CHAINSECURITY

PUBLIC

Code Assessment
of the Diamond PAU v1.16 Smart Contracts

November 10, 2026

Produced for

Sky



Contents

1	Executive Summary	3
2	Assessment Overview	4
3	System Overview	6
4	Trust Model	9
5	System Considerations	11
6	Terminology	12
7	Findings	13
8	Limitations and use of report	16

1 Executive Summary

Dear Sky Team,

Thank you for trusting ChainSecurity with the review of Diamond PAU v1.16.

As a globally trusted security partner for decentralized finance, ChainSecurity provides security assessments of on-chain liquidity management systems and their integrations.

We greatly appreciate the opportunity to collaborate with Sky on the Diamond PAU framework again.

Sky implements a new facet that swaps assets 1:1 between the ALMProxies of two PAUs in a single transaction. `SwapAssetFacet` is bounded by directional rate limits on both PAUs and protects the initiating PAU with exact balance checks. Additionally, `BeaconConfig` now exposes builders for every integration, and `L2BeaconSpell` lets L2 governance update the integrations of a Beacon. In **Version 2**, the initiate rate limit keys also name the counterparty Controller, `PAUFactory` deploys PAUs with `CREATE2`, and `PAUInit` performs additional checks before governance onboards a new PAU.

We have conducted a thorough review of the scope. We covered:

- Functional correctness of the swap flow across two PAUs, including the decimal conversion and the directional rate limits. No issues were found in the conversion or the rate limit accounting.
- The trust placed between PAUs, including several PAUs swapping with each other and an ALMProxy shared by several Controllers. We found that the balance checks of the initiating PAU can be satisfied with its own incoming funds. Additionally, the counterparty cannot stop accepting swaps by revoking its allocators, only by setting its rate limits to zero.
- The deployment and initialization of new PAUs in **Version 2**. Salts bound to the caller prevent front-running the deployment. The checks of `PAUInit` offer some protection against PAUs prepared maliciously before the handover to governance, but they can be bypassed and do not replace a review before onboarding.
- Correctness of the integration wiring. The refactored `BeaconConfig` produces the same wiring for every existing integration, and the new Swap Asset wiring does not collide with any other selector.

We provide this audit report to assist in forming an opinion on the risk profile of Diamond PAU v1.16. In summary, we find that the codebase provides a high level of security, if operated under the assumptions documented in **System Overview**, **Trust Model** and **System Considerations**.

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 v1.16 repository based on the documentation files. The code versions relevant to this report and when they were received are indicated below.

Version	Date	Commit Hash	Note
1	2 November 2026	2a2d0e3dad4b32601bccdf5547a03b159bdca142	v1.16.0-beta.0
2	10 November 2026	b0b7efd0d56f83b4543381750283fb77c718d3e5	v1.16.0

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

The review covers the changes introduced on top of the previously reviewed `v1.15.0` release. The following files are in scope:

```
deploy/
└── L2BeaconSpell.sol
src/
├── facets/
│   └── swap-asset/
│       ├── ISwapAssetController.sol
│       ├── ISwapAssetFacet.sol
│       └── SwapAssetFacet.sol
└── libraries/
    └── BeaconConfig.sol
```

For `BeaconConfig.sol`, only the split of each `set<FacetName>Integration()` into a `build<FacetName>Integration()` function and a setter, and the new Swap Asset integration, are in scope.

In **Version 2**, the changes to the following files were added to the scope:

```
deploy/
└── PAUInit.sol
script/
└── DeployPAU.s.sol
src/
├── interfaces/
│   ├── IALMProxy.sol
│   ├── IALMProxyFreezable.sol
│   ├── IPAUFactory.sol
│   └── IRateLimits.sol
├── libraries/
│   └── RateLimitHelpers.sol
├── ALMProxy.sol
├── ALMProxyFreezable.sol
├── PAUFactory.sol
└── RateLimits.sol
```

For these files, only the changes made in **Version 2** are in scope.

