CHAINSECURITY

PUBLIC

Code Assessment
of the Diamond PAU v1.15 Smart Contracts

November 02, 2026

Produced for

Sky



Contents

1	Executive Summary	3
2	Assessment Overview	4
3	System Overview	7
4	Trust Model	10
5	System Considerations	12
6	Terminology	13
7	Findings	14
8	Limitations and use of report	16

1 Executive Summary

Dear all,

Thank you for trusting us to help Sky with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Diamond PAU v1.15 according to **Scope** to support you in forming an opinion on their security risks.

Sky implements a new facet integrating Aave V4 into the PAU system. `AaveV4Facet` supplies underlying held by the ALMProxy into an Aave V4 market and withdraws it back, bounded by rate limits keyed on the market and, on deposits, by a slippage tolerance configured per market and a bad debt tolerance configured per hub asset. In **Version 2**, the `PAUInit` library, which wires a PAU stack from a governance spell, entered the scope. In **Version 3**, the wiring of every facet moved on-chain into `BeaconConfig` and the corresponding `I<FacetName>Controller` dispatch interfaces entered the scope.

The most critical subjects covered in our audit are functional correctness, security of assets and external integrations. The deposit path additionally requires the bad debt Aave V4 reports for the asset to stay within a tolerance configured per hub asset, which initially blocked deposits for reasons unrelated to the market being used, see **Bad Debt Too Small to Be Worth Liquidating Blocks All Deposits**. Security regarding all the aforementioned subjects is high.

The general subjects covered are documentation, trustworthiness and event handling. The codebase features extensive NatSpec and a dedicated integration document, and underlying only ever moves between the ALMProxy and a market the admin has configured. Security regarding all the aforementioned subjects is good.

In summary, we find that the codebase provides a high level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't 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

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 assessment was performed on the source code files inside the Diamond PAU v1.15 repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

Version	Date	Commit Hash	Note
1	17 August 2026	6229e7627d995527e561a3da8f6547b83455c780	v1.15.0-beta.0
2	8 September 2026	b39a94d0f702d9bcd217ffc463c0189421a5898f	PAUInit initial review
3	11 September 2026	1b6743a6c2d9e29d9e4e107d92e70a4f20a51423	v1.15.0-beta.1
4	21 September 2026	71604ddcee14cff3561fff66eac17e749c66384b	v1.15.0-beta.2
5	22 September 2026	82992d704e9533bd277c826be19e9e6ec794b6ca	PAUInit fix review
6	21 October 2026	c5c961a8bd5568c7ca73ba1151b23ab9c7cd9113	v1.15.0

For the Solidity smart contracts, the compiler version `0.8.34` was chosen.

This report covers two review streams: **Version 1**, **Version 3** and **Version 4** assessed the changes introduced on top of the previously reviewed `v1.14.0` release in the client's repository, while **Version 2** and **Version 5** assessed the `PAUInit` library against the River One Eco fork. They converge in **Version 6**, where the reviewed `PAUInit` entered the client's repository unchanged.

