedShares(old) - shares`	**Passed**	**Passed**
AAVE-INV-17: `spoke.addedShares(new) == spoke.addedShares(old) - shares`	**Passed**	**Passed**
AAVE-INV-18: User `healthFactor` does not increase on withdraw (`HF_old ≥ HF_new`)	**Passed**	**Passed**
AAVE-INV-19: User `healthFactor` does not go below `HEALTH_FACTOR_LIQUIDATION_THRESHOLD` when withdrawing	**Failed**	**Passed**
AAVE-INV-20: Operation must succeed when preconditions hold	**Passed**	**Passed**

Spoke.Borrow Functional Invariants

These invariants validate the borrow functionality and verify correct state updates. The tests check that returned shares and amounts are greater than zero; that user-drawn shares increase correctly; that underlying tokens transfer from the Hub to users; that Hub liquidity decreases by the borrowed amount; that both Hub and Spoke drawn shares update properly; and that the user's health factor does not increase during borrowing.

Property	Result	Fix Status
AAVE-INV-21: Returned `shares > 0`	**Passed**	**Passed**
AAVE-INV-22: Returned `amount > 0`	**Passed**	**Passed**
AAVE-INV-23: `user.drawnShares(new) == user.drawnShares(old) + shares`	**Passed**	**Passed**
AAVE-INV-24: Hub underlying balance decreases by `amount`	**Passed**	**Passed**
AAVE-INV-25: User underlying balance increases by `amount`	**Passed**	**Passed**
AAVE-INV-26: `asset.liquidity(new) == asset.liquidity(old) - amount`	**Passed**	**Passed**
AAVE-INV-27: `asset.drawnShares(new) == asset.drawnShares(old) + shares`	**Passed**	**Passed**
AAVE-INV-28: `spoke.drawnShares(new) == spoke.drawnShares(old) + shares`	**Passed**	**Passed**
AAVE-INV-29: User `healthFactor` does not increase on borrow (`HF_old ≥ HF_new`)	**Passed**	**Passed**
AAVE-INV-30: User `healthFactor` does not go below `HEALTH_FACTOR_LIQUIDATION_THRESHOLD` when borrowing	**Failed**	**Passed**

Spoke.Repay Functional Invariants

These invariants test the repay operation to ensure debt is properly reduced. The tests verify that the user drawn shares decrease correctly; that underlying tokens transfer from users to the Hub; that Hub liquidity increases by the repaid amount; and that both Hub and Spoke drawn shares decrease appropriately to reflect the debt reduction.

Property	Result	Fix Status
AAVE-INV-31: Returned `restoredAmount > 0`	**Passed**	**Passed**
AAVE-INV-32: `user.drawnShares(new) == user.drawnShares(old) - restoredShares`	**Passed**	**Passed**
AAVE-INV-33: Hub underlying balance increases by `restoredAmount`	**Passed**	**Passed**
AAVE-INV-34: User underlying balance decreases by `restoredAmount`	**Passed**	**Passed**
AAVE-INV-35: `asset.liquidity(new) == asset.liquidity(old) + restoredAmount`	**Passed**	**Passed**
AAVE-INV-36: `asset.drawnShares(new) == asset.drawnShares(old) - restoredShares`	**Passed**	**Passed**
AAVE-INV-37: `spoke.drawnShares(new) == spoke.drawnShares(old) - restoredShares`	**Passed**	**Passed**
AAVE-INV-38: User `healthFactor` does not decrease when repaying	**Failed**	**Passed**
AAVE-INV-39: User `healthFactor` does not go below `HEALTH_FACTOR_LIQUIDATION_THRESHOLD` when repaying	**Failed**	**Passed**
AAVE-INV-40: Operation must succeed when preconditions hold	**Passed**	**Passed**

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	The codebase demonstrates strong arithmetic practices through custom libraries (`WadRayMath`, `PercentageMath`, `SharesMath`) that enforce explicit rounding semantics and overflow protection. All mathematical operations use `WAD` (18 decimals) and `RAY` (27 decimals) precision with clearly defined rounding directions (up/down), preventing precision loss vulnerabilities. Unchecked arithmetic is minimal and documented. However, the codebase lacks a comprehensive specification document that provides a one-to-one mapping between formulas and code implementations with complete edge case analysis.	**Satisfactory**
Auditing	The system implements comprehensive event logging covering critical state changes across Hub operations (add, draw, remove, restore, deficit reporting), Spoke actions (supply, borrow, withdraw, repay, liquidations), and configuration updates. However, we identified one informational documentation issue (TOB-AAVE-4) where event parameter descriptions do not accurately reflect emitted values, which could lead to integration errors in off-chain monitoring systems.	**Satisfactory**
Authentication / Access Controls	Access control is centralized through OpenZeppelin's `AccessManaged` pattern with role-based permissions managed by an `AccessManager` contract. The system properly gates administrative functions (asset configuration, Spoke/Hub management, parameter updates) and implements a position manager delegation system for user operations. While the access control implementation is sound, the upgradeability pattern (TOB-AAVE-2) does not follow best practices for storage layout in upgradeable contracts, creating potential risks	**Satisfactory**

	for future upgrades.	
Complexity Management	The architecture demonstrates clear separation of concerns, with Hub handling liquidity and interest accrual, Spokes managing risk configuration and user positions, and Gateways abstracting user interactions. Complex logic is properly isolated into libraries (`LiquidationLogic` for liquidation calculations, `AssetLogic` for Hub operations, `SharesMath` for share-to-asset conversions). However, the premium debt mechanism introduces significant complexity across multiple components, requiring careful coordination between Hub premium tracking and Spoke user premium accounting. Additionally, validation consistency between components requires attention; this is shown in TOB-AAVE-5, where Hub accepts asset IDs beyond the range Spoke can process, and TOB-AAVE-6, where `SignatureGateway` function parameters contain assumptions that should be explicitly validated.	**Satisfactory**
Decentralization	The system uses the `AccessManaged` pattern with role-based controls through `HubConfigurator` and `SpokeConfigurator` contracts. The `Spoke` contract is upgradeable but does not follow ERC-7201 storage layout best practices (TOB-AAVE-2), while the Hub uses a standard non-upgradeable deployment. Governance controls critical system parameters and can temporarily block user operations, including withdrawals and repayments, through configuration changes (TOB-AAVE-3). Governance is assumed to act correctly, with unsafe parameter changes considered out of scope.	**Moderate**
Documentation	The codebase features comprehensive NatSpec documentation across interfaces and implementations. Functions include clear descriptions of purpose, parameters, and return values. Some complex mechanisms, like the premium debt system and deficit reporting, could benefit from additional high-level architectural documentation.	**Satisfactory**
Low-Level Manipulation	Inline assembly is used extensively but appropriately in mathematical libraries to optimize gas consumption for critical operations, including `WAD`/`RAY` arithmetic, percentage calculations, and overflow checks. Assembly	**Satisfactory**

