PUBLIC

Code Assessment

of the Aave V4 Smart Contracts

February 19th, 2026

Produced for

Aave Labs

by

CHAINSECURITY

Contents

1 Executive Summary

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

Thank you for trusting us to help Aave Labs with this security audit. Our executive summary provides an overview of the subjects covered in our audit of the latest reviewed contracts of Aave V4 according to Scope to support you in forming an opinion on their security risks.

Aave V4 introduces a lending protocol with a modular architecture centered around the Hub and Spoke model, designed to unify liquidity management across multiple markets while preserving isolation and configurability.

The most critical subjects covered in our audit are arithmetic precision and asset solvency. Regarding arithmetic precision the invariant violation reported in Share price can decrease because of fee rounding has been satisfactorily addressed. Regarding asset solvency, issue 2 Wei rounding can trigger liquidations and create additional debt has been addressed by increasing the accounting precision of premium debt.

The general subjects covered are liquidations, functional correctness and event handling. All reported issues have been addressed.

This version of the code (v0.5.7) is still under review. Additional code and specification changes can be found in a successive version (v0.5.9), which ChainSecurity is currently reviewing.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but do not replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered, and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	3
• Code Corrected	3
Low-Severity Findings	9
• Code Corrected	6
• Specification Changed	3

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The following contracts are in scope:

```
src/hub/AssetInterestRateStrategy.sol
src/hub/Hub.sol
src/hub/HubConfigurator.sol
src/hub/libraries/AssetLogic.sol
src/hub/libraries/Premium.sol
src/hub/libraries/SharesMath.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/GatewayBase.sol
src/position-manager/NativeTokenGateway.sol
src/position-manager/SignatureGateway.sol
src/position-manager/libraries/EIP712Hash.sol
src/spoke/AaveOracle.sol
src/spoke/Spoke.sol
src/spoke/SpokeConfigurator.sol
src/spoke/TreasurySpoke.sol
src/spoke/instances/SpokeInstance.sol
src/spoke/libraries/KeyValueList.sol
src/spoke/libraries/LiquidationLogic.sol
src/spoke/libraries/PositionStatusMap.sol
src/spoke/libraries/ReserveFlagsMap.sol
src/spoke/libraries/UserPositionDebt.sol
src/utils/Multicall.sol
src/utils/NoncesKeyed.sol
src/utils/Rescuable.sol
```

The assessment was performed on the source code files inside the Aave V4 repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	6 Oct 2025	dc31f9a4d54c0503093ef6939e6e8a8d2586709d	Initial Version
2	9 Oct 2025	14a824b258eb76b7a1ec86bd6cbe79da2539aed8	updated code
3	19 Nov 2025	91bb988323a635cdf67c1fb36d039306a8a6528b	third version
4	30 Nov 2025	6959e3219b5506bf2acae18551cbb2a68a5b8fba	v0.5.6
5	28 Jan 2026	31afa65a91f99ca7ec0437a1b438f65d3261d164	v0.5.7

For the solidity smart contracts, the compiler version `0.8.28` was chosen.

2.1.1 Excluded from scope

Any contracts not explicitly listed in the scope section above are excluded from the assessment. Third party dependencies are assumed to behave according to their specification.

Deployment scripts and tests are also excluded from the scope.

2.2 System Overview

This system overview describes Version 2 of the contracts as defined in the Assessment Overview.

At the end of this report section, we have added subsections for each of the changes according to the versions.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Aave Labs offers Aave V4, which introduces a Hub - Spoke architecture that separates global liquidity accounting from user interactions. The Hub acts as a central vault that holds all assets and tracks the system's total liquidity and debt, while Spokes serve as gateways through which users supply, withdraw, borrow, repay, and interact with the protocol. It is expected that one or more Hubs get deployed per chain and that a variety of Spokes connect to these Hubs. This design aims at increasing modularity and flexibility.

2.2.1 Hub

The Hub is an immutable contract that holds all assets. The users of the Hub are the Spokes, and for every supported asset, the Hub keeps track of the amount added (supplied) by every Spoke, the available liquidity, the amount drawn (borrowed) by every Spoke, the amount *swept* by governance, and possibly the deficit (realized bad debt). In short, the Hub keeps a global view of the liquidity and the debt positions of Spokes for all assets. It also manages the interest rate strategies for each asset and handles interest accrual over time for each asset.

The Hub implements share-based accounting to keep track of the "added" and "drawn" amount for each Spoke.

2.2.1.1 Adding Liquidity

Spokes supply (*add* in Hub terminology) assets to the Hub and get *added shares*. The value of added shares is determined by the amount of assets in the Hub and the total existing supply of shares. The total amount of one asset in the Hub is the sum of the available liquidity, the amount *swept*, the deficit and the total owed amount of that asset. The share price for added shares is defined as the ratio between total assets and total added shares, and is strictly increasing over time. It can increase due to interest accrual, premium debt interest, and rounding in favor of existing shareholders when assets are added or removed. The total assets include an amount of 10^6 virtual assets and 10^6 *dead* shares, to prevent first depositor attacks and prevent share price inflation.

The `add()` function can be called by a Spoke to supply an asset. The Spoke must be enabled for the asset, and the Spoke's total added amount cannot exceed the Spoke's `addCap` for the asset. `add()` takes the asset amount as an argument, and mints the corresponding amount of shares, possibly rounded down. The asset is transferred to the Hub from an address specified by the calling Spoke. Liquidity can be withdrawn with the `remove()` function, which lets the Spoke specify an asset amount and a recipient. Shares belonging to the Spoke are burnt by converting the asset amount, possibly rounding up. The `remove()` call can fail because the requested liquidity is unavailable, because it has been drawn or swept.

2.2.1.2 Drawing Liquidity

Spokes can borrow assets from the Hub (*draw* in Hub terminology). The amount Spokes owe is recorded as *drawn amount* (stored in debt shares) and *premium debt*. The *drawn amount* of debt is equal to the amount of drawn shares times the *drawn index*. The drawn index of the asset is a 27 decimals precision number which is linearly increased by the interest rate accrual since the last update starting from 1. When accrued often, the linear accruals are functionally equivalent to exponentially compounding accruals. The interest rate is queried from an *interest rate strategy* and depends on the utilization factor of the asset in the Hub. Since the interest rate is positive, the drawn index is strictly increasing over time.

Another component of the total owed amount is *premium debt*. This is a novel Aave V4 feature that allows charging higher interest rates when borrowing against riskier collaterals. It is described in detail in the Premium Debt section

Spokes can call the `draw()` function, specifying the requested amount and the recipient. The function validates that the requested amount plus current drawn amount do not exceed the `drawCap` for the Spoke. Drawn shares are minted by dividing the requested amount by the current drawn index, and rounding up. Calls to `draw()` can fail if not enough liquidity is available. Debt can be repaid with the `restore()` function which allows specifying both a drawn debt amount and premium debt amount to repay, and pulls the required funds from the address specified.

It is important to note that the Hub does not perform any solvency checks on Spokes or users. Each Spoke is responsible for ensuring that its users maintain sufficient collateralization. The Hub is therefore trusting the Spokes to manage user solvency properly and maintain the overall health of the system. Only trusted Spokes should be given access to the Hub's liquidity.

2.2.1.3 Assets

Supported assets are non-rebasing ERC20 tokens, without transfer fees and hooks. An asset can be registered in the Hub through the permissioned `addAsset()` function. Registering an asset assigns it an `assetId`, and the same underlying token can be listed multiple times under different `assetIds`. The accounting for every listing of the same underlying will be kept separately. For every asset an interest rate strategy and interest rate data are configured. Each asset also has its own liquidity fee and fee receiver, described more in depth in the Fees section.

The permissioned `addSpoke()` allows enabling a Spoke to add and draw liquidity for an asset in the Hub. Each Spoke-Asset pair is configured with a `addCap` and a `drawCap` to limit the total amount of assets that can be added or drawn by the Spoke. This allows the Hub to manage risk across multiple Spokes.