The interaction of `SwapAssetFacet` with the rest of the PAU system was considered, in particular several PAUs swapping with each other and an ALMProxy that grants the `CONTROLLER` role to more than one Controller.

2.1.1 Excluded from scope

All other contracts in the repository. However, these contracts have been covered in other reviews. The L2 governance relay and the bridge that deliver messages to `L2BeaconSpell` are out of scope and assumed to behave as documented. Additionally, all dependencies, other scripts, tests and similar are out of scope.

3 System Overview

This system overview describes the latest version (**Version 2**) of the contracts as defined in the Assessment Overview. The changes per version are listed in the Changelog.

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

Sky operates several PAUs. Each PAU has an `ALMProxy` that holds the funds, a `Controller` that routes allocator calls to facets through `delegatecall`, a `RateLimits` and an `AccessControls`. A `Beacon` holds the facet wiring, and each `Controller` copies it when its admin calls `updateIntegrations()`. An `ALMProxy` can grant the `CONTROLLER` role to more than one `Controller`.

3.1 Swap Asset Integration

`SwapAssetFacet` swaps two assets 1:1 between the `ALMProxy` contracts of two PAUs in a single transaction. Amounts are converted using `decimals()` of both tokens, and a conversion that would lose precision reverts. Both `Controller` contracts must have the facet wired.

An allocator of the initiating PAU calls `initiateSwap(counterpartyController, assetOut, assetIn, amountOut)`:

1. The facet reads the counterparty `ALMProxy` from `counterpartyController.proxy()` and requires that `counterpartyController` holds `CONTROLLER` on it.
2. The initiating `ALMProxy` approves the counterparty `ALMProxy` for `amountOut` and calls `completeSwap()` on the counterparty `Controller`.
3. The counterparty `ALMProxy` pulls `amountOut` with `transferFrom()` and sends back the converted amount. `completeSwap()` has no role check. It is restricted by the rate limit key, which contains `msg.sender`.
4. The initiating `ALMProxy` resets the approval and requires that exactly `amountOut` left and exactly the converted amount arrived.

3.1.1 Rate Limits

Each PAU decreases the key of the direction it sends and increases the key of the opposite direction, if that key is configured. Keys are kept in the decimals of the asset the PAU sends. `initiateSwap()` uses `LIMIT_SWAP_ASSET_INITIATE` keys and `completeSwap()` uses `LIMIT_SWAP_ASSET_COMPLETE` keys.

Operation	Key tuple	Change
`initiateSwap`	`(assetOut, counterpartyProxy, counterpartyController, assetIn)`	`-amountOut`
`initiateSwap`	`(assetIn, counterpartyProxy, counterpartyController, assetOut)`	`+amountIn` if configured
`completeSwap`	`(assetOut, msg.sender, assetIn)`	`-amountOut`
`completeSwap`	`(assetIn, msg.sender, assetOut)`	`+amountIn` if configured

The increase is capped at `maxAmount`, so a round trip restores a key only as far as the other key had room. The limits bound the net flow in each direction, not the volume.

The initiate keys identify the counterparty by its `ALMProxy` and its `Controller`, so an allocator can only pass a `Controller` that its governance has whitelisted. The complete keys identify the initiating PAU by its `ALMProxy` only, so `completeSwap()` cannot tell which `Controller` of the initiating `ALMProxy` started a swap.

3.2 BeaconConfig

`BeaconConfig` holds the wiring of every facet. Each former `set<FacetName>Integration(beacon, facet)` is split into a pure `build<FacetName>Integration(facet)`, which returns the integration identifier and its configuration, and a setter that passes the result to `Beacon.setIntegration()`. The wiring of all existing integrations is unchanged. `buildSwapAssetIntegration()` wires the five functions of `ISwapAssetController` to `SwapAssetFacet`.