	usage is justified for performance-critical code paths that execute frequently in DeFi protocols. The assembly is confined to well-tested library functions rather than scattered throughout business logic.	
Testing and Verification	The protocol includes a substantial test suite built on Foundry with unit tests, integration tests, scenario tests, gas benchmarks, and stateless fuzzing tests covering critical mechanisms like liquidations, premium debt accumulation, share rounding, and multi-Hub interactions. Test organization follows a clear structure with dedicated files for each contract and mechanism. The testing of administrative parameter changes and their edge case interactions could be strengthened; this is evidenced by TOB-AAVE-3, where lowering the `riskPremiumThreshold` below the accumulated premium unexpectedly blocks user operations until governance intervention. However, the current testing approach lacks a stateful invariant testing suite using fuzzing tools like Echidna or Medusa, which would provide deeper property-based verification of critical protocol invariants across the Hub-Spoke architecture and premium debt system. We also recommend running mutation testing tools like Necessist and slither-mutate (appendix D).	**Moderate**
Transaction Ordering	The system uses virtual shares (1e6 offset) to reduce first-deposit and share manipulation risks. Signature-based operations include deadline checks that prevent replay attacks. However, the liquidation flows remain exposed to ordering manipulation, as demonstrated by TOB-AAVE-1.	**Satisfactory**

Summary of Findings

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

ID	Title	Type	Severity
1	Deficit reporting denial of service via micro-collateral	Denial of Service	**Medium**
2	Spoke contract does not follow upgradeability best practices	Configuration	**Informational**
3	Lowering riskPremiumThreshold can temporarily block premium refresh and user actions	Denial of Service	**Informational**
4	Incorrect liquidation event documentation	Auditing and Logging	**Informational**
5	A Spoke may not be able to add a valid Hub asset	Data Validation	**Low**
6	The setSelfAsUserPositionManagerWithSig function assumes the positionManager value in the params argument is address(this)	Data Validation	**Informational**
7	Users can become immediately liquidatable after executing an action	Data Validation	**Medium**

Detailed Findings

1. Deficit reporting denial of service via micro-collateral	
Severity: **Medium**	Difficulty: **Medium**
Type: Denial of Service	Finding ID: TOB-AAVE-1
Target: `src/spoke/libraries/LiquidationLogic.sol`	

Description

During liquidation, a deficit is reported only if the liquidated collateral is emptied and the user has no other active collateral while some debt remains. This condition is sound in principle, but "active collateral" is defined as any non-zero collateral, so a user can keep deficit reporting blocked by maintaining dust collateral on another reserve.

Liquidation first computes user account data and then evaluates whether a deficit should be reported. The decision gates on whether the targeted collateral is emptied and whether the user still has any other active collateral legs.

```
function _evaluateDeficit(
  bool isCollateralPositionEmpty,
  bool isDebtPositionEmpty,
  uint256 activeCollateralCount,
  uint256 borrowedCount
) internal pure returns (bool) {
  if (!isCollateralPositionEmpty || activeCollateralCount > 1) {
    return false;
  }
  return !isDebtPositionEmpty || borrowedCount > 1;
}
```

Figure 1.1: Deficit decision gated by active collateral count in the `_evaluateDeficit` function

In account data computation, any non-zero `suppliedShares` on a reserve marked as collateral increments `activeCollateralCount`, regardless of how economically insignificant the value is. This allows a 1-wei collateral position to keep the user "not in deficit" even after the main collateral leg is fully liquidated.

The impact is a DoS against timely deficit crystallization. Liquidators must clear every collateral leg alongside the main liquidation to allow for deficit reporting, which increases gas and operational complexity. Until that happens, uncollectible debt continues to accrue interest.

Exploit Scenario

Alice owes 100,000 USDC with a single LINK collateral initially worth $130,000. A 40% LINK price drop reduces collateral to $78,000, so liquidating LINK would exhaust it and leave residual debt that should be recognized as a deficit. Just before liquidation, Alice supplies 1 wei as collateral to other reserves (e.g., WETH, AAVE) and enables them. After LINK is fully liquidated, these dust legs still count as “active collateral,” blocking deficit reporting. The residual bad debt then remains as owed and continues to accrue interest until liquidators also clear the dust positions, increasing the operational burden.

Recommendations

Short term, treat collateral below a protocol dust threshold as inactive when determining whether “other active collateral” exists.

Long term, revisit the deficit reporting design to remove this grief surface.

2. Spoke contract does not follow upgradeability best practices

Severity: **Informational**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-AAVE-2
Target: `src/spoke/Spoke.sol`	

Description

The `Spoke` contract is upgradeable, but it does not follow best practices for handling storage in upgradeable contracts.