2.2.1.4 Swept Liquidity

Every asset potentially has a `reinvestmentController` address. This is a special role that can allocate the liquidity of the asset to external products without minting debt shares and therefore accruing interest. The `reinvestmentController` can use `sweep()` to take liquidity from the Hub, and can give it back with `reclaim()`. Yield potentially generated by swept liquidity is opaque to the Hub, how the yield is attributed is at the discretion of the `reinvestmentController`. Swept liquidity does not contribute to the utilization factor used to compute the interest rate, for that purpose it is treated as available liquidity.

2.2.1.5 Deficit

Deficit, that is bad debt incurred by a Spoke, is tracked at the Hub level. The deficit amount is included in the total asset amount, and therefore still contribute to the value of supply shares, which is therefore non-decreasing. When a Spoke incurs a deficit, it reports this deficit to the Hub using `reportDeficit()`. The Hub then increases the total amount of assets in deficit for that asset. The deficit can be reduced with `eliminateDeficit()`. This function must be called from a Spoke and burns the amount of added shares by that Spoke to cover the deficit. It is assumed that an insurance mechanism is put in place to cover deficits when they occur.

2.2.2 Spokes

Spokes are contracts that serve as gateways for user interactions with the protocol. Each Spoke is deployed as an upgradeable proxy that points to the `SpokeInstance` implementation contract.

Users are expected to interact with the protocol through Spokes. Spokes are responsible for ensuring user-level solvency. As Spokes keep track of individual user positions, they enforce collateralization requirements and manage liquidations. They derive a health factor for each user based on their collateral balances and debt balances. Through the `LiquidationLogic` library invoked by `liquidationCall()`, Spokes allow liquidators to liquidate unhealthy users.

2.2.2.1 Reserves

Each Spoke maintains a list of reserves, where each reserve corresponds to an `(assetId,hub)` pair. Remapping `assetId` s to `reserveId` s allow Spokes to support assets from different Hubs. Every asset in a Hub can only be listed once as a reserve in the Spoke, but the same underlying token could be listed multiple times in a Spoke as different assets in the same Hub, or in different Hubs.

The reserve configuration includes whether it is in the `paused` or `frozen` state, where:

- `frozen`: supplying and borrowing are disabled; the reserve can still be disabled as collateral. Liquidations are not affected by freezing the collateral or debt reserve.
- `paused`: supplying, withdrawing, borrowing, and repaying are disabled, the reserve cannot be enabled/disabled as collateral. Liquidation with that reserve as collateral or debt are also disabled.

The reserve configuration also includes the `collateralRisk`, better described in section Premium Debt, and *dynamic reserve configurations*: the `collateralFactor`, the `maxLiquidationBonus`, and the `liquidationFee`.

Dynamic reserve configuration is a novel Aave V4 mechanism to update the parameters of a reserve without affecting existing loans that use that reserve. It allows the protocol to update collateral factors and liquidation parameters without negatively impacting existing positions right away. Thus, when reducing the collateral factor of a reserve, existing borrowers cannot be liquidated and have time to adjust their positions accordingly. Actions that reduce the users health factor (withdraw, borrow) will always use the latest configuration when verifying the user's solvency but actions that increase the health factor (repay, supply) will use the configuration key of the user position to allow the user to improve their health factor even if the current configuration would make their position liquidatable.

In the Spoke's reserve data, a mapping is maintained between an integer `configKey` and a dynamic reserve configuration. When the configuration of a reserve is updated, the `configKey` is incremented and the new configuration values are stored under the new key. For every user, a `configKey` is stored in their position, and it is used when calculating a user's health factor or determining liquidation parameters. The configuration key for all collaterals of a user are updated whenever they take actions that can worsen their health, such as `withdraw()`, `borrow()` or an asset is removed from the collateral list with `setUsingAsCollateral()`. When a reserve is added as collateral, the latest configuration for that reserve is used.

The user or its position manager can decide to `updateUserDynamicConfig()` to the latest configuration key for all collaterals. Authorized callers can also update any user's dynamic configuration if needed. Note that updating the dynamic configuration can only be done for all the user's collaterals and not individually.

The Spoke authority can also update any existing configuration with `updateDynamicReserveConfig()`. This action potentially worsens the health of existing positions, making them liquidatable.

2.2.2.2 Supplying liquidity

Users (or their authorized Position Managers) can supply liquidity to a reserve (therefore to an asset in a Hub) by using the `supply()` function. The supplied amount is deposited in the Hub of the specified `reserveId`, through `Hub.add()`, which pulls the underlying amount from the user's balance. The Hub operation increases the Spoke's share in the Hub, and the new amount of shares minted is credited to user. Therefore, for every reserve, the Spoke holds shares of the corresponding Hub asset, and in the Spoke the shares are credited to individual users.

Users (or their authorized Position Managers) can withdraw supplied reserves through the Spoke's `withdraw()` function. The requested amount (or at most the user's balance) of a reserve is removed from the Hub, and the amount of shares burned for this operation is subtracted from the user's balance. Supplied reserves can be used as collateral for borrowing, therefore a collateralization check is performed when withdrawing a reserve that is used as collateral. If the withdrawn reserve is used as collateral, the user's risk premium is recomputed after the withdrawal. The withdrawal can also fail in case not enough liquidity is available in the Hub.

2.2.2.3 Borrowing

Spokes allow users (and Position Managers) to borrow liquidity available in the Hub, through the `borrow()` function. The Spoke draws the requested reserve amount from the Hub, through `Hub.draw()`, which increases the Spoke *drawn shares* balance for that asset. These drawn shares are in turn credited to the borrowing user. To ensure solvency of borrowers, the user's health is validated after borrowing. The risk premium of the borrower is also updated. Borrowing can revert if not enough liquidity is available in the Hub.

Debt can be repaid through function `repay()`. Each user has two separate debt balances, the *drawn* debt and the *premium* debt. The *drawn* debt is due to the accrual of the borrowed amount according to the interest rate calculated in the Hub. Additionally, the accrued amount is multiplied with the *risk premium* and goes to constitute the premium debt. When repaying, first the premium debt of a user is reduced, and then the drawn debt.

2.2.2.4 Position Managers

Users can enable `positionManager`'s to manage their positions on their behalf. The `NativeTokenGateway` and the `SignatureGateway` are two instances of position managers. The `NativeTokenGateway` allows users to supply and withdraw the native chain token (e.g., ETH) by wrapping and unwrapping it to its corresponding wrapped token (e.g., WETH). The `SignatureGateway` allows users to perform gasless operations by signing messages off-chain that are then relayed on-chain by a third party. Position managers must be approved by the user to manage their positions through `setUserPositionManager()`. Spokes decide which positions managers are available to users through `updatePositionManager()`.

2.2.3 Collateralization

Aave V4's borrowing is overcollateralized to ensure the solvency of the system. The borrowing power of a user is the sum of the value of their supplied collaterals multiplied by the collateral factors of the collaterals. The health factor of a user is then calculated as follows:

$$\mathcal{HF} = \frac{\sum_i (CF_i \cdot V_i)}{D_v}$$

where CF_i is the collateral factor of asset i, V_i is the value of the supplied amount of asset i, and D_v is the total value of the user's debt.

A Spoke can configure the collateral factor for each of its reserves. If a user's health score falls below 1, they become liquidatable.

A user can decide which of their supplied assets to use as collateral with `setUsingAsCollateral()`. Only assets used as collateral contribute to the user's borrowing power and can be seized during liquidation.

2.2.4 Position Status Map

A user's positions within a Spoke are stored in a `PositionStatusMap`. For each `reservedId`, two bits are used to indicate whether the reserve is being used as collateral and/or borrowed by the user. One word can store information for up to 128 reserves. The first 7 bits of the `reserveId` are used to index the bit within the word, while the remaining bits specify the `bucketId` that holds the word for the corresponding reserve.

Whenever a user adds a reserve as collateral or borrows from a reserve, the corresponding bits in the `PositionStatusMap` are set with `setBorrowing()` and `setUsingAsCollateral()`. When a user removes a reserve as collateral or repays all their debt from a reserve, the corresponding bits are cleared. The Spoke uses this map to efficiently iterate over a user's collaterals and debts when calculating their health factor and risk premium or when updating the premium debt for every borrow position.