Across both streams, the following files are in scope:

```
deploy/
└── PAUInit.sol
src/
├── facets/
│   ├── aave/
│   │   └── IAaveController.sol
│   ├── aave-v4/
│   │   ├── AaveV4Facet.sol
│   │   ├── IAaveV4Facet.sol
│   │   └── IAaveV4Controller.sol
│   ├── basin/
│   │   └── IBasinController.sol
│   ├── cctp/
│   │   └── ICCTPController.sol
│   ├── centrifuge/
│   │   └── ICentrifugeController.sol
│   ├── curve/
│   │   ├── CurveFacet.sol
│   │   └── ICurveController.sol
│   ├── dai-usds/
│   │   └── IDAIUSDSController.sol
│   ├── erc4626/
```

```
    │   └── IERC4626Controller.sol
    ├── erc7540/
    │   └── IERC7540Controller.sol
    ├── ethena/
    │   └── IEthenaController.sol
    ├── farm/
    │   └── IFarmController.sol
    ├── layer-zero/
    │   └── ILayerZeroController.sol
    ├── maple/
    │   └── IMapleController.sol
    ├── merkl/
    │   └── IMerklController.sol
    ├── nfat-halo/
    │   └── INFATHaloController.sol
    ├── nfat-prime/
    │   └── INFATPrimeController.sol
    ├── otc/
    │   └── IOTCController.sol
    ├── pendle/
    │   └── IPendleController.sol
    ├── psm/
    │   └── IPSMController.sol
    ├── psm3/
    │   └── IPSM3Controller.sol
    ├── spark-vault/
    │   └── ISparkVaultController.sol
    ├── superstate/
    │   └── ISuperstateController.sol
    ├── transfer-asset/
    │   └── ITransferAssetController.sol
    ├── uniswap-v3/
    │   ├── UniswapV3Facet.sol
    │   └── IUniswapV3Controller.sol
    ├── uniswap-v4/
    │   ├── UniswapV4Facet.sol
    │   ├── IUniswapV4Facet.sol
    │   └── IUniswapV4Controller.sol
    ├── usds/
    │   └── IUSDSController.sol
    ├── weeth/
    │   └── IWEETHController.sol
    ├── wrap-proxy-eth/
    │   └── IWrapProxyETHController.sol
    └── wsteth/
        └── IWSTETHController.sol
└── libraries/
    ├── BeaconConfig.sol
    └── RateLimitHelpers.sol
```

For `RateLimitHelpers.sol`, only the newly added key helpers `makeAddressUint256Key` and `makeAddressUint256AddressUint16AddressKey` are in scope.

In **Version 2**, `deploy/PAUInit.sol` was added to scope. `PAUInit` is the initialization library a governance spell uses to wire a deployed PAU stack.

In **Version 3**, the following was added to scope:

- `BeaconConfig.sol` and the `I<FacetName>Controller` interfaces of all facets were added to the scope, except `setDualPoolIntegration()` and `IDualPoolController.sol`.

- For `CurveFacet.sol` and `UniswapV3Facet.sol`, only the comments corrected in **Version 3** are in scope.
- For `UniswapV4Facet.sol` and `IUniswapV4Facet.sol`, only the change of `swap()` to take a `PoolKey` instead of a pool identifier, and the rejection of pools with a hook in the liquidity functions, are in scope. **Note** these changes of `UniswapV4Facet` have been reverted in **Version 4**.

2.1.1 Excluded from scope

All other contracts in the repository. However, these contracts have been covered in other reviews. The Aave V4 protocol, which the `AaveV4Facet` integrates with, is out of scope and assumed to behave as documented. It has been covered in a separate ChainSecurity engagement. Additionally, all dependencies, scripts test and similar are out of scope.

3 System Overview

This system overview describes the complete changes introduced in the contracts as defined in the Assessment Overview.

3.1 Aave V4 Integration

Aave V4 features a hub and spoke architecture:

- A **Hub** custodies the pooled liquidity of an asset and performs the share accounting, keyed by an `assetId`.
- A **Spoke** is the market that users interact with. It maps a `reserveId` to the triple `(underlying, hub, assetId)`, records each user's supplied position and enforces the controls defined on the reserve (pause, freeze).

A market is therefore identified by the pair `(spoke, reserveId)`. Supplying mints no transferable receipt token: the spoke credits the supplier with a share balance held in its own storage, readable as shares through `spoke.getUserSuppliedShares(reserveId, user)` or, converted to underlying at the current exchange rate, through `spoke.getUserSuppliedAssets(reserveId, user)`.

The integration only supports supplying and withdrawing: it supplies underlying held by the ALMProxy into a market and withdraws it back to the ALMProxy. It does not borrow, does not enable the position as collateral, and never holds an intermediate token. The ALMProxy always acts for its own position (`onBehalfOf == ALMProxy`), which Aave V4 authorises implicitly, so it does not have to be onboarded as a position manager.

Because there is no receipt token whose balance could be measured, both operations size their accounting from an observed state delta rather than from the values the spoke returns: the change in the supplied position on deposit, and the change in the ALMProxy underlying balance on withdrawal.

3.1.1 AaveV4Facet

The facet is delegatecalled by the Controller and keeps its own ERC-7201 namespaced storage, holding the deposit tolerance `maxSlippages[spoke][reserveId]` in 1e18 precision and the tolerated deficit `maxDeficits[hub][assetId]` in RAY (1e27) precision in the asset's own units.