3.3 L2BeaconSpell

`L2BeaconSpell` updates the integrations of a `Beacon` on another chain. The L2 governance relay, which holds `DEFAULT_ADMIN_ROLE` on the `Beacon`, executes it through `delegatecall`. `removeAndSetIntegrations(idsToRemove, integrationsToSet, deadline)` reverts after `deadline`. Otherwise, it removes and then sets integrations, all atomically. Removals run first, so a selector released by a removed integration can be wired into a new one in the same message.

The spell does not update any `Controller`. Each `Controller` keeps its previous wiring until its admin calls `updateIntegrations()` or `removeIntegrations()`. When a selector moves from one integration to another, the old integration must be removed from the `Controller` before the new one is synced.

3.4 PAU Deployment

The deployment of PAUs was reviewed in the Diamond PAU v1.15 report. Compared to v1.15, it changes as follows:

- `PAUFactory` deploys with `CREATE2` instead of `CREATE`. Each `deploy*()` function takes an additional `salt` and uses `keccak256(abi.encode(msg.sender, salt))`, so the addresses depend on the caller and cannot be taken by someone else. The factory records every deployed contract in the `is*` mappings. The new `deployPAU(admin, salt)` deploys an `AccessControls`, an `ALMProxy` and a `RateLimits` with the given admin and a `Controller` wired to them in one call. `script/DeployPAU.s.sol` calls it.
- `ALMProxy`, `ALMProxyFreezable` and `RateLimits` inherit `AccessControlEnumerable` instead of `AccessControl`, so their role members can be counted.
- `PAUInit.init()` previously required a sole admin only on the `AccessControls`. It now also requires it on the `ALMProxy` and the `RateLimits`, and that neither has a `CONTROLLER` before it grants the role to the new `Controller`. As a result, `init()` only initializes stacks with a new `ALMProxy` and `RateLimits`. Stacks that share them with another `Controller` must be initialized without `init()`.

3.5 Changelog

In **Version 2**, the following changes were introduced:

- `SwapAssetFacet`: the initiate keys contain the counterparty `Controller` in addition to its `ALMProxy`. The keys were renamed from `LIMIT_INITIATE_ASSET_SWAP` and `LIMIT_COMPLETE_ASSET_SWAP` to `LIMIT_SWAP_ASSET_INITIATE` and `LIMIT_SWAP_ASSET_COMPLETE`, and the single `SwapAssetSwap` event was split into `SwapAssetInitiateSwap` and `SwapAssetCompleteSwap`.

- `RateLimitHelpers`: adds `makeAddressAddressAddressAddressKey()`.
- `PAUFactory`, `PAUInit`, `ALMProxy`, `ALMProxyFreezable` and `RateLimits`: the changes described in PAU Deployment.

4 Trust Model

This trust model covers only the components in scope: `SwapAssetFacet`, the `BeaconConfig` builders and `L2BeaconSpell`, introduced in **Version 1**, and the changes to `PAUFactory` and `PAUInit` in **Version 2**. The trust model of the underlying PAU framework (`Controller`, `ALMProxy`, `RateLimits`, facet wiring) is described in the full Diamond PAU report.

A swap involves two PAUs. The initiating PAU calls `initiateSwap()`, and the counterparty PAU executes `completeSwap()`. Each PAU has its own governance, and each trusts the other PAU only within the rate limits it configures.

4.1 Roles

Governance (`DEFAULT_ADMIN_ROLE` on the `AccessControls`, `ALMProxy` and `RateLimits` of a PAU): Fully trusted. It wires `SwapAssetFacet`, configures the `LIMIT_SWAP_ASSET_INITIATE` keys that whitelist counterparty `ALMProxy` and `Controller` pairs and the `LIMIT_SWAP_ASSET_COMPLETE` keys that whitelist counterparty `ALMProxy` contracts, per asset pair and direction, and grants the `CONTROLLER` role on its `ALMProxy`. It onboards new PAUs with `PAUInit.init()`. A key for an unintended counterparty or asset pair lets assets leave the `ALMProxy` at 1:1 up to that limit. Granting `CONTROLLER` on its `ALMProxy` to an arbitrary contract lets that contract initiate swaps against every PAU that whitelisted the `ALMProxy` in its complete keys.