2.2.5 Premium Debt

Aave V4 introduces premium debt, which represents an additional portion of a borrower's debt that accrues interest based on the risk of their collateral. Borrowers supplying riskier collateral will accrue higher premium debt, which increases their total debt. Aave V4 updates a borrower's premium debt whenever the premium risk could increase, such as when a reserve is no longer used as collateral or when a collateral is withdrawn. As this operation can be costly, the premiums are not updated when the risk decreases (e.g., when a borrower supplies more collateral or repays debt). Borrowers should themselves call `updateUserRiskPremium()` to update their premium debt when the risk decreases.

Note that the premium debt is updated during liquidations as the risk premium change could increase if the liquidator liquidates a collateral that is configured as safe, while leaving riskier collaterals.

The risk premium of the borrower is a number, and is determined by sorting the collaterals from least risky to most risky based on their `collateralRisk` parameter. Then, starting from the least risky collateral, the system sums up the value of the collaterals multiplied by their respective `collateralRisk` until the total value of the collaterals reaches the total debt of the borrower. This weighted sum is then divided by the total debt to yield the risk premium. In this sense, the risk premium can be thought of as an average of the collateral risk factors, weighted by collateral amount, truncating the collaterals where the debt is fully covered:

$$riskPremium = \frac{\sum_i (V_i \cdot r_i)}{D_v}$$

subject to

$$\sum_i V_i \leq D_v$$

and

$$r_i \leq r_{i+1}$$

The last collateral included in the sum may be only partially included to ensure that the total collateral value matches the debt value D_v exactly.

The risk premium of a user is used to calculate the extra amount of interest that the user is charged. The risk premium is a factor that is multiplied to the base interest, and the result is the premium interest charged to the user.

This is implemented by minting *premium shares* to the user, proportionally to their drawn shares and their risk premium. The premium shares accrue interest at the same interest as drawn shares, but only the interest of premium shares has to be repaid and not the principal, so when they are issued, their current value is recorded (this is the `premiumOffset`), so that it can be later deducted from the future value of the shares. The difference is the *accrued premium*.

$$premiumShares_j = drawnShares_j \cdot RiskPremium$$

Every time the drawn shares of the user change (on `borrow()`, `repay()`, or liquidations), the risk premium of the user changes (disable collateral, `withdraw()`), or the premium debt is repaid (`repay()` or liquidations), the accounting of the premium debt is updated. The risk premium in principle is continuously changing, because the debt and collateral continuously changes because of interest, and their values change because of price changes, however these changes are ignored as they are considered minor. In case a user's risk premium in use becomes considerably different to the risk premium that would be computed, it can be refreshed by a restricted role (or by the user themselves) through `updateUserRiskPremium()`.

The amount of premium shares changes when the drawn shares change, or the risk premium of the user changes. The premium debt accrued (`accruedPremium`) by the old premium shares is therefore "stashed" in the `realizedPremium` accumulator, and new `premiumShares` and `premiumOffset` amounts are computed. Likewise, if only premium debt is repaid (and drawn shares and risk premium do not change), the accounting must also be updated: the new realized premium is computed, the repaid amount is deducted from it, and the premium offset and shares are updated.

$$accruedPremium = premiumShares \cdot drawnIndex - premiumOffset$$

$$realizedPremium_{n+1} = realizedPremium_n + accruedPremium$$

Premium debt is always repaid before drawn debt (which accrues interest) in repayment operations (liquidations and repayments).

2.2.5.1 Premium delta

The accounting of premium debt is performed at the user level in the Spoke, and at the Spoke level in the Hub. When a user is updated in a Spoke, the change in their balances is forwarded to the Hub as signed integer deltas, which are applied to the overall Spoke balances maintained in the Hub.

In conclusion, a borrower's total debt D is the sum of their drawn debt, the realized premium, and the accrued premium debt of each of their borrowing positions.

2.2.6 Interest Model

The interest rate strategy for an asset is defined in the Hub. The interest rate strategy determines the interest rates for drawn amounts based on the utilization rate of the asset in the Hub. The utilization rate is defined as the ratio of total borrowed amount to the total supplied amount of the asset. As the utilization rate increases, the borrow interest rate increases according to the strategy defined for the asset.

The `AssetInterestRateStrategy` contract implements a piecewise linear model with two slopes. The strategy is defined by four parameters: `optimalUsageRatio`, `baseVariableBorrowRate`, `variableRateSlope1`, and `variableRateSlope2`.

When the utilization rate is below the optimal utilization rate, the borrow interest rate increases linearly from the base variable borrow rate to a rate determined by the first slope. When the utilization rate exceeds the optimal utilization rate, the borrow interest rate increases linearly according to the steeper second slope.

The maximum interest rate is capped at 1000% in the `AssetInterestRateStrategy` strategy, and is determined by

$$maxInterestRate = baseVariableBorrowRate + variableRateSlope1 + variableRateSlope2$$

`AssetInterestRateStrategy` is used by the Hub to define interest strategies for its assets with custom parameters set through restricted Hub function `setInterestRateData()`. Other interest rate strategies can be set with `updateAssetConfig()`.

2.2.7 Liquidations

When a user's health factor falls below 1, their position becomes eligible for liquidation. Liquidators can repay a portion or all of the user's debt in exchange for a corresponding amount of a user's collateral, plus a liquidation bonus. Liquidators are limited in the amount they can liquidate based on the `targetHealthFactor` parameter. A liquidator cannot liquidate more than what would bring the user's health factor above the target.

The liquidation bonus is controlled by three parameters: `maxLiquidationBonus` which varies per reserve, and `healthFactorForMaxBonus` and `liquidationBonusFactor` which are global per Spoke. The bonus starts increasing once a user's health factor drops below the liquidation threshold and begins at `maxLiquidationBonus * liquidationBonusFactor`. From there, it rises linearly as the health factor declines, reaching `maxLiquidationBonus` when the health factor is equal to or below `healthFactorForMaxBonus`.

Liquidations cannot leave less than `DUST_LIQUIDATION_THRESHOLD` amount of debt or collateral in a position, which is defined in terms of value (dollars) as $1000. A requested liquidation amount leaving less debt than this will be increased to the total debt amount, and likewise if the liquidated collateral would be below the threshold after the liquidation, the full collateral amount is liquidated instead. Amount below `DUST_LIQUIDATION_THRESHOLD` can be left in the following conditions:

- A dust amount of collateral can be left after the liquidation if the full debt amount is liquidated.
- A dust amount of debt can be left if the initial requested liquidated debt would leave the corresponding collateral amount below the threshold. In such a case, the full collateral is liquidated, potentially leaving debt below the dust threshold.

2.2.8 Oracles

The Spoke uses the `AaveOracle` contract to query reserve prices. The AaveOracle contains a mapping from reserves to *price sources*, which are contracts exposing the Chainlink *AggregatorV3Interface* (`latestRoundData()`). Only the `answer` return value of the call to the price source is used, the timestamp of the answer is not validated. It is assumed that the price sources themselves will perform any necessary sanitization and validation.

2.2.9 Fees

The protocol collects fees from the interest paid by borrowers to suppliers and from liquidations.

For each asset in a Hub, every time interest accrues, some fee shares are given to the `feeReceiver` of the specific asset. The amount of fee shares is determined by the `liquidityFee` parameter of the asset. The fee shares accrue interest at the same rate as the supply shares.

Liquidation fees are collected during liquidations. A portion of the collateral seized during a liquidation is allocated to the `feeReceiver` of the collateral asset through `payFeeShares()`. The amount of collateral allocated as fee is determined by the `liquidationFee` parameter of the collateral asset. The seized collateral allocated as fee is converted to supply shares at the current share price and given to the `feeReceiver`.

The `feeReceiver` is expected to typically be a `TreasurySpoke` that receives fees and cannot `borrow()`, `withdraw()` or perform a `liquidationCall()`.

2.2.10 Hub & Spoke configurations

The Spoke authority can set global configurations for the Spoke and for its reserves. Additionally, the authority can be transferred to the `SpokeConfigurator` which can only call configuration functions on the Spoke, and it is owned by an `owner` address.

In a Spoke, a reserve can be frozen or paused.