The following functions are gated on `DEFAULT_ADMIN_ROLE`:

1. **`setMaxSlippage(spoke, reserveId, maxSlippage)`**: configures the deposit tolerance of a market, expressed in 1e18 precision, where a higher value is stricter and `1e18` demands a credited position at least equal to the deposited amount. `spoke` must not be the zero address; the value itself is not bounded from above. Setting the tolerance back to `0` disables deposits into that market and leaves withdrawals untouched.

2. **`setMaxDeficit(hub, assetId, maxDeficit)`**: configures how much outstanding deficit a deposit may buy into, expressed in RAY precision in the asset's own units, so a 1,000 unit tolerance on a 6-decimal asset is `1_000e6 * 1e27`. `hub` must not be the zero address; the value itself is not bounded from above. It defaults to `0`, under which any deficit blocks deposits. The tolerance is keyed per hub asset rather than per market, matching the scope over which the hub reports the deficit, so every market fronting that asset reads one value.

The following functions are gated on `ALLOCATOR_ROLE`:

1. **`deposit(spoke, reserveId, amount)`**: requires that a `maxSlippage` is configured for the market, resolves `(underlying, hub, assetId)` through `spoke.getReserve(reserveId)`, requires the hub to hold the same underlying under `assetId` (`hub.getAssetUnderlyingAndDecimals(assetId)` returns `underlying`), and requires the hub's deficit for the asset not to exceed the configured tolerance

(`hub.getAssetDeficitRay(assetId) <= maxDeficits[hub][assetId]`). It then decrements the deposit rate limit by the requested `amount`, approves `underlying` from the ALMProxy to the spoke, and calls `spoke.supply(reserveId, amount, proxy)`. The credited position is measured as the change in `getUserSuppliedAssets` and must satisfy `amountReceived * 1e18 >= amount * maxSlippage`. The approval is reset to zero afterwards. Aave enforces its own supply cap per spoke and asset, so a deposit can revert there independently of the rate limit.

2. **`withdraw(spoke, reserveId, amount)`**: resolves the reserve, snapshots the ALMProxy `underlying` balance and calls `spoke.withdraw(reserveId, amount, proxy)`. The spoke caps the request at the full position, so passing `type(uint256).max` exits the position entirely. The measured balance delta `amountWithdrawn` decrements the withdraw rate limit and is then credited back to the deposit rate limit through `_tryIncreaseRateLimit`, which does nothing when the deposit key is unconfigured. Withdrawals require no configured `maxSlippage` and are not gated on the deposit key.

The deficit gate reflects that an Aave V4 deficit, which records socialised bad debt, never marks down the supplier share price. Supplying into an asset that carries one would buy into the pool at par against unbacked debt, so deposits are blocked until the deficit falls to or below the tolerance, which is zero unless governance sets it.

Additionally, `getMaxSlippage`, `getMaxDeficit`, `getDepositRateLimitKey` and `getWithdrawRateLimitKey` expose the configured tolerances and the derived rate limit keys.

3.1.2 Rate Limits

Rate limit keys act as the whitelist of markets: a market can only receive capital once governance has configured a limit for the exact key the facet derives.

Operation	Key tuple	Rate limit identifier
`deposit`	`(spoke, reserveId, hub, assetId, underlying)`	`LIMIT_AAVE_V4_DEPOSIT`
`withdraw`	`(spoke, reserveId)`	`LIMIT_AAVE_V4_WITHDRAW`

The deposit key embeds the values read from the reserve on top of the market identity: `underlying` is the token the facet approves and transfers, `hub` is the target of the deficit gate, and `assetId` is the argument passed into it. The spoke sets all three when the reserve is listed and cannot update them afterwards, so they change only if the spoke is upgraded; such a change invalidates the key and deposits stop until governance configures a limit for the new tuple.

The withdraw key embeds none of them. Withdrawals return funds to the ALMProxy and are sized by the measured balance delta. A reserve remapped between deposit and withdrawal can still be exited.