The current architecture has a `SpokeInstance` contract that inherits `Spoke` and will be the actual deployed implementation. `Spoke` inherits multiple contracts:

- `Multicall`: This contract does not have any storage variables
- `NoncesKeyed`: This contract does have storage variables. It is a custom version of the OpenZeppelin `NoncesKeyed` non-upgradeable contract.
- `AccessManagedUpgradeable`: This is a contract from OpenZeppelin Upgradeable. It handles storage variables following ERC-7201.
- `EIP712`: This is a standard `EIP712` contract from Solady. It does not have storage variables (only immutables).

```
abstract contract Spoke is ISpoke, Multicall, NoncesKeyed, AccessManagedUpgradeable,
EIP712 {
```

Figure 2.1: The `Spoke` contract definition (`Spoke.sol#L26`)

For the `Multicall`, `EIP712`, and `NoncesKeyed` contracts it will not be possible to add a new storage variable if needed, as it would change the storage layout of the deriving contracts. While it is unlikely that `Multicall` and `EIP712` would need new storage variables, it could be possible for `NoncesKeyed` if a different or new nonces strategy is added.

Additionally, the `Spoke` contract itself has many storage variables and does not follow any upgradeable pattern, meaning that storage variables could be easily added without changing the storage layout of the `SpokeInstance` contract. The `SpokeInstance` contract does not currently have any storage variables, so the `Spoke` would be able to add storage variables without causing problems.

Exploit Scenario

The developers want to add a feature to the `NoncesKeyed` contract that requires adding a storage variable. However, this is not possible, since doing so would change the Spoke storage layout.

Recommendations

Short term, follow the ERC-7201 for contracts that can be used as base for an upgradeable contract. This will make it possible in the future to add new storage variables without causing any issues.

3. Lowering riskPremiumThreshold can temporarily block premium refresh and user actions

Severity: **Informational**	Difficulty: **High**
Type: Denial of Service	Finding ID: TOB-AAVE-3
Target: `src/hub/Hub.sol`	

Description

Lowering a Spoke's `riskPremiumThreshold` below the premium already accrued in the system causes `hub.refreshPremium` to revert, which in turn temporarily blocks user flows that touch premium (repay, borrow, withdraw, and explicit updates). This is reversible by governance, which restores the parameter to a permissive value; however, until then, the market remains blocked.

The check enforces that, unless the threshold is disabled, a Spoke's `premiumShares` must not exceed `drawnShares` scaled by the configured `riskPremiumThreshold`. If governance lowers this threshold below the effective premium ratio that users have already accumulated, the very next premium refresh, often embedded inside user actions, will revert, blocking risk reduction and normal operations until the parameter is relaxed or state is migrated.

```
uint24 riskPremiumThreshold = spoke.riskPremiumThreshold;
require(
  riskPremiumThreshold == MAX_RISK_PREMIUM_THRESHOLD ||
    spoke.premiumShares <= spoke.drawnShares.percentMulUp(riskPremiumThreshold),
  InvalidPremiumChange()
);
```

Figure 3.1: Threshold check reverts if `premiumShares` exceed the new cap (`Hub.sol#L730-L735`)

The impact is a protocol-level DoS on premium refresh that cascades into user actions, including repayments that would otherwise decrease risk, because these paths invoke `refreshPremium` under the hood and will hit the stricter, now-violated invariant.

Exploit Scenario

Consider ten users on a Spoke, each with 1,000 `drawnShares` and 500 `premiumShares`, so the Spoke totals are 10,000 `drawnShares` and 5,000 `premiumShares` (an effective 50%). Governance lowers `riskPremiumThreshold` to 40%. When any user attempts to repay, the operation triggers a premium refresh with the new totals (e.g., `drawnShares` falls to 9,000 and `premiumShares` to 4,500, still 50%), thereby violating the 40% cap. The call

reverts, so even a risk-reducing repayment cannot proceed; the market remains blocked until the threshold is raised.

Recommendations

Short term, avoid lowering `riskPremiumThreshold` below the current effective premium ratio on the Spoke.

Long term, implement a comprehensive test suite that simulates different scenarios when the admin modifies market parameters.

4. Incorrect liquidation event documentation

Severity: **Informational**	Difficulty: **Low**
Type: Auditing and Logging	Finding ID: TOB-AAVE-4
Target: `src/spoke/interfaces/ISpokeBase.sol`	

Description

The `LiquidationCall` event documentation incorrectly describes the `liquidatedCollateral` parameter. The documentation states that this parameter represents "the amount of collateral received by the liquidator," but the implementation intentionally emits the total collateral liquidated from the user's position, which includes both the amount transferred to the liquidator and the liquidation fee retained by the protocol. Off-chain systems relying on this documentation will misinterpret the emitted value as the liquidator's net proceeds rather than the gross collateral seized.

```
/// @dev Emitted when a borrower is liquidated.
/// @param collateralAssetId The identifier of the asset used as collateral, to
receive as result of the liquidation.
/// @param debtAssetId The identifier of the asset to be repaid with the
liquidation.
/// @param user The address of the borrower getting liquidated.
/// @param liquidatedDebt The debt amount of borrowed asset to be liquidated.
/// @param liquidatedCollateral The amount of collateral received by the liquidator.
/// @param liquidator The address of the liquidator.
/// @param receiveShares Whether the liquidator receives collateral in supplied
shares or in underlying assets.
event LiquidationCall(
  uint256 indexed collateralAssetId,
  uint256 indexed debtAssetId,
  address indexed user,
  uint256 liquidatedDebt,
  uint256 liquidatedCollateral,
  address liquidator,
  bool receiveShares
);
```

Figure 4.1: Incorrect parameter documentation (ISpokeBase.sol#L61-L77)

Recommendations

Short term, update the event documentation to accurately describe `liquidatedCollateral` as the total collateral seized from the user's position. This prevents integration errors caused by the documentation mismatch.

Long term, consider adding a separate parameter to emit the collateral amount transferred to the liquidator. This provides complete liquidation data in a single event without requiring off-chain calculations of fee deductions.