A `HubConfigurator` can freeze or pause an asset:

- Frozen: The supply and borrows caps for each Spoke that is enabled for that asset are set to zero. Operations on the asset are therefore constrained to reduce existing supply and borrow positions.
- Paused: All reserves of this asset in all Spokes are set to inactive. `add()`, `draw()`, `repay()`, `withdraw()`, `reportDeficit()`, `eliminateDeficit()`, `payFeeShares()`, `refreshPremium()`, and `transferShares()` are therefore disabled.

A specific Spoke can also be frozen or paused for all its reserves coming from the Hub:

- Frozen: The supply and borrow caps for all reserves of that Spoke that are linked to an asset from the Hub are set to zero. Operations on these reserves are therefore constrained to reduce existing supply and borrow positions.
- Paused: All reserves of that Spoke that are linked to an asset from the Hub are set to inactive. `add()`, `draw()`, `repay()`, `withdraw()`, `reportDeficit()`, `eliminateDeficit()`, `payFeeShares()`, `refreshPremium()` and `transferShares()` are therefore disabled for these reserves.

2.3 Trust Model

The following roles exist in the protocol:

- `Hub Authority`: The Hub has `restricted` functionality, an `authority` contract defines by which callers these functions can be accessed. The `authority` contract can therefore give permission to configure the Hub and its assets, adding Spokes, setting interest rate strategies, liquidity fee, supply and borrow caps for Spokes, and pausing or freezing assets, and setting the `feeReceiver` and `reinvestmentController`. The `authority` contract is expected to be correctly configured, in a way to limit privileged actions to **fully trusted** roles.
- `Spoke Authority`: The Spokes have `restricted` functionality, an `authority` contract defines by which callers these functions can be accessed. The `authority` contract can therefore give permission to configure the Spokes and their reserves, including setting collateral factors, collateral risk, liquidation parameters such as the max liquidation bonus and fee, and pausing or freezing reserves. The roles that can access these functionalities are **fully trusted**.
- `Spoke Proxy Admin`: **fully trusted**, can upgrade the Spoke implementation to a new version if the Spoke is deployed as an upgradeable proxy.
- **Spokes**: if Spokes are added in a Hub which are not instances of the `Spoke.sol` under review, they are **fully trusted** by the protocol.
- `SpokeConfigurator Owner`: **fully trusted**, can use the SpokeConfigurator to configure Spokes.
- `HubConfigurator Owner`: **fully trusted**, can use the HubConfigurator to configure Hubs.
- `Fee Receivers`: receive fees collected by the protocol. Expected to be instances of the `TreasurySpoke` contract. Otherwise **fully trusted**.
- `Reinvestment Controller`: fully trusted, manages the allocation of assets swept from the Hub to external products and can reclaim them when needed.
- `Position Managers`: fully trusted, can manage user positions on their behalf, including supplying, withdrawing, borrowing, repaying, and liquidating positions. They must be approved by the user to act on their behalf and by the Spoke to be allowed as a position manager.

- `Oracles`: fully trusted to provide accurate prices on-chain.
- `Users`: untrusted, can supply, borrow, repay, withdraw and liquidate positions through Spokes.

Tokens added to Hubs are considered to behave as standard ERC20 tokens, they are non-rebasing, they do not have hooks on transfers, they are not double-entrypoint tokens, and do not implement fees on transfers.

Spokes have full control over the premium drawn values in the Hub. Therefore, Hubs fully trust Spokes to truthfully report premium debt updates to the Hub when a user's risk premium changes. In the worst-case scenario, a malicious Spoke could inflate the amount of premium shares for a given asset in the Hub, which would inflate the value of added shares after accrual allowing the malicious Spoke to drain the Hub liquidity for that asset. Therefore, users using a Spoke must not only trust the Hub but also all other Spokes connected to that Hub that have at least one common asset with the Spoke they are using.

2.3.1 Version 4 changes

Version 4 introduces several changes including:

- Underlying asset in the Hub are now unique, i.e. the same underlying token cannot be added multiple times as different assets in the same Hub.
- `supply()` and `repay()` in the Spoke now pull the underlying tokens from the user, and push them to the Hub, instead of the Hub pulling them directly from the user.
- A new flag `paused` was introduced for Spokes to allow for a more granular pausing mechanism instead of only having `active/inactive`. The `paused` state is similar to `inactive`, but allows `refreshPremium()` to succeed, which is part of the liquidation flow of unrelated reserves in the Spoke.
- The `SpokeConfigurator` holds a reserve limit per Spoke that limits the number of reserves that can be added to a Spoke.
- Dynamic key configurations keys are now `uint24` instead of `uint16`. Moreover, dynamic configs are no longer stored in a ring buffer but are capped to `uint24.max`.
- On liquidations where deficit is reported, the risk premium is now updated to 0.
- Fee shares are no longer minted in `accrue` but in a separate function `mintFeeShares` which is permissioned.
- Liquidators can now choose to receive collateral shares instead of the underlying asset during liquidations.
- `ReserveFlagMap` is now used to store Reserve states in Spokes.
- Premium debt calculations have been optimized by no longer tracking the realized premium fee separately but including it in the premium offset, thus making the premium offset signed.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Open Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	0

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Open Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0

High-Severity Findings	0

Medium-Severity Findings	3

- 2 Wei Rounding Can Trigger Liquidations and Create Additional Debt Code Corrected
- Liquidation Revert Due to Unrelated Paused Asset in Hub Code Corrected
- Share Price Can Decrease Because of Fee Rounding Code Corrected

Low-Severity Findings	9

- Liquidation With Deficit Reverts Due to Paused Assets Code Corrected
- Fees Can Be Minimized by Accruing Often Code Corrected
- Liquidations Can Fail Due to Low Liquidity Code Corrected
- Missing Events Code Corrected
- Overflow in Config Key Makes It Possible to Update CF to 0 Code Corrected
- Profitable Multi-Step Liquidations Specification Changed
- Slippage in Liquidations Specification Changed
- Spokes Do Not Return Minted/Burned Share Amounts Code Corrected
- Array.sort() Can Revert Due to EVM Stack Limit Specification Changed

6.1 2 Wei Rounding Can Trigger Liquidations and Create Additional Debt

Correctness Medium Version 1 Code Corrected

CS-AAVE4-001

When recalculating a user's premium debt, 1 to 2 wei of additional debt can be introduced due to rounding. Although the impact is bounded to 2 wei, it can be enough to make a position liquidatable if the user's health factor is very close to 1, as the added debt may push it just below the liquidation threshold.

This rounding effect can also occur during a liquidation, since the premium debt of a user is recalculated after the liquidation has taken place. As a result, the rounding difference can slightly increase the user's total debt each time a liquidation occurs.

There are two main scenarios where this can be exploited to create bad debt:

1. A liquidator can perform a series of 1 wei liquidations where each action increases the user's debt by 1 or 2 wei. Over time, this effectively creates additional debt, allowing the liquidator to seize all the user's collateral even if the position was originally overcollateralized (collateral value > debt value) before the first liquidation. The profit for the liquidator is then the whole overcollateralization of the loan, instead of just the liquidation bonus.

2. A user can open a small position, and by repeatedly calling `updateUserRiskPremium()` create bad debt. Each iteration increases the `totalAddedAssets()` in the Hub, and this increase can be (partially) extracted as profit from the attacker by being a supplier of that asset. The gains can be maximized by holding a larger fraction of the total supply of the asset. This can be achieved by flashloaning the debt asset and supplying it to Spoke up to the add cap before triggering the repeated rounding errors to create debt. Then, the liquidator can withdraw the supplied assets, repay the flashloan and keep the profits made.

While 1 or 2 wei of assets are insignificant in most cases, the effect could be significant for low-decimal tokens (e.g., WBTC) on chains with low gas prices.

Another scenario where the rounding due to the recomputation of the premium debt could be harmful is in external protocols integrating with Aave V4: There a user of the external protocol could cause the recomputation (and increase in debt) on a position they do not own, possibly leading to liquidations.

Additionally, this rounding behavior can allow a user to open a position that is liquidatable in the same transaction as it is opened.