Two key builders were added to `RateLimitHelpers.sol` for these keys, `makeAddressUint256Key` and `makeAddressUint256AddressUint16AddressKey`, which hash the rate limit identifier together with the key components.

3.2 BeaconConfig

`BeaconConfig` holds the wiring of every facet, which was previously performed by the deploy repository. Per facet, `set<FacetName>Integration` builds the list of `(controller selector, facet selector)` pairs and registers it on the Beacon under a `bytes32` integration identifier. The external interfaces are defined in the corresponding `I<FacetName>Controller` interfaces.

3.3 PAUInit

`PAUFactory` deploys the components of a PAU stack with governance as sole admin, but it does not connect them. A fresh stack is inert: the `Controller` holds no `CONTROLLER` role on the `ALMProxy` or its `RateLimits`, and no integrations are synced, so it can neither move funds nor expose allocator entry points. Wiring it up is a governance action, and `PAUInit` is the library used by the spells that perform it. Its functions are internal, so they are inlined into the spell action and execute as the governance proxy (`PauseProxy` or `SubProxy`), the address holding `DEFAULT_ADMIN_ROLE` on every component.

`init()` makes a deployed `Controller` operational. It grants the `Controller` the `CONTROLLER` role on the `ALMProxy` and on its `RateLimits`, so it can route calls through the proxy that custodies the funds and consume rate limit capacity, and then syncs the integrations that give it its allocator entry points. Since several `Controller`s can share one `ALMProxy`, `init()` is not only a bootstrap step: it can also grant a second `Controller` authority over a proxy that is already live and holding funds. The grants happen only after a set of sanity checks on the `PAUInstance` the spell passed. The `Controller`'s immutable wiring (`accessControls`, `proxy`, `beacon`, `rateLimits`) must match the values governance expects, and governance must itself hold `DEFAULT_ADMIN_ROLE` on the `AccessControls`, the `ALMProxy` and the `RateLimits`. The `AccessControls` must further have exactly one admin and no `ALLOCATOR_ROLE` members, so governance neither shares control of the stack with a leftover deployment key nor attaches a `Controller` to an `AccessControls` that already serves one.

`addAllocator()` grants `ALLOCATOR_ROLE` to an agent on the stack's `AccessControls`. It keeps the consistency check from `init()`, requiring the `Controller` to agree with the `AccessControls` it is about to write to, and rejects the zero address. `setIntegrations()` and `removeIntegrations()` keep neither guard: they forward straight to the `Controller`.

3.4 Changelog

In **Version 2**, the `PAUInit` library entered the scope, see PAUInit.

In **Version 4**, the changes made to the `UniswapV4Facet` in **Version 3** were reverted: `swap()` takes a `bytes32` pool identifier again instead of a `PoolKey`, and `mintPosition()`, `increasePosition()` and `decreasePosition()` no longer reject pools with a hook. Effectively, the `UniswapV4Facet` is without changes (except some comment corrections, which remain).

In **Version 6**, the following changes were made:

- `AaveV4Facet.deposit()` requires the underlying the hub holds under `assetId` to match the reserve's underlying, so that the deficit gate reads the deficit of the asset actually supplied.
- The `PAUInit` library entered the client's repository, unchanged from the version reviewed in **Version 5**.

4 Trust Model

This trust model covers only the components introduced by the Aave V4 integration and the `PAUInit` library. The trust model of the underlying PAU framework (Controller, ALMProxy, RateLimits, facet wiring) is described in the full Diamond PAU report.

4.1 Roles

Admin (`DEFAULT_ADMIN_ROLE`): Fully trusted. It configures the rate limits, through the new `setMaxSlippage`, the tolerance every deposit is measured against, and the tolerance of deficit for a deposit with `setMaxDeficit`. A tolerance that is too loose admits deposits crediting materially less than the amount supplied. Values at or above `1e18`, which the facet does not reject, require the measured position delta to reach the amount supplied. Neither can move funds out of the system. The tolerance is held per market, so it can be set independently for each reserve on a spoke.

Allocator (`ALLOCATOR_ROLE`): Partially trusted. Within the configured rate limits it controls the timing and sizing of all flows between the ALMProxy and Aave V4. It cannot direct funds to arbitrary addresses: underlying only ever moves between the ALMProxy and a market, and a market can only receive capital once the admin has configured both a tolerance and a deposit rate limit for it, while an exit requires only a withdraw limit. The worst it can do is griefing through mistimed or unwanted flows into whitelisted markets.