5. A Spoke may not be able to add a valid Hub asset

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-AAVE-5
Target: `src/spoke/Spoke.sol`	

Description

The `Spoke` contract has an `addReserve` function that allows adding a new reserve that is a combination of the Hub address and its corresponding `assetId`, among other parameters (figure 5.1). The `assetId` has a restriction: it must be less than `MAX_ALLOWED_ASSET_ID` (`type(uint16).max`). However, this restriction is not present on the Hub's `addAsset` function (figure 5.2), meaning that a Spoke may not be able to use a valid Hub's `assetId`.

```
function addReserve(
   address hub,
   uint256 assetId,
   address priceSource,
   ReserveConfig calldata config,
   DynamicReserveConfig calldata dynamicConfig
 ) external restricted returns (uint256) {
   require(hub != address(0), InvalidAddress());
   require(assetId <= MAX_ALLOWED_ASSET_ID, InvalidAssetId());
   ...
```

Figure 5.1: A snippet of the `addReserve` function (Spoke.sol#L98-L106)

```
function addAsset(
   address underlying,
   uint8 decimals,
   address feeReceiver,
   address irStrategy,
   bytes calldata irData
 ) external restricted returns (uint256) {
   ...
   uint256 assetId = _assetCount++;
   ...
```

Figure 5.2: A snippet of the `addAsset` function (Hub.sol#L55-L71)

Exploit Scenario

The Hub governance adds assets over an extended operational period, eventually reaching 65,536 assets. Subsequent `addAsset` calls assign IDs exceeding `type(uint16).max`. A Spoke administrator attempts to add one of these newer assets as a reserve through

`addReserve`, but the call reverts at the `assetId` validation check. This prevents the Spoke from using valid Hub assets.

Recommendations

Short term, enable the Spoke to support the same range of Hub's asset ID. If that is not currently possible due to design decisions, document the relationship between Hub and Spoke asset ID constraints and establish an upgrade plan for Spoke contracts to support asset IDs exceeding `type(uint16).max` if the protocol approaches this limit. This ensures operational continuity without restricting the immutable Hub's capabilities.

Long term, add tests that verify Hub and Spoke validation rules at boundary conditions, such as asset counts near `MAX_ALLOWED_ASSET_ID`. This catches inconsistent limits between components before deployment.

6. The setSelfAsUserPositionManagerWithSig function assumes the positionManager value in the params argument is address(this)

Severity: **Informational**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-AAVE-6
Target: `src/position-manager/SignatureGateway.sol`	

Description

The `setSelfAsUserPositionManagerWithSig` function (figure 6.1) allows setting the gateway as the user position manager on the specified Spoke. The `params` argument is a `SetUserPositionManager` struct (figure 6.2) that has a `positionManager` field. However, this address is never used when calling the `setUserPositionManagerWithSig` function; instead, `address(this)` is used. While it is correct to use `address(this)` and not allow the user to specify an arbitrary position manager, this could create confusion for the user or eventually for the monitoring system, since the `positionManager` specified in the params argument is not used.

```
function setSelfAsUserPositionManagerWithSig(
   address spoke,
   EIP712Types.SetUserPositionManager calldata params,
   bytes calldata signature
 ) external onlyRegisteredSpoke(spoke) {
   try
     ISpoke(spoke).setUserPositionManagerWithSig(
       address(this),
       params.user,
       params.approve,
       params.nonce,
       params.deadline,
       signature
     )
   {} catch {}
 }
```

Figure 6.1: The `setSelfAsUserPositionManager` function (`SignatureGateway.sol#L171-L186`)

```
struct SetUserPositionManager {
   address positionManager;
   address user;
   bool approve;
   uint256 nonce;
```

```
    uint256 deadline;
}
```

Figure 6.2: The `SetUserPositionManager` struct (EIP712Types.sol#L9-L15)

Exploit Scenario

The user calls the `setSelfAsUserPositionManagerWithSig` function with a `positonManager` set to a different address than the `SignatureGateways` is calling. The signature is valid for the `params` struct data sent; however, the transaction reverts unexpectedly with an `InvalidSignature` error.

Recommendations

Short term, in the `setSelfAsUserPositionManagerWithSig` function, add a check that `params.positionManager` is equal to `address(this)`.

Long term, add tests that validate EIP-712 signature parameters match the actual values used in function calls.

7. Users can become immediately liquidatable after executing an action

Severity: **Medium**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-AAVE-7
Target: `src/spoke/Spoke.sol, src/spoke/libraries/LiquidationLogic.sol`	

Description

The `Spoke` contract validates a user's health factor before applying risk premium updates, which can leave users in a liquidatable state immediately after performing legitimate actions such as withdrawing collateral or disabling collateral.

In the `withdraw`, `borrow`, and `setUsingAsCollateral` functions, the contract first calls `_refreshAndValidateUserPosition` to check that the user's health factor remains above the liquidation threshold, and then calls `_notifyRiskPremiumUpdate` to update the user's premium debt parameters:

```
function withdraw(
  uint256 reserveId,
  uint256 amount,
  address onBehalfOf
) external onlyPositionManager(onBehalfOf) returns (uint256) {
  // ... withdrawal logic ...

  userPosition.suppliedShares -= withdrawnShares.toUint128();

  if (_positionStatus[onBehalfOf].isUsingAsCollateral(reserveId)) {
    uint256 newRiskPremium = _refreshAndValidateUserPosition(onBehalfOf);
    _notifyRiskPremiumUpdate(onBehalfOf, newRiskPremium);
  }
  // ...
}
```

Figure 7.1: The `withdraw` function validates health factor before updating risk premium (Spoke.sol#L213-L240)

The `_refreshAndValidateUserPosition` function calculates the user's total debt value using the current `premiumShares` and `premiumOffset` stored in the user's position:

```
function _refreshAndValidateUserPosition(address user) internal returns (uint256) {
  UserAccountData memory accountData = _calculateAndRefreshUserAccountData(user);
  require(
    accountData.healthFactor >= HEALTH_FACTOR_LIQUIDATION_THRESHOLD,
    HealthFactorBelowThreshold()
```

```
  );
  return accountData.riskPremium;
}
```

Figure 7.2: Health factor validation occurs before risk premium is applied (Spoke.sol#L680-L690)