Code corrected:

The premium debt calculation has been updated to keep the amount of premium debt in full precision (RAY + asset decimal precision) internally. Thus, the 2 wei rounding error no longer occurs when updating the premium debt. Rounding still needs to occur when the premium debt must be converted to asset decimals (e.g., when repaying debt).

6.2 Liquidation Revert Due to Unrelated Paused Asset in Hub

Correctness **Medium** **Version 1** **Code Corrected**

CS-AAVE4-002

The function `pauseAsset` in the `HubConfigurator` contract allows pausing an asset in the Hub by marking all corresponding Spoke configurations for that asset as inactive.

When an asset is inactive for a given Spoke, the following Hub actions are disabled for that asset within the Spoke: `add`, `remove`, `borrow`, `repay`, `reportDeficit`, `eliminateDeficit`, and `transferShares`.

During a liquidation in a Spoke, executed via `liquidationCall()`, the system recalculates the liquidated user's risk premium. This updated premium is propagated to `_notifyRiskPremiumUpdate()`, which calls `hub.refreshPremium()` to update the premium debt for all assets borrowed by the user.

However, if any borrowed asset has been paused (i.e., marked inactive) in the Hub, the premium debt update for that asset will revert, causing the entire liquidation transaction to fail. This occurs because `refreshPremium()` includes a validation check ensuring that the asset is active for the Spoke.

As a result, when a Hub sets an asset to inactive, liquidations involving users who borrowed that asset will revert. This behavior can be exploited: a malicious user could borrow a minimal amount (e.g., 1 wei) of every available reserve in a Spoke, along with their actual desired borrow. If any of those borrowed assets later becomes inactive in the Hub, all liquidations for that user will revert, making their position unliquidatable.

Note that the inactive asset itself cannot be liquidated as `restore` will revert during the liquidation process. Similarly, if the liquidation logic enters the deficit branch, it will also revert as `reportDeficit` will revert for the inactive asset. Thus, the debt in the inactive asset cannot be reduced and the only way to unlock liquidations is to unpause the asset in the Hub.

Code corrected:

An additional flag, `paused`, has been introduced in the Hub for each Spoke configuration. If a Spoke configuration is `paused` and `active` then `hub.refreshPremium()` is allowed to update the premium debt for that Spoke configuration. Liquidations are therefore no longer blocked by an unrelated paused Spoke configuration.

6.3 Share Price Can Decrease Because of Fee Rounding

Design | Medium | Version 1 | Code Corrected

CS-AAVE4-003

One invariant of the system is that the added share price ratio between `totalAddedAssets` and `totalAddedShares` should never drop. However, it is possible for the ratio to decrease because of the rounding of the fee computation.

Every time interest accrues because of a change in `drawnIndex` (as time passes), a portion of that interest is given to the protocol as *liquidity fee*. The portion of interest for the fee is converted to shares, which increase the fee receiver balance. View functions are aware of this behavior and simulate fee accrual and share minting when computing `totalAddedShares`.

Fee shares are computed in the following way:

```
uint256 feesAmount = (asset.drawnShares.rayMulDown(indexDelta) +
  asset.premiumShares.rayMulDown(indexDelta)).percentMulDown(liquidityFee);

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

The returned amount can therefore round down in multiple places, when multiplying with the `indexDelta`, when multiplying with the `liquidityFee`, and when converting to shares with `toSharesDown()`.

If the amount rounds down and does not result in the minting of shares, it can be considered as a donation to the existing shares, and increases their value. However, if the amount causes shares to be minted, the existing shares do not increase value.

Every time `accrue()` is called, the rounding behavior is "realized", and the share price either increases or does not, however, when accessing the vault state through view functions, the behavior is "simulated". It could in a block round down, therefore increasing the total assets, but `accrue()` is not called until the next block. The amount that was considered "donated" in the previous block (but not realized), is now reallocated to mint fees. This appears as a drop of share price.

As an example let us consider a `liquidityFee` of 100%, and unrealistic index values which, however, explain the behavior. At step 2, `accrue()` is not called, the accrual is therefore not "realized".

asset drawnIndex	simulated drawnIndex	drawn Shares	totalAdded Assets	totalAdded Shares
2e+27	2e+27	5	10	5
2.2e+27	2.2e+27	5	11	5
2.4e+27	2.4e+27	5	12	6 (1 minted for fee)

The violation of the monotonicity of share price can cause issues in integrations which follow the initial Aave V4 specification, which specify that share price is only increasing.

Code corrected:

In Version 3, the fee mechanism has been modified such that accrual stashes the fee amount, in asset units, as a separate accounting variable.

6.4 Liquidation With Deficit Reverts Due to Paused Assets

Correctness Low Version 3 Code Corrected

CS-AAVE4-021

`reportDeficit()` ensures that the asset being reported is active and not paused in the Hub.

If a user is being liquidated in a Spoke and the liquidation process enters the deficit branch, `reportDeficit()` is called for each borrowed asset of the user to report any deficits. If any of those borrowed assets are paused in the Hub, the `reportDeficit()` call for that asset will revert, causing the entire liquidation transaction to fail. Thus, liquidations on assets that are not paused can be blocked if the user has borrowed any assets that are now paused. Note that for this to happen, the liquidation must result in bad debt that needs to be reported.

Code corrected

The Hub `reportDeficit` function is now allowed for paused assets, ensuring that this call does not block liquidations of unrelated assets.

Aave Labs states

> The Hub reportDeficit function is now allowed for paused spokes, ensuring that this call is not blocked when it is executed as part of higher-level operations triggered through the Spoke by user actions

6.5 Fees Can Be Minimized by Accruing Often

Correctness Low Version 1 Code Corrected

CS-AAVE4-004

Repeated `asset.accrue()` calls can be used to reduce or suppress protocol fees by forcing the computed amount to round down (possibly to zero) on each call. In `getFeeShares()`, the fee is obtained by rounding down the products of the drawn and premium shares with the index delta, and only then applying the liquidity fee percentage. This ordering enables a loss of fees when accrues are made at very short intervals.