Allocator of the initiating PAU (`ALLOCATOR_ROLE`): Partially trusted. It calls `initiateSwap()` and chooses the counterparty `Controller`, the assets and the amount. Every swap is bounded by the initiate limit of its own PAU and by the complete limit of the counterparty. It cannot choose the price, since swaps always settle 1:1 after decimal conversion. In a depeg, it can swap the depegged asset into the counterparty `ALMProxy`, up to the counterparty's complete limit. It can only pass a counterparty `Controller` that holds `CONTROLLER` on its `ALMProxy` and that its own PAU whitelisted in an initiate key. If its PAU whitelisted several `Controller`s of the same counterparty `ALMProxy`, the complete limits of all of them apply. Revoking the counterparty's own allocators does not stop it. The admin of `ALLOCATOR_ROLE` on the initiating PAU can stop it by revoking the allocator. The counterparty can stop it by setting its `LIMIT_SWAP_ASSET_COMPLETE` rate limits to zero. Its governance can always do this. Under the Sky PAS setup, a cBEAM of its `RateLimits` can also do it immediately through the `Configurator`, unless the key is unlimited or the PAS `BeamState` is stopped.

Counterparty Controller: Trusted by the initiating PAU, whose governance whitelists it together with its `ALMProxy` in the initiate keys. It holds `CONTROLLER` on that `ALMProxy` and receives the call from the initiating `ALMProxy`. **Its logic can change when its own PAU wires new facets.** During that call it can spend the `assetOut` approval and move funds of its own `ALMProxy`. The balance checks of `initiateSwap()` require the exact amounts to move, but they do not check where the incoming `assetIn` comes from. If the `Controller` can cause another payment of `assetIn` to the initiating `ALMProxy` during the call, it can take up to `amountOut` per swap, bounded by the initiate limit.

Initiating ALMProxy: Trusted by the counterparty PAU. Any `Controller` that holds `CONTROLLER` on it can make it call `completeSwap()`. The counterparty cannot tell these `Controller`s apart and bounds all of them with the same complete limit.

PAU deployer (caller of `PAUFactory`): Untrusted. Anyone can deploy PAU components and choose their admin, who can prepare them maliciously before handing them over to governance. The checks of `PAUInit.init()` offer some protection, but they are not complete and can be bypassed in several ways, for example by changing role admins with `setRoleAdmin()`, by writing arbitrary `ALMProxy` storage through a temporary `CONTROLLER` and `doDelegateCall()` (e.g., role memberships that `getRoleMemberCount()` does not count), by presetting rate limits, or by using components that were not deployed by the `PAUFactory`. Governance is assumed to review every PAU that it did not deploy with itself as admin before onboarding it.

L2 governance relay: Fully trusted. It holds `DEFAULT_ADMIN_ROLE` on the `Beacon` and executes `L2BeaconSpell` with arguments from L1 governance. A malicious configuration wires arbitrary facet code,

which each `Controller` executes through `delegatecall` with full access to its `ALMProxy` once its admin syncs the integration.

Controller admin (`DEFAULT_ADMIN_ROLE` on the `AccessControls` of a PAU): Fully trusted. It syncs `Beacon` changes into the `Controller` with `updateIntegrations()` and `removeIntegrations()`. It is assumed to verify that the `Beacon` holds the reviewed configuration before syncing.

4.2 External Components

Swapped assets: Assumed to be standard ERC-20 tokens that are pegged 1:1 to each other, return a constant `decimals()`, and neither rebase, charge a transfer fee nor call back into the caller. Tokens that return nothing on `transfer()`, `transferFrom()` and `approve()`, such as USDT, are supported. A depeg is borne by the PAU that receives the depegged asset, up to its rate limits.