The `_notifyRiskPremiumUpdate` function then refreshes the user's premium debt accounting:

```
function _notifyRiskPremiumUpdate(address user, uint256 newRiskPremium) internal {
  PositionStatus storage positionStatus = _positionStatus[user];
  // ...
  positionStatus.hasPositiveRiskPremium = newRiskPremium > 0;

  uint256 reserveId = _reserveCount;
  while ((reserveId = positionStatus.nextBorrowing(reserveId)) !=
PositionStatusMap.NOT_FOUND) {
    UserPosition storage userPosition = _userPositions[user][reserveId];
    // ...
    uint256 newPremiumShares =
userPosition.drawnShares.percentMulUp(newRiskPremium);
    uint256 newPremiumOffset = hub.previewDrawByShares(assetId, newPremiumShares);

    userPosition.premiumShares = newPremiumShares.toUint128();
    userPosition.premiumOffset = newPremiumOffset.toUint128();
    userPosition.realizedPremium += accruedPremium.toUint128();

    IHubBase.PremiumDelta memory premiumDelta = IHubBase.PremiumDelta({
      sharesDelta: newPremiumShares.signedSub(oldPremiumShares),
      offsetDelta: newPremiumOffset.signedSub(oldPremiumOffset),
      realizedDelta: accruedPremium.toInt256()
    });

    hub.refreshPremium(assetId, premiumDelta);
    // ...
  }
}
```

Figure 7.3: Risk premium update occurs after health factor validation (Spoke.sol#L812-L850)

In theory, the premium refresh operation should not increase the user's debt, as it simply transitions accounting from in-flight to realized premium debt. However, during the computation of premium debt accruals at the user level, the premium debt can increase by up to 2 wei. This occurs because opposite rounding directions are applied: the conversion from premium drawn shares to assets rounds up, while the offset calculation rounds down.

Since the health factor validation occurs before `_notifyRiskPremiumUpdate`, a user whose health factor is exactly at the liquidation threshold will pass validation but then have their premium debt increase by up to 2 wei, pushing their health factor below the threshold and making them immediately liquidatable.

Additionally a related issue was found in an ongoing PR trying to improve this behavior during the audit.

Note: This was found by running our invariant testing suite against the PR #978 and then discussing an invariant failure with the Aave team. The Aave team later decided to close the PR in favor of a different approach.

The idea behind PR #978 was to overestimate the user's debt by adding 2 wei for every borrowed reserve to account for the possible impact of 2 wei imprecision in premium debt calculations on the user's health factor.

That situation could cause the user to be immediately liquidatable when the `_refreshAndValidateUserAccountData` function is called before the `_notifyRiskPremiumUpdate` function, because the health factor check is in `_refreshAndValidateUserAccountData` while `_notifyRiskPremiumUpdate` could increase the user's debt, hence lowering the health factor potentially under the liquidation threshold but the transaction would still succeed. In particular, this pattern is present in the `borrow` (figure 7.4), `withdraw`, `setUsingAsCollateral`, and `updateUserDynamicConfig` functions.

```
function borrow(
   uint256 reserveId,
   uint256 amount,
   address onBehalfOf
 ) external onlyPositionManager(onBehalfOf) returns (uint256, uint256) {
   ...

   uint256 newRiskPremium =
_refreshAndValidateUserAccountData(onBehalfOf).riskPremium;
   _notifyRiskPremiumUpdate(onBehalfOf, newRiskPremium);
```

Figure 7.4: The `borrow` function validates the health factor before updating premium accounting (Spoke.sol#L266-L267)

The PR code at the time of our review was using the over estimated debt value to compute the health factor that was used to check if a position was liquidatable (figure 7.5). That would have caused users with a health factor near the liquidation threshold to be incorrectly liquidated due to the overestimation of their debt value when their real premium debt could have been lower.

```
function _validateLiquidationCall(
   mapping(address user => ISpoke.PositionStatus) storage positionStatus,
   ValidateLiquidationCallParams memory params
 ) internal view {
   ...
   require(
     params.healthFactor < HEALTH_FACTOR_LIQUIDATION_THRESHOLD,
     ISpoke.HealthFactorNotBelowThreshold()
```

```
    );
```

Figure 7.5: The liquidation validation uses the health factor computed with overestimated debt (`LiquidationLogic.sol#L362-L365`)

Exploit Scenario

Alice has a position with her health factor at slightly more than 1.0 (the liquidation threshold). She attempts to withdraw a small amount of collateral. The health factor check passes because her position is calculated to be exactly at the threshold. After the validation, `_notifyRiskPremiumUpdate` is called, and due to rounding in the premium debt computation, her premium debt increases by 2 wei. This small increase pushes her health factor below 1.0, and a liquidator immediately liquidates her position despite having passed all validation checks during her withdrawal transaction.

Recommendations

Short term, improve the design around rounding operations of the system to avoid instant debt creation, and add adequate tests to avoid regressions on the introduction of new bugs in other parts of the system.

Long term, add invariant tests that verify users with health factors above the liquidation threshold cannot be liquidated, and that users cannot become liquidatable immediately after a successful borrow or withdraw operation. This prevents the issue because the test suite will detect rounding mismatches between validation and liquidation checks.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage
Transaction Ordering	The system's resistance to transaction-ordering attacks

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category does not apply to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Code Quality Recommendations

This appendix contains recommendations for findings that do not have immediate or obvious security implications. However, addressing them may enhance the code's readability and may prevent the introduction of vulnerabilities in the future.

- **Prefer `getBucketWord` for consistency in `isUsingAsCollateralOrBorrowing`.** The function manually indexes `self.map` and shifts bits, rather than using the `getBucketWord` helper, as is done in the `isBorrowing` and `isUsingAsCollateral` functions. While the current usage is correct, using the helper instead would improve readability and maintain consistent bucket access if the layout changes.

```
function isUsingAsCollateralOrBorrowing(
  ISpoke.PositionStatus storage self,
  uint256 reserveId
) internal view returns (bool) {
  unchecked {
    return (self.map[reserveId.bucketId()] >> ((reserveId % 128) << 1)) & 3 !=
0;
  }
}
```

Figure C.1: The `isUsingAsCollateralOrBorrowing` function in `PositionStatusMap.sol`

- **Unused `OperationNotSupported` custom error can be removed.**

```
/// @notice Thrown when the operation is not supported.
error OperationNotSupported();
```

Figure C.2: Unused error (`src/misc/UnitPriceFeed.sol#L20-L21`)

- **`asset.drawnIndex` can be used instead of `asset.getDrawnIndex()`.** The `_applyPremiumDelta` function is always called after the `asset.accrue()` function; as a result, the `asset.drawnIndex` is already updated and can be directly used.

```
function _applyPremiumDelta(
   Asset storage asset,
   SpokeData storage spoke,
   PremiumDelta calldata premium,
   uint256 premiumAmount
 ) internal {
   uint256 drawnIndex = asset.getDrawnIndex();
```

Figure C.3: Snippet of the `_applyPremiumDelta` function (`Hub.sol#L702-L708`)

D. Mutation Testing

During our review, we ran two mutation testing tools, Necessist and slither-mutate, to identify potential shortcomings in the test suite and implementation.