```
uint256 feesAmount = (
    asset.drawnShares.rayMulDown(indexDelta) + asset.premiumShares.rayMulDown(indexDelta)
).percentMulDown(liquidityFee);
```

We illustrate the effect and estimate its monetary impact with a concrete example. Let's assume a chain with a 1-second block time. Consider a liquidity fee of 10% and $1M WBTC as borrowed liquidity. With a 10% APR, `indexDelta` after 1 second will be 3170979198376459264 (~3*10**19) and the amount of drawn shares will be 10**9 assuming 1 WBTC is worth $100k.

Then, with no premium debt, the `feesAmount` will be:

$$feesAmount = (10 * * 9 * 31709791983764592 64/10 * * 27) * 0.1 = 0$$

In general, up to one added share per second can be lost due to this rounding pattern, which corresponds roughly (assuming 1 share = 1 asset) to $30k per year in foregone fees.

This issue is more pronounced on chains with short block times and low fees, and for assets with high values for a single wei of shares (e.g., WBTC).

Code corrected:

In Version 1, the accrued fee rounds down in two locations: the muldiv with `liquidityFee` and the conversion from asset to added shares. In Version 3, `accrue()` no longer mints fee shares, which has been moved to a separate function `mintFeeShares()`. This function is restricted so that griefing here is protected. However, the first muldiv with `liquidityFee` still rounds down, losing a fee amount of up to 1 asset "wei" per block.

Aave Labs acknowledges the possiblity of losing 1 asset "wei" per block but considers it as negligible and that it is not lost, as it is instead forwarded to liquidity providers. Aave Labs states:

> Interest is not lost under any circumstances; it is either accrued as fees for the treasury or forwarded to suppliers. This behavior can be modulated through risk configuration parameters, depending on the specific scenario. To ensure that at least 1 unit of the underlying asset is accrued as fees per block (in the case of accrual occurring every block), the condition `deltaGrowth × liquidityFee > 1` must hold. This may be more challenging for assets with a low number of decimals, as precision constraints increase the likelihood of rounding to zero. However, by adjusting either the liquidityFee or the draw rate, this condition can be consistently satisfied. These parameters are assessed and configured by risk providers to ensure that the treasury accrues fees when appropriate.

6.6 Liquidations Can Fail Due to Low Liquidity

Design | Low | Version 1 | Code Corrected

CS-AAVE4-005

During a liquidation, `hub.remove()` is called to repay the liquidator. The liquidator is awarded part of the liquidated user's collateral in exchange for repaying part of their debt. However, `remove()` can fail if the Hub has insufficient liquidity and additional liquidity cannot be provided because the `addCap` has been reached.

This was not an issue in Aave V3, as the liquidator could choose to receive aTokens instead. In Aave V4, liquidators can only receive the underlying asset, so if the Hub does not have enough liquidity of that asset, the liquidation will fail.

Code corrected:

Liquidations have been updated to allow the liquidator to receive shares instead of the underlying asset.

6.7 Missing Events

Design | Low | Version 1 | Code Corrected

CS-AAVE4-006

1. In `_liquidateDebt` the `premiumDelta` is applied but never emitted. This makes it difficult to track a user's premium position based on events. Another `premiumDelta` is applied to that position later in a liquidation, in _notifyRiskPremiumUpdate(). That `premiumDelta` is emitted, but since the previous one is not, the information about the total premium change during the liquidation is lost.

2. In `_reportDeficit`, the debt of a user is cleared for each borrowed asset, similarly to what happens in `repay()` or `liquidate()`; however, no event is emitted.

Code corrected:

Since Version 3, events are emitted for both.

6.8 Overflow in Config Key Makes It Possible to Update CF to 0

Design Low Version 1 Code Corrected

CS-AAVE4-008

The function `updateDynamicReserveConfig()` forbids setting the `collateralFactor` of an existing Dynamic Reserve Config to `0`, a special value that makes the asset invalid as collateral (it has no borrowing power and cannot be liquidated).

An existing Dynamic Reserve Config, however, can be updated to have a `collateralFactor` of `0` by using `addDynamicReserveConfig()`. `addDynamicReserveConfig()` will create new configurations until `Reserve.dynamicConfigKey` is below `type(uint16).max`. When `type(uint16).max` is reached, the reserve config key will wrap around and start overwriting existing dynamic reserve keys. This allows setting the `collateralFactor` of an existing config to `0`, which would otherwise be forbidden through `updateDynamicReserveConfig()`.

Setting `collateralFactor` to `0` in an existing dynamic config can lead to a position with outstanding debt but an `activeCollateralCount` of `0`, which makes it impossible to report the deficit.

Code corrected:

Aave Labs modified the type of `Reserve.dynamicConfigKey` from `uint16` to `uint24`. Furthermore, `addDynamicReserveConfig()` was updated to prevent overwriting existing dynamic reserve config keys by ensuring that the new key is always smaller than `type(uint24).max`.

6.9 Profitable Multi-Step Liquidations

Correctness Low Version 1 Specification Changed

CS-AAVE4-009

Liquidations are intended to improve the health factor of a user by reducing their debt in exchange for a discounted amount of collateral. However, the health factor can worsen in certain circumstances. If this is the case and the health factor for the maximum liquidation bonus has not yet been reached, a liquidator is incentivized to perform multiple smaller liquidations instead of a single large one. Each smaller liquidation worsens the health factor further, allowing for a larger liquidation bonus to be earned in subsequent liquidations.

First, we will describe under which conditions a liquidation worsens the health factor of a user. Then, we will analyze the conditions under which the maximum liquidation bonus is not yet reached and show that an overlap exists between these two conditions.

The health factor HF_{pre} of a user before a liquidation is given as

$$HF_{pre} = \frac{C_v}{D_v}$$

where C_v is the collateral value, with the collateral factor applied, and D_v is the total debt value.

The health factor after a liquidation HF_{post} is given as:

$$HF_{post} = \frac{C_v - f \cdot x}{D_v - \frac{x}{L_b}}$$

where f is the collateral factor of the collateral being liquidated, x the value of the collateral being liquidated, and L_b the liquidation bonus earned by the liquidator.

The condition for a liquidation to worsen the health factor is

$$HF_{post} < HF_{pre}$$

If we replace HF_{pre} and HF_{post} with their respective formulas, we get

$$\frac{C_v - f \cdot x}{D_v - \frac{x}{L_B}} < \frac{C_v}{D_v}$$

$$\Leftrightarrow C_v \cdot D_v - f \cdot x \cdot D_v < C_v \cdot D_v - C_v \cdot \frac{x}{L_B}$$

$$\Leftrightarrow f \cdot L_B > \frac{C_v}{D_v}$$

$$\Leftrightarrow f \cdot L_B > HF_{\text{pre}}$$

Thus, a liquidation worsens the health factor of a user if the product of the collateral factor and the liquidation bonus is greater than the health factor before the liquidation.

If this condition is satisfied, but the maximum liquidation bonus has not yet been reached, a larger liquidation can be split into multiple smaller liquidations. Each liquidation will have an increasing liquidation bonus, therefore seizing more of the borrower's collateral than intended by the protocol.

Specification changed:

Aave Labs will ensure that:

$$f \cdot max_{L_B} \leq HF_{\text{maxBonus}}$$

Thus, liquidations that worsen the health factor do not create an incentive for multi-step liquidations, because when the health starts to worsen because of a liquidation the maximum bonus has already been reached.

6.10 Slippage in Liquidations

Design Low Version 1 Specification Changed

CS-AAVE4-010

Liquidators will compete with one another to perform profitable liquidations. The auction mechanism in the liquidation bonus means that the liquidation bonus can change unexpectedly if the health of the user being liquidated improves. This creates a scenario where:

1. User A is liquidatable, health factor is such that bonus is `B1`.
2. Liquidator E1 sends a transaction to liquidate A, expecting bonus `B1`.
3. Liquidator E2 sends a transaction to (partially) liquidate A.
4. Transaction of E2 is executed first, health factor of A improves and is now giving bonus `B2 < B1`.

5. Transaction of E1 is executed, bonus is B2, lower than expected, so the liquidator has overpaid for the liquidation.

Liquidators should therefore implement slippage protection in a calling contract, making integration with the protocol more cumbersome.

Specification changed:

Aave Labs will clearly describe the behavior in developer documentation and states it is the responsability of the liquidators to enforce slippage protection. Aave Labs states:

> Liquidations guarantee at least the minimum liquidation bonus, which can increase progressively depending on the user's health factor. It is reasonable to assume that liquidators enforce profitability and slippage constraints, using the minimum bonus as a baseline and estimating the final bonus based on user conditions, such that transactions revert or are not executed if those expectations are not met.

6.11 Spokes Do Not Return Minted/Burned Share Amounts

Design | Low | Version 1 | Code Corrected

CS-AAVE4-011

The Spoke contracts expose core actions such as `supply()`, `withdraw()`, `borrow()`, and `repay()`. However, these functions do not return the actual amounts of shares or assets supplied, withdrawn, borrowed, or repaid. As a result, third-party integrators must independently compute these values, typically by calling additional view functions before and after each operation, or simulating the operations.

A similar issue is present in *NativeTokenGateway* `supplyNative()`, `withdrawNative()`, `borrowNative()`, and `repayNative()`, and in *TreasurySpoke* `supply()` and `withdraw()`.

This makes integration with the system harder and increases the likelihood of errors on the integrator side.

Code corrected:

The Spoke contract has been updated such that `supply()`, `withdraw()`, `borrow()`, and `repay()` now return the actual amounts of shares and assets involved in the operations.

6.12 `Array.sort()` Can Revert Due to EVM Stack Limit

Correctness | Low | Version 1 | Specification Changed

CS-AAVE4-012

When computing the risk premium for a user, the Spoke sorts the collateral assets provided by the user using `Array.sort()` from least to most risky. When 170 or more collateral assets are sorted, the sorting algorithm (quicksort) can revert because the EVM limit of 1024 stack elements is reached. Quicksort is a recursive algorithm, and every internal recursive call increases the stack utilization. The revert happens with 170 elements when the list before being processed by quicksort is initially already in sorted order (ascending risk), or inversely sorted (descending risk), as these conditions provide the worst-case complexity for this variant of quicksort: `O(n^2)` operations, and n recursive calls. A randomly sorted list requires `O(log(n))` recursive calls.

In practice, enabling more than 170 collaterals could mean that a position can no longer be liquidated as `_calculateUserAccountData()` reverts in `liquidationCall()`.

Specification changed:

Aave Labs modified the Spoke Configurator to set a reserve limit on Spokes.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Fee Receiver Has Non-Zero Add Cap

Informational Version 3 Acknowledged

CS-AAVE4-023

In `_addFeeReceiver()`, the add and draw caps are set for a fee receiver. While the draw cap is set to 0, the add cap is set to `MAX_ALLOWED_SPOKE_CAP`. However, the fee receiver is not expected to add liquidity and receiving fees does not require a non-zero add cap.

Acknowledged:

Aave Labs acknowledges the behavior with the following statement:

> The fee receiver is initialized with a non-zero supply cap to facilitate its ability to add assets and use the linked Spoke to deposit additional funds. This configuration can be updated by the Hub owner if deemed necessary, either to block further additions or to regulate this behavior.

7.2 Risk Premium Threshold Can Prevent Borrows and Liquidations

Informational Version 3 Acknowledged

CS-AAVE4-022

`riskPremiumThreshold` is a parameter in the Hub for a Spoke configuration that defines the maximum ratio of premium debt shares to total debt shares that a Spoke can have. This threshold is intended to prevent a malicious Spoke from inflating its premium debt excessively. Currently, there is no minimum value enforced for this parameter. As such, if the collateral risk factor for a collateral listed in a Spoke is larger than `riskPremiumThreshold`, it could lead to a situation where users are unable to borrow due to their premium debt exceeding the threshold immediately after borrowing.

Furthermore, liquidations could fail since the liquidated user's premium shares can increase during a liquidation, and can exceed the threshold after liquidation. The premium shares of a liquidated user can increase if the liquidated collateral is a "safe" one, leaving the remaining debt covered by "unsafe" collaterals.

The value of the `riskPremiumThreshold` should therefore be considered carefully.

Risk accepted:

Aave Labs accepts the risk with the following statement:

> 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 Hub authority.

7.3 `assetId` Cannot Be Determined From Underlying

Informational **Version 3** **Acknowledged**

CS-AAVE4-024

The Hub contract uses an internal mapping from `assetId` to `Asset` which contains the address of the underlying asset. However, there is no reverse mapping from the underlying asset address to the corresponding `assetId`. This might make it difficult for third-party integrators to determine the `assetId` associated with a given underlying asset address.

Acknowledged:

Aave Labs acknowledges the finding.

7.4 `riskPremiumThreshold` Condition Is Slightly Overestimated

Informational **Version 3** **Acknowledged**

CS-AAVE4-025

In `_applyPremiumDelta()`, the premium shares of a Spoke are enforced to be less than or equal to a certain ratio of its drawn shares, defined by `riskPremiumThreshold`. However, the condition is implemented as:

```
spoke.premiumShares <= spoke.drawnShares.percentMulUp(riskPremiumThreshold)
```

This condition allows the premium shares to be slightly higher than the intended threshold due to the rounding behavior of `percentMulUp()`. Specifically, if the product of `spoke.drawnShares` and `riskPremiumThreshold` is not an integer, `percentMulUp()` rounds it up to the nearest integer, effectively allowing a small excess in premium shares.

Acknowledged:

Aave Labs acknowledges the behavior with the following statement:

> The condition is validated using the same rounding direction as the one applied in the Spoke when calculating a user's premium shares based on drawn shares and the risk premium, ensuring consistency and alignment.

7.5 Ambiguous Revert Reasons

Informational **Version 1** **Code Partially Corrected** **Acknowledged**

CS-AAVE4-013

- In `_validateReportDeficit()` and `_validateRestore()`, the `drawnAmount` and the `premiumAmount` are verified to be respectively less than or equal to `drawn` and `premium`. However, both `require` calls will revert with the same error `SurplusDeficitReported`. The error has an argument, which could represent either drawn or premium debt, but with no indication on which one it is. Therefore, which condition failed in case of a revert, and the meaning of the error's argument could be ambiguous.

- In `SignatureGateway.updateUserDynamicConfigWithSig()`, `block.timestamp <= deadline` is checked to ensure that the signature is still valid. However, if this condition fails, the function will revert with `InvalidSignature` error. This error name does not clearly indicate the exact reason of the revert (i.e. an expired signature).

Code partially corrected:

The first issue was addressed by introducing two distinct errors : `SurplusDrawnDeficitReported` and `SurplusPremiumRayDeficitReported` in `_validateReportDeficit()` and `_validateRestore()`.

The second issue remains unaddressed, as the `InvalidSignature` error is still used when the signature has expired.

Acknowledged:

Aave Labs acknowledges the findings and states:

> The first issue has been fixed. For the second, introducing a dedicated error for expired signatures is not considered necessary, as an expired signature is still an invalid signature in all cases.

7.6 Hanging Approval in SignatureGateway

Informational Version 1 Risk Accepted

CS-AAVE4-015

Function `repayWithSig()` of *SignatureGateway* computes the `repayAmount` by taking the minimum of the `amount` passed by the user as an argument and the user's actual debt in the Spoke:

```
uint256 repayAmount = MathUtils.min(
    amount, _spoke.getUserTotalDebt(reserveId, onBehalfOf));
```

The calculated `repayAmount` is therefore possibly lower than the `amount` passed as an argument, which should correspond to the user's approval of the underlying to the contract. The `transferFrom()` call is therefore likely to leave a hanging approval from the user to the contract.