Bridge and L2 governance relay: Assumed to deliver L1 governance messages correctly and only once. The spell's `deadline` is compared with the timestamp of the destination chain.

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 Counterparty Cannot Instantly Stop Accepting Swaps

System Consideration **Version 1**

In `SwapAssetFacet`, a swap is started by an allocator of the initiating PAU through `initiateSwap()`. The initiating ALMProxy then calls `completeSwap()` on the counterparty Controller, which only requires a configured `LIMIT_SWAP_ASSET_COMPLETE` rate limit for the calling ALMProxy. The counterparty's allocators take no part in such a swap.

Revoking `ALLOCATOR_ROLE`, the documented emergency response of a PAU, therefore does not stop incoming swaps. To stop accepting swaps, the counterparty must set the rate limits of the affected `LIMIT_SWAP_ASSET_COMPLETE` keys to zero. There is one key per asset pair, direction and initiating ALMProxy. Under the Sky PAS setup, a cBEAM of the counterparty's `RateLimits` can do this immediately with `Configurator.setRateLimit()` and a `maxAmount` and `slope` of zero, since decreases need no waiting period. This does not work in two cases:

- A key set to unlimited cannot be lowered through the `Configurator`. Only governance can change it.
- The `Configurator` cannot be used while the PAS `BeamState` is stopped.

Until the keys are zeroed, incoming swaps settle 1:1. For example, if USDT trades at 0.90 USD, a compromised allocator of the initiating PAU can send 10M USDT and take 10M USDS, up to the counterparty's rate limit.

SC2 Initiating PAU Can Change the Counterparty's Asset Mix at Any Time

System Consideration **Version 1**

In `SwapAssetFacet`, the counterparty's allocators take no part in the swap and cannot approve, delay or refuse it.

The initiating PAU therefore decides when, and how much of, the counterparty's balance is exchanged, up to the counterparty's `LIMIT_SWAP_ASSET_COMPLETE` rate limits and token balances. This changes the counterparty's asset distribution outside its own allocation plans. For example, the counterparty may hold 50M USDS that its allocators plan to deploy. The initiating PAU can swap these USDS into USDT, up to the complete rate limit, right before the counterparty's own transaction. The counterparty's planned operation can then fail for lack of USDS, and the counterparty holds more USDT than it chose to.

Each such swap also raises the counterparty's rate limit for the opposite direction, so the initiating PAU can later swap the assets back at a time of its choosing. The counterparty's complete rate limits are thus the only bound on how far another PAU can move its asset mix.

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

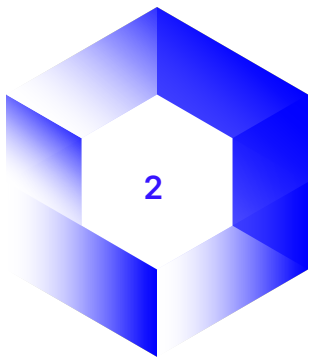
We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

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.

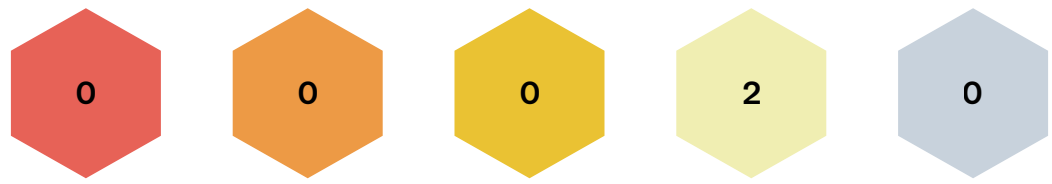
7 Findings

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.



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.