Necessist mutates tests; it runs tests with statements and method calls removed to help identify broken tests.

To run Necessist, you must first rename/copy the `tests` folder to `test` and change the test field in `foundry.toml` to `test = 'tests'`. After that, `necessist .` will try to mutate and run every test present. It also has other configurations, which are explained in the repository.

Figure D.1 represents a snippet of the Necessist output, in particular for the `Hub.Add.t.sol` file. The first line of this snippet indicates that the test still passes if we remove `params.assetAddedAmount = hub1.previewRemoveByShares(assetId, params.assetAddedShares);`. Depending on the context, this could mean that the test is missing some assertions on the value for that variable.

```
test/unit/Hub/Hub.Add.t.sol:684:5-684:92: `params.assetAddedAmount =
hub1.previewRemoveByShares(assetId, params.assetAddedShares);` passed
test/unit/Hub/Hub.Add.t.sol:685:5-685:55: `params.availableLiq = amount -
params.drawnAmount;` passed
test/unit/Hub/Hub.Add.t.sol:687:5-687:94: `params.spoke2AddedAmount =
hub1.previewRemoveByShares(assetId, params.spoke2AddedShares);` passed
test/unit/Hub/Hub.Add.t.sol:707:7-707:63: `(drawn, ) = hub1.getSpokeOwed(assetId,
address(spoke1));` passed
test/unit/Hub/Hub.Add.t.sol:711:7-711:44: `params.assetAddedShares += addShares;`
passed
test/unit/Hub/Hub.Add.t.sol:712:7-712:94: `params.assetAddedAmount =
hub1.previewRemoveByShares(assetId, params.assetAddedShares);` passed
test/unit/Hub/Hub.Add.t.sol:714:7-714:96: `params.spoke1AddedAmount =
hub1.previewRemoveByShares(assetId, params.spoke1AddedShares);` passed
```

Figure D.1: Small snippet of the Necessist output

Some lines removed, such as lines 711–712, may be a benign result from Necessist, since the following assertion where they are used is checking if a certain value is greater than them. While the variables at lines 687–714 are never used in an assertion, this could highlight a missing assertion.

```
params.assetAddedAmount = hub1.previewRemoveByShares(assetId,
params.assetAddedShares);
params.availableLiq = amount - params.drawnAmount;
params.spoke2AddedShares = hub1.getSpokeAddedShares(assetId, address(spoke2));
params.spoke2AddedAmount = hub1.previewRemoveByShares(assetId,
params.spoke2AddedShares);
```

```
params.aliceBalance = MAX_SUPPLY_AMOUNT + params.drawnAmount;
params.bobBalance = MAX_SUPPLY_AMOUNT - amount;

uint256 addShares = 1; // minimum for 1 share
uint256 addAmount;
for (uint256 i = 0; i < numAdds; i++) {
  addAmount = minimumAssetsPerAddedShare(hub1, assetId);

  // bob add minimal amount
  Utils.add({
    hub: hub1,
    assetId: assetId,
    caller: address(spoke1),
    amount: addAmount,
    user: bob
  });

  (uint256 drawn, ) = hub1.getAssetOwed(assetId);
  assertGt(drawn, 0);
  (drawn, ) = hub1.getSpokeOwed(assetId, address(spoke1));
  assertGt(drawn, 0);

  params.availableLiq += addAmount;
  params.assetAddedShares += addShares;
  params.assetAddedAmount = hub1.previewRemoveByShares(assetId,
params.assetAddedShares);
  params.spoke1AddedShares += addShares;
  params.spoke1AddedAmount = hub1.previewRemoveByShares(assetId,
params.spoke1AddedShares);
params.bobBalance -= addAmount;
   // hub
    assertGe(hub1.getAddedAssets(assetId), params.assetAddedAmount, 'hub addedAmount
after');
    assertGe(hub1.getAddedShares(assetId), params.assetAddedShares, 'hub addedShares
after');
    assertEq(hub1.getAssetLiquidity(assetId), params.availableLiq, 'asset liquidity
after');
    assertEq(
      hub1.getAsset(assetId).lastUpdateTimestamp,
      vm.getBlockTimestamp(),
      'asset lastUpdateTimestamp after'
    );
    // spoke1
    assertEq(
      hub1.getSpokeAddedAssets(assetId, address(spoke1)),
      hub1.previewRemoveByShares(assetId, params.spoke1AddedShares),
      'spoke1 addedAmount after'
    );
    assertEq(
      hub1.getSpokeAddedShares(assetId, address(spoke1)),
      params.spoke1AddedShares,
      'spoke1 addedShares after'
    );
```

```
    // spoke2
    assertEq(
      hub1.getSpokeAddedAssets(assetId, address(spoke2)),
      hub1.previewRemoveByShares(assetId, params.spoke2AddedShares),
      'spoke2 addedAmount after'
    );
    assertEq(
      hub1.getSpokeAddedShares(assetId, address(spoke2)),
      params.spoke2AddedShares,
      'spoke2 addedShares after'
    );
    // token balance
    assertEq(
      tokenList.dai.balanceOf(address(hub1)),
      params.availableLiq,
      'hub token balance after'
    );
```