```
underlying.safeTransferFrom(onBehalfOf, address(this), repayAmount);
```

Risk accepted:

Aave Labs accepts the risk with the following statement:

> The amount of debt to be repaid is adjusted to allow the function to fully repay the user's outstanding debt. It is assumed that a user performing a max repayment is aware that a residual allowance may remain, as the approved amount must exceed the current debt at the time of transaction signing in order to cover interest accrued between signing and confirmation. Furthermore, since the function returns the amount of assets repaid, a subsequent action can be performed to adjust any remaining allowance, if required.

7.7 Inconsistent Variable Names

Informational | Version 1 | Acknowledged

CS-AAVE4-016

- In `SignatureGateway`, some functions use `onBehalfOf` as parameter name (e.g. `setUsingAsCollateralWithSig()`), while others use `user` (e.g. `setSelfAsUserPositionManagerWithSig()`) to refer to the same address.

Acknowledged:

Aave Labs acknowledges the finding.

7.8 Low Granularity in `drawCap` and `addCap`

Informational | Version 1 | Risk Accepted

CS-AAVE4-018

The `drawCap` and `addCap` parameters offer low granularity for expressing cap amounts. The minimum granularity is 1 unit of an asset (`10**asset.decimals`). This granularity can be very different between assets (e.g., $1 for USD stablecoins vs. $100k for BTC derivatives). In the case of high-value assets, this prevents setting precise caps.

Risk accepted:

Aave Labs accepts the risk with the following statement:

> Caps are considered appropriate to be expressed in asset terms as a best-effort measure, given that total supplied assets and total drawn amounts may exceed the limits over time due to accrued interest.

7.9 Multicall Operations Can Be Front-Run and Forced to Revert

Informational | Version 1 | Acknowledged

CS-AAVE4-007

The `SignatureGateway` contract inherits from `Multicall`, allowing users to execute multiple operations in a single batched transaction. However, many of the actions supported by `SignatureGateway` rely on off-chain signatures that can be used by anyone who possesses them.

If another party submits the same signed message before the original caller's transaction is confirmed, the signature becomes invalid for subsequent calls (e.g., due to nonce consumption or replay protection).

As a result, any transaction batching multiple operations via `multicall()` can be front-run and reverted, because a single operation in the batch (using the same signature) will fail.

For example, the following flow can be problematic:

- A relayer sends a transaction for Multicall to execute three intents from a user: [A1, A2, A3].
- An attacker front-runs the call, and executes [A1].
- The batch from the relayer reverts, [A2, A3] are not executed.

If this is not handled correctly by the relayer, the intents of a user can remain partially fulfilled. An attacker could have some incentive to censor some user intents: for example they might profit from a delayed repayment through increased interest accrual, or because of the victim becoming liquidatable. Therefore, attention is required when implementing systems that relay user signatures on-chain, such that user intents are not left partially fulfilled.

Acknowledged:

Aave Labs acknowledges the behavior and states:

> The `SignatureGateway` is not intended to function as a transaction builder. Users should not rely on this contract to batch multiple actions into a single transaction, as it operates on a best-effort basis. Additionally, there is no economic benefit to performing such batching in this context, making the described behavior a griefing vector only. In any case, the user can retry the action if necessary.

7.10 No On-Chain Reverse Mapping for `reserveId`

Informational Version 1 Acknowledged

CS-AAVE4-019

In the Spoke, it is possible to obtain the `assetId` and `Hub` address from a `reserveId`, but it is difficult to check if a given `(assetId, hub)` pair is already listed in the Spoke, and with what `reserveId`, without iterating through all reserves.

An inverse mapping could be made available by expanding the internal `_reserveExists` function for this purpose.

Acknowledged:

Aave Labs acknowledges the finding.

7.11 `preview*` and `convert*` Return 1:1 Exchange Rate for Unsupported Assets

Informational Version 1 Risk Accepted

CS-AAVE4-020

In the `Hub` contract, functions such as `previewAddByShares()` or `convertToAddedShares()` will not revert if the asset passed as a parameter is not yet added to the Hub. Instead, they will return amounts corresponding to an exchange rate of 1:1 due to the virtual shares mechanism. This can lead to problems for integrators who expect these functions to revert in such cases.

Note that `drawnAssets`/ `drawnShares` preview and conversions will always return `0` for an unsupported asset as the `drawnIndex` will be zero for it.

Changes in Version 3:

`convertTo..` functions are removed in Version 3 thus this only applies to `preview*` functions.

Risk accepted:

In (Version 5) Aave Labs accepts the remaining risk with the following statement:

> Integrators should not rely on preview functions returning `1` to determine whether an asset is listed, and should instead use explicit getters such as `isUnderlyingListed`.

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 Liquidation Gas Cost Is Possibly More Than Reward

Note **Version 1**

Liquidations must iterate through all the borrowed assets and collaterals of the liquidated user. For each asset, the process involves loading the asset configuration, querying an oracle, loading user balances, and querying the Hub to convert share balances to assets. Every iteration consumes some gas, and the number of iterations is only capped by the number of enabled reserves in the spoke.

The number of iterations is potentially unbounded because:

- there is no maximum number of reserves.
- there is no maximum number of collaterals or borrowed assets per user.

So in practice, a borrower could enable all collaterals and all borrowable assets on their account to maximize the liquidation cost.

Moreover, the cost of each iteration as a portion of the liquidation reward is not bounded. There is no minimum debt size for a borrowed asset, therefore the liquidation incentive could be smaller than the liquidation gas cost. A borrower could split a position into several smaller ones to make it less attractive to liquidate.

The positions could be created when gas prices are low; however, liquidations often need to happen during times of high asset price volatility when gas prices are high. Therefore, the gas price for the borrowers is less than for the liquidators.

As an example, let us consider a position with 10 borrowed assets and 10 collateral assets. For simplicity, only 1 of the borrowed assets has a significant debt balance, and only one of the collaterals has a significant balance, and the liquidator therefore only needs to perform a single liquidation.

Let us assume gas prices are 1 Gwei, ETH price is $3000, liquidation bonus is 5%, and every asset enabled by the borrower costs 80'000 gas to iterate through.

The gas cost of the liquidation is therefore:

```
20 (assets enabled) * 80k (gas per asset) * 10^-9 (ETH per gas) * 3000 (USD p
er ETH) == $4.8
```

With a 5% liquidation bonus, debt smaller than `~$100` becomes therefore unprofitable to liquidate.

Despite the asymmetry of gas prices between the time of creation of the position and the time of liquidation, the borrower uses considerably more gas to create the position than what is used to calculate their health. Creating the position involves setting storage slots for the first time, which is considerably more gas intensive than reading (cold) storage slots.

Aave Labs confirms that this behavior is intended and adds the following statement:

> The described scenario occurs only when the liquidation bonus is lower than the gas cost required to execute the liquidation, which represents an edge case. This would require a combination of unlikely conditions, such as exceptionally high network gas prices and user positions composed of many collateral and borrow reserves. Even if this situation arises, the position can still be liquidated at a loss in order to resolve it. Additionally, the number of listed reserves in a Spoke can be

restricted through off-chain coordination or enforced via the *SpokeConfigurator*, with the goal of reducing the likelihood of complex user positions involving a large number of reserves.

8.2 Paused Assets Keep Accruing Interest

Note Version 1

In the `HubConfigurator` contract, `pauseAsset()` can be used to pause a specific asset in the Hub. This leads to all Spoke configs for this asset being set to inactive.

When an asset is paused, no actions can be performed with this asset in the Spokes, e.g., no new loans can be opened, no new deposits can be made, and no repayments can be performed. However, interest on existing loans continues to accrue. Users that have an open loan in such an asset cannot repay their loan, as repayments are disabled for paused assets.

Users are therefore forced to let their loan continue to accrue interest.

Typically, such a situation is not expected to last long, and interest accrual during negligible time frames is not a big issue. However, if an asset remains paused for a long time, this can lead to significant interest accrual on existing loans that users cannot repay.

Aave Labs confirms that this behavior is intentional, and states:

> Pausing an asset is not expected to be a long-lasting condition, and therefore the interest accrued during that period is not considered significant enough to cause harm to the protocol. By contrast, if the interest generated during the pause is deemed problematic, the Spoke owner can update the interest rate to mitigate its impact.

8.3 Tokens With Hooks Are Not Supported

Note Version 1

Tokens that implement callbacks on transfers (for example to the receiver), such as tokens adhering to the ERC-777 standard, are not supported by Aave V4. Aave V4 does not implement reentrancy guards and sometimes violates the CEI (checks - effects - interactions) pattern. For example, during liquidations, the liquidated collateral is transferred to the liquidator before updating the debt of the position. Therefore, during the interaction with the external contract, the liquidated position is in an inconsistent state.

This can be an issue if the liquidated collateral implements arbitrary logic on transfer, for example, calls to the receiver. Tokens should therefore be thoroughly vetted to not implement arbitrary logic during transfers.

8.4 Type Bound Considerations

Note Version 1

- The `drawnIndex` has been modified from a `uint128` to a `uint120` in Version 2. This implies that for an asset at 30% APR (compounded daily) the `drawnIndex` will overflow after approximately 70 years. This is a theoretical limit and in practice it is unlikely that an asset will remain in the protocol for such a long time without any intervention.
- Premium debt is stored with added precision (RAY) which is 27 extra decimals of precision. As `premiumOffset` is stored as a `int200`, possibility of overflow/underflow should be considered. Premium risk factor can be at most 1000%, i.e. 10 times more premium shares than drawn shares, so we can have the following:

- 27 decimals precision takes log2(10^27) = 89.69 bits
- 1000% (10x) premium risk takes up to log2(10) = 3.32 bits
- sign of the integer takes 1 bit

We are left with 200 - 89.69 - 3.32 - 1 = 105.98 bits for the maximum amount of borrowed assets (`drawnShares * drawnIndex`). This corresponds to 31 decimal digits. For tokens with high circulating supply (in terms of balance amounts), such as PEPE, balances of the order of 10**31 are feasible. The total supply of PEPE is 4.2 * 10**32.

Aave Labs agrees that balances of the order of 10**31 are technically feasible in the Hub, but points out that with the current Spoke implementation, tokens with such a high balance are not feasible. Aave Labs states:

> While such values may be theoretically possible from the Hub's perspective, the Spoke only supports oracles with 8 decimals. As a result, the minimum representable price is 1e-8, implying that a balance of 10^31 would correspond to 10^23 USD, which is not considered a realistic or feasible amount.

- `drawCap` and `addCap` are represented as `uint40` which limits the cap to `2**40 - 1` units of asset which is about 1 Trillion. For assets with low unit price the caps might not be able to grow enough to accommodate a significant amount of liquidity.