`PAUInit` holds no roles of its own. It has no storage, no deployed address and no callers other than the spell it is inlined into, so the two roles below belong to the surrounding system.

Governance (`PauseProxy` or `SubProxy`): Fully trusted. It executes the spell and holds `DEFAULT_ADMIN_ROLE` on the `AccessControls`, the `ALMProxy` and the `RateLimits` of the stack. Through the library it grants the `CONTROLLER` role on the `ALMProxy` and the `RateLimits`, grants `ALLOCATOR_ROLE` on the `AccessControls`, and syncs or removes integrations on the `Controller`. At worst it grants `CONTROLLER` on a funded `ALMProxy` to an arbitrary contract, which can then move all custodied assets: the library constrains the addresses in the `PAUInstance` only against each other, never against an external reference.

Spell author: Fully trusted, since the spell executes with governance's authority. It supplies the `PAUInstance` and the integration IDs, which are assumed to be taken from the verified deployment output of the stack being initialized. A `PAUInstance` naming a different but internally consistent stack passes every sanity check, so the roles are granted on components governance did not intend. The checks cannot establish that a stack is in a clean state either: the `ALMProxy` and the `RateLimits` are not enumerable, so additional admins or `CONTROLLER` holders granted before the handover to governance cannot be detected, and the role admins of the `AccessControls` are not checked. Spell reviewers are therefore expected to verify the role history of every component, for example from emitted events, in addition to the inputs.

4.2 External Components

Aave V4 spoke and hub: Every spoke and hub the facet touches is an upgradeable proxy under the control of Aave governance. They are fully trusted with the supplied position: a hostile or faulty upgrade can take it in full, and the facet has no defence against its own counterparty. Exposure is bounded by position sizing and by the deposit rate limit of the market rather than by facet code. Short of an upgrade, the spoke and hub retain several powers within the trust model:

- **Withdrawal availability**: an exit reverts when the hub lacks liquidity, when the reserve is paused on the spoke, or when the hub has deactivated or halted the spoke. Pausing the asset on the hub disables withdrawals in every spoke that lists it. A *frozen* reserve still permits withdrawal, as freezing blocks supply and borrow only. The hub's reinvestment controller, itself fully trusted, may additionally deploy idle liquidity elsewhere, reducing what is immediately withdrawable without changing the recorded position; that liquidity

still counts as available when the hub computes utilisation, so utilisation does not indicate what can be withdrawn.

- **Deficit**: socialised bad debt is recorded as a hub deficit and never marks down the supplier share price. The loss surfaces as a shortfall in exit liquidity, borne by the last suppliers out. The deficit gate stops new capital from buying into unbacked debt at par, but it does not protect the position already held; monitoring hub liquidity against position size therefore remains an operational requirement. Clearing a deficit requires a whitelisted spoke to burn its own added shares, so how long it blocks deposits is outside the facet's control.
- **Reserve mapping**: a reserve's `underlying`, `hub` and `assetId` are set when it is listed and cannot be updated afterwards, so they change only if the spoke is upgraded. Such a change invalidates the deposit rate limit key, so no budget configured for the original mapping can be spent against the new one. Withdrawals remain available regardless.
- **Other spokes on the same hub**: the hub runs no solvency checks on its spokes and trusts them to report their debt truthfully, so a spoke that is malicious or compromised can inflate the value of added shares and drain the hub's liquidity for an asset it shares. Trusting a market therefore extends to every other spoke connected to the same hub for the same asset, including spokes that are not instances of the reviewed implementation.

Underlying assets: Assumed to be standard ERC-20 tokens that neither rebase nor charge a transfer fee. Both operations measure an observed delta rather than trusting the spoke's return values. On withdrawal that delta is what the rate limit is decremented by, so a misreported return value cannot skew the accounting; on deposit the rate limit is decremented by the requested amount and the delta is checked against the slippage floor.