Figure D.2: Snippet of the test case referred in figure D.1

slither-mutate executes mutations on the source code and then runs the tests to check that they are caught. Figure D.3 is a snippet of the output for the mutation of the Hub contract while running the Hub unit tests; these tests should presumably catch a mutation:

`slither-mutate src/hub/Hub.sol --contract-names Hub --test-cmd='forge test --match-path="tests/unit/Hub/**"'`

```
...
INFO:Slither-Mutate:[CR] Line 232: 'asset.accrue()' ==> '//asset.accrue()' -->
UNCAUGHT
INFO:Slither-Mutate:[CR] Line 243: 'asset.updateDrawnRate(assetId)' ==>
'//asset.updateDrawnRate(assetId)' --> UNCAUGHT
...
```

Figure D.3: Small snippet of the slither-mutate output

Figure D.4 represents the `remove` function with the uncaught mutations applied. Running `forge test --match-path="tests/unit/Hub/**"` passes all the tests, highlighting that the tests may be improved.

```
function remove(uint256 assetId, uint256 amount, address to) external returns
(uint256) {
   Asset storage asset = _assets[assetId];
   SpokeData storage spoke = _spokes[assetId][msg.sender];


   //asset.accrue();
   _validateRemove(spoke, amount, to);


   uint256 liquidity = asset.liquidity;
```

```
    require(amount <= liquidity, InsufficientLiquidity(liquidity));

    uint120 shares = asset.toAddedSharesUp(amount).toUint120();
    asset.addedShares -= shares;
    spoke.addedShares -= shares;
    asset.liquidity = liquidity.uncheckedSub(amount).toUint120();

    //asset.updateDrawnRate(assetId);

    asset.underlying.safeTransfer(to, amount);

    emit Remove(assetId, msg.sender, shares, amount);

    return shares;
  }
```

Figure D.4: The `remove` function with the commented lines of Figure D.3

E. Slither Script to Detect Access Control Misconfigurations

This appendix provides a Slither script to check the following properties between `Hub`/`HubConfigurator` and `Spoke`/`SpokeConfigurator`:

- Every external function declared in `HubConfigurator` and `SpokeConfigurator` must have the `onlyOwner` modifier.
- Every state-changing function (not `view` or `pure`) in the `Hub` or `Spoke` contract called by the corresponding configurator must have the `restricted` modifier.
- A function with the `restricted` modifier declared in the `Hub` or `Spoke` contract must be called at least once by the corresponding configurator.

Functions that are an exception to the last property can be added to the corresponding exception list; for example, the `Hub`'s `mintFeeShares` function has the `restricted` modifier but is not meant to be called by the `HubConfigurator`.

```
from slither import Slither
from slither.core.declarations import Contract
from slither.core.declarations.function import FunctionType

slither = Slither(".")

# Check every external function in the HubConfigurator contract has the onlyOwner
modifier
# Check every Hub restricted function can be reached by the HubConfigurator
contract.
# If a function is restricted but meant to not be called by the HubConfigurator
contract, it should be added to the hub_restricted_functions_exception list.
# Check if a state changing function in Hub is called by the HubConfigurator but
does not have the restricted modifier.
# Same thing for Spoke/SpokeConfigurator contracts

# Hub functions that are restricted but expected to not be called by the
HubConfigurator contract
hub_restricted_functions_exception = ["mintFeeShares"]
# Spoke functions that are restricted but expected to not be called by the
SpokeConfigurator contract
spoke_restricted_functions_exception = []


def find_contract_in_compilation_units(contract_name: str) -> Contract:
    contracts = slither.get_contract_from_name(contract_name)
    return (contracts[0]) if len(contracts) > 0 else print("Contract not found")
```

```
def check_configurator(configurator: Contract) -> set:
    functions_called = set()
    interface_name = "IHub" if configurator.name == "HubConfigurator" else "ISpoke"
    for function in configurator.functions_declared:
        # Check every external function declared in the configurator contract has
the onlyOwner modifier
        if (
            function.visibility == "external"
            and function.function_type == FunctionType.NORMAL
        ):
            if "onlyOwner" not in [modifier.name for modifier in
function.modifiers]:
                print(
                    f"Function {function.name} in the {configurator.name} contract
does not have the onlyOwner modifier"
                )
        # Get all the functions name called by the configurator contract with the
specific interface
        for (c_called, f_called) in function.high_level_calls:
            if c_called == interface_name:
                functions_called.add(f_called.function_name)
    return functions_called


def check_restricted_functions(
    contract: Contract, functions_called: set, restricted_functions_exception: list
) -> None:
    for function in contract.functions_declared:
        if (
            function.name not in restricted_functions_exception
            and "restricted" in [modifier.name for modifier in function.modifiers]
            and function.name not in functions_called
        ):
            print(
                f"Function {function.name} in the {contract.name} contract is
restricted but not called by the configurator contract"
            )
        if (
            function.name in functions_called
            and "restricted" not in [modifier.name for modifier in
function.modifiers]
            and not (function.view or function.pure)
        ):
            print(
                f"Function {function.name} in the {contract.name} contract is not
restricted but called by the configurator contract"
            )


hub_configurator = find_contract_in_compilation_units("HubConfigurator")
spoke_configurator = find_contract_in_compilation_units("SpokeConfigurator")
hub = find_contract_in_compilation_units("Hub")
spoke = find_contract_in_compilation_units("Spoke")
```

```
ihub_functions_called = check_configurator(hub_configurator)
ispoke_functions_called = check_configurator(spoke_configurator)

check_restricted_functions(
    hub, ihub_functions_called, hub_restricted_functions_exception
)
check_restricted_functions(
    spoke, ispoke_functions_called, spoke_restricted_functions_exception
)
```

Figure E.1: `check_access_controls.py`

We recommend adding the script to the project's CI to prevent future bugs.