Low	#001	Counterparty Controller Can Pay with the Initiator's Own Funds	Risk Accepted	ⓘ
Low	#002	Third-Party EtherFi Claims Strand ETH in WEETHModule	Risk Accepted	ⓘ

7.2 Descriptions

#001 Counterparty Controller Can Pay with the Initiator's Own Funds

Security | Low | Version 1 | Risk Accepted

In `SwapAssetFacet`, `initiateSwap()` protects the initiating PAU against a malicious counterparty Controller with balance checks. During the call to the counterparty, the ALMProxy balance of `assetOut` must drop by exactly the amount sent, and its balance of `assetIn` must grow by exactly the amount expected. The checks do not verify where the incoming `assetIn` comes from, and they ignore every other asset. A malicious counterparty Controller can therefore make another source pay `assetIn` to the initiating ALMProxy during the call, so that funds the PAU already owns are counted as the swap payment. Two ways to do this follow.

Permissionless incoming transfers: `CCTPFacet` lets anyone relay a CCTP transfer. Suppose PAU A has sent 1M USDC from another chain to its own ALMProxy, and the transfer is waiting to be relayed. An allocator of A swaps 1M USDS for 1M USDC with the counterparty. The counterparty Controller takes the 1M USDS, relays the pending CCTP transfer, and pays nothing. Both balance checks pass, and A has lost 1M USDS.

A second Controller on the initiating ALMProxy: ALMProxy A can serve two Controllers, A1 and A2, and A2 may accept swaps from ALMProxy B. The counterparty can then pay A1 with a second swap on A2, which ALMProxy A itself pays for in another asset:

1. Through A1, an allocator of A swaps 1M USDS for 1M USDC with the malicious Controller M on ALMProxy B. M takes the 1M USDS.
2. Instead of paying, M makes ALMProxy B swap with A2. ALMProxy B gives 1M USDC and takes 1M of a third asset X from ALMProxy A.
3. A1 only checks USDS and USDC. ALMProxy A sent exactly 1M USDS and received exactly 1M USDC, so the swap succeeds.

ALMProxy A	USDS	USDC	X
Change	-1M	+1M	-1M

ALMProxy A has paid twice for the same 1M USDC, so it loses 1M. This requires:

- ALMProxy A grants `CONTROLLER` to A1 and A2.
- A2 has `SwapAssetFacet` wired and a complete swap rate limit with ALMProxy B as counterparty, sending X and receiving USDC.
- ALMProxy B grants `CONTROLLER` to M.
- X differs from USDS and USDC, since the checks of A1 would see it otherwise.

Risk Accepted

In **Version 2**, the initiate rate limit keys contain the counterparty `Controller`, so an allocator can only swap with a `Controller` whitelisted by governance. The described scenarios remain possible with a malicious whitelisted counterparty `Controller`, which Sky considers trusted.

#002 Third-Party EtherFi Claims Strand ETH in WEETHModule

Correctness | Low | Version 1 | Risk Accepted

`WEETHModule` is outside the scope of this review. We report this issue because EtherFi's upgrade changed its behavior.

In `WEETHModule`, the module owns the EtherFi withdraw request NFTs of a PAU. The allocator later calls `claimWithdrawal()` through the `WEETHFacet`. The module then settles the request, wraps the ETH it received into WETH and sends it to the ALMProxy. This relies on only the NFT owner being able to settle a request.

On 2026-07-14, EtherFi upgraded its `WithdrawRequestNFT` so that any account can settle a finalized request, with the ETH paid to the NFT owner. A third party, including EtherFi's own claim keeper, can now settle a request that the module owns. The ETH is paid into the module, which accepts it without accounting for it. The allocator's `claimWithdrawal()` for that request then reverts, because the request no longer exists.

The ETH leaves the module only when the allocator claims another finalized request through it, potentially one owned by another party, since the module does not check ownership. This sweeps the whole balance to the ALMProxy but reports it under the wrong request. For example, a 10 ETH request settled by a third party leaves 10 ETH in the module, and a later 1 ETH claim reports 11 WETH.

> **Risk Accepted**
>
> Sky postpones the fix to a later review, as `WEETHModule` is not used in production and is not expected to be used soon.

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.