5 System Considerations

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

SC1 Facet Dispatch Interfaces

System Consideration | Version 3

The `I<FacetName>Controller` interfaces implement the external "dispatch" interfaces which `BeaconConfig.sol` wires. A few considerations:

- The wire order differs from the deploy repository. `BeaconConfig` wires the same selectors as the `_deployAndWire<X>Facet` functions there, but in a different order for all facets both configure. Dispatch is unaffected.
- The `I<FacetName>Controller` drop parameter names their facets declare, most often on return values. This is cosmetic only.

6 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

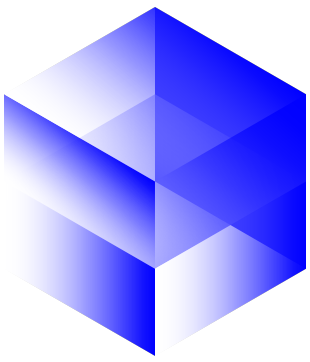
We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

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.

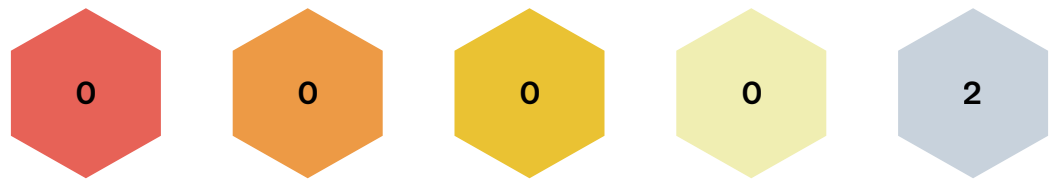
7 Findings

In this section, we describe the findings identified during the course of the engagement. The findings are categorized by severity as explained in the Terminology section. Below we provide an overview of the total number of findings per severity, as well as the number of open issues.



No Open Issues

Total number of findings



Critical | High | Medium | Low | Informational

7.1 Overview

The following table lists all identified findings along with their severity and current status. A finding is considered resolved once it has been fully corrected or the specification has been changed accordingly. Non-informational findings with any other status, including partial fixes, acknowledgements, and accepted risks, remain open.

Info	#001	`init()` Does Not Check for Existing `ALLOCATOR_ROLE` Members	Code Corrected
Info	#002	Bad Debt Too Small to Be Worth Liquidating Blocks All Deposits	Code Corrected

7.2 Descriptions

#001 `init()` Does Not Check for Existing `ALLOCATOR_ROLE` Members

Informational **Version 2** **Code Corrected**

Several `Controller`s can share an `ALMProxy` and a `RateLimits`, so `init()` can attach a new `Controller` to a stack that is already in use. `AccessControls` is enumerable, and `init()` relies on that to require a single `DEFAULT_ADMIN_ROLE` holder, but it does not check `ALLOCATOR_ROLE`. Members there would indicate the `AccessControls` already serves a `Controller`.

Code Corrected

In **Version 5**, `init()` requires the `AccessControls` to have no `ALLOCATOR_ROLE` members.

#002 Bad Debt Too Small to Be Worth Liquidating Blocks All Deposits

Informational **Version 1** **Code Corrected**

`deposit()` reverts unless the reserve's Hub reports exactly zero deficit for the asset.

Aave V4 prevents bad debt through liquidation incentives: once a position falls below its required collateralisation it is liquidated at a bonus, which happens well before the collateral stops covering the debt. That incentive scales with position size, while a liquidator's transaction cost does not. Below the size at which the bonus covers that cost no liquidator has a reason to act, and Aave V4 enforces no minimum position size, so such a position might be left alone and keep sinking until the collateral no longer covers the debt. Whenever it is eventually liquidated, the shortfall is recorded as a deficit on the Hub.

The amount involved is dust to the protocol. A few base units of unpaid debt against a Hub holding millions changes nothing for Aave, and stays in the supplier share price rather than marking existing suppliers down. `deposit()` strictly requires that no such debt exists. Bad debt worth a fraction of a cent, arising in any market on that Hub, then blocks deposits into a market of the integration's own that is unaffected.

Code Corrected

`deposit()` now accepts a deficit up to a per-Hub-asset tolerance set by governance through `setMaxDeficit()`, defaulting to zero.

8 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.