F. Recommendations to Expand the Fuzz Testing Suite

This appendix provides recommendations for adding enhancements and features to the fuzz testing suite to increase its coverage of the Hub and Spoke components' system states. Prioritize mathematical invariants over other types of invariants that could be tested with a unit or integration tests (e.g., checking that a borrow action can succeed only when the reserve is in a borrowable state). Use both Echidna and Medusa, since they work on the same fuzzing test suite interchangeably, and each tool may have its own generation and mutation strategy. Other fuzzing tools are recommended as well for similar reasons. Note that Echidna needs a special configuration to correctly handle libraries with external functions (see here), as it is the case for the `LiquidationLogic` library.

Liquidation Invariants

The following are some possible invariants related to the liquidation flow.

- Liquidated user collateral shares removed > 0.
- Liquidated user collateral shares removed equals expected `collateralToLiquidate` in shares.
- When `receiveShares` is true:
    - Liquidator shares gained equals the expected `collateralSharesToLiquidator`.
    - The collateral hub underlying balance is unchanged.
    - Liquidator underlying balance is unchanged.
- When `receiveShares` is false:
    - Liquidator underlying balance increases by the expected `collateralToLiquidator`.
    - Liquidator collateral shares are unchanged.
    - Collateral hub underlying outflow equals underlying delivered to liquidator.
- Shares received by the `feeReceiver` + shares to liquidator are equal to shares removed to the liquidated user.
- Debt hub underlying inflow equals expected `debtToLiquidate`.

- Liquidator debt-token outflow equals expected `debtToLiquidate`.
- Debt asset liquidity increases by the expected `debtToLiquidate`.
- Premium debt reduction equals `min(userPremiumDebt, debtToLiquidate)`.
- If drawn debt is liquidated, `drawnSharesReduced == previewRestoreByAssets(drawnDebtLiquidated)`.
- If only premium debt is liquidated, `drawnSharesReduced == 0`.
- If the liquidated user has `userDrawnShares` equal to 0 for the debt reserve, the user's isBorrowing flag for that debt reserve is false.
- Liquidated user `healthFactor` increases after liquidation.
- If the liquidated user's collateral taken by the liquidator is greater than 0, its value ≥ `DUST_LIQUIDATION_THRESHOLD`.
- If the liquidated user's debt reserve that was liquidated is greater than 0, its value ≥ `DUST_LIQUIDATION_THRESHOLD`.

General Functionality

Support two Hubs and have the Spokes use reserves from the different Hubs.

Expose administrative functions to change the system configuration, such as updating a `Reserve` configuration, to the fuzzer.

G. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

From January 21 to January 23, 2026, Trail of Bits reviewed the fixes and mitigations implemented by the Aave team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, of the seven issues described in this report, Aave has resolved three issues and has accepted the remaining four issues. The Aave team has added context for each issue it chose not to address. Additionally, we note that the Aave team has improved the overall testing of the codebase with the addition of invariant testing.

For additional information, please see the Detailed Fix Review Results below.

ID	Title	Severity	Status
1	Deficit reporting denial of service via micro-collateral	Medium	Risk Accepted
2	Spoke contract does not follow upgradeability best practices	Informational	Resolved
3	Lowering riskPremiumThreshold can temporarily block premium refresh and user actions	Informational	Risk Accepted
4	Incorrect liquidation event documentation	Informational	Resolved
5	A Spoke may not be able to add a valid Hub asset	Low	Risk Accepted
6	The setSelfAsUserPositionManagerWithSig function assumes the positionManager value in the params argument is address(this)	Informational	Risk Accepted
7	Users can become immediately liquidatable after executing an action	Medium	Resolved

Detailed Fix Review Results

TOB-AAVE-1: Deficit reporting denial of service via micro-collateral
Risk Accepted. The client provided the following context for this finding's fix status:

> *Acknowledged. 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.*

TOB-AAVE-2: Spoke contract does not follow upgradeability best practices
Resolved in PR #1138. The `NoncesKeyed` contract now follows ERC-7201 namespaced storage, allowing future storage additions without disrupting the Spoke storage layout.

TOB-AAVE-3: Lowering riskPremiumThreshold can temporarily block premium refresh and user actions
Risk Accepted. The client provided the following context for this finding's fix status:

> *Acknowledged. The `riskPremiumThreshold` is a security lever in the Hub that can be used to protect against faulty or malicious spokes submitting invalid premium data, by blocking any actions that would update the value beyond the established threshold. It also enables configuration of spoke activity, as it can be set to zero for spokes that do not rely on any premium mechanism.*
>
> *Updates to this parameter undergo risk and technical assessment by the DAO and risk providers.*

TOB-AAVE-4: Incorrect liquidation event documentation
Resolved in PR #992. The event has been revised along with its documentation to correctly distinguish between total collateral liquidated and the amount received by the liquidator.

TOB-AAVE-5: A Spoke may not be able to add a valid Hub asset
Risk Accepted. The client provided the following context for this finding's fix status:

Acknowledged. 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.

TOB-AAVE-6: The setSelfAsUserPositionManagerWithSig function assumes the positionManager value in the params argument is address(this)
Risk Accepted. The client provided the following context for this finding's fix status:

Acknowledged. All parameters passed in the struct are included in the signature and are later validated by the Spoke, meaning the parameters of the signed struct and parameters actually used (including address(this)) must exactly match. For this reason, adding an additional validation is not considered necessary.

TOB-AAVE-7: Users can become immediately liquidatable after executing an action
Resolved in PR #991. Premium accounting now uses full precision, token decimals combined with Ray (27 decimals) to avoid truncation errors inherent in Ray math. With this change, a user's premium debt remains unchanged immediately after any executable action.

H. Fix Review Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	**Description**
Undetermined	The status of the issue was not determined during this engagement.
Risk Accepted	The issue was acknowledged by the client.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review assessments, supporting client organizations in the technology, defense, blockchain, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, Uniswap, Solana, Ethereum Foundation, Linux Foundation, and Zoom.

To keep up with our latest news and announcements, please follow @trailofbits on X or LinkedIn and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

Trail of Bits performed all activities associated with this project in accordance with a statement of work and an agreed-upon project plan.

Security assessment projects are time-boxed and often rely on information provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test software controls and security properties. These techniques augment our manual security review work, but each has its limitations. For example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. A project's time and resource constraints also limit their use.