

Code Assessment of the Parallelized Allocation System Smart Contracts

October 07, 2026

Produced for



by



Contents

1	Executive Summary	3
2	Assessment Overview	5
3	Limitations and use of report	13
4	Terminology	14
5	Open Findings	15
6	Resolved Findings	16
7	Informational	18
8	Notes	19

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 Parallelized Allocation System according to [Scope](#) to support you in forming an opinion on their security risks.

Sky implements PAS (Parallelized Allocation System), a governance framework for managing rate limited operations through authorized actors called *cBEAMs* (Configurator BEAMs, Bounded External Access Modules that modify the guardrails of external systems). PAS provides timelocked proposal execution, configurable rate limits, and role-based access control for *cBEAMs* to interact with external controllers (e.g. ALM controllers, rate limiters). It is intended to be deployed on both Ethereum mainnet and L2s.

The most critical subjects covered in our audit are functional correctness, access control, and compatibility with the Sky system. Security regarding all the aforementioned subjects is high.

The general subjects covered include extensibility with ALM Controller changes and compliance with Laniakea specifications. Security regarding all the aforementioned subjects is high.

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

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	0
Low -Severity Findings	0



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 Parallelized Allocation System 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	08 Feb 2026	4fab3aacacb0cb04302f57b14ac67cdd0e4cf9ac	Initial Version
2	17 Feb 2026	027ea6d2d210ca9f9b6f26065aeb98b46fbcd6dc	After Intermediate Report
3	04 May 2026	905cc215032e40f5a928d2392b6a542c0ee4408d	More Extensive Initialization
4	04 Aug 2026	947e71cd5dbaaf9c5b3840dd1b23e8e99d9a564d	Wrapper Removal and Unlimited Limit Protection
5	07 Sep 2026	403aebfd55141147432c7fede3d0cfc9e470046	PAS Init Updates
6	07 Oct 2026	5263f26010d6543c7a95b3b2a94533e456af1911	PAS L2 Deployer

For the solidity smart contracts, the compiler version 0.8.24 and evm version cancun was chosen.

The following files are in scope:

```
src/  
  BeamState.sol  
  Configurator.sol  
  PASMom.sol  
  timelock/  
    Bytes32LinkedList.sol  
    Timelock.sol  
    TimelockWrapperForeign.sol  
    TimelockWrapperMainnet.sol  
deploy/  
  PASDeploy.sol  
  PASInit.sol  
  PASInstance.sol  
  L2PASSpell.sol
```

As of **(Version 4)**, `TimelockWrapperForeign.sol` and `TimelockWrapperMainnet.sol` were deleted. Additionally, the following files have been added to scope:

```
deploy/  
  PASAuthorizeInPAU.sol
```

As of **Version 6**, L2PASSpell.sol was deleted. Additionally, the following file has been added to scope:

```
deploy/  
L2PASDeployer.sol
```

2.1.1 Excluded from scope

Any file not explicitly listed above as well as third-party libraries are out of the scope of this review.

2.2 System Overview

This system overview describes the latest received version (**Version 6**) of the contracts as defined in the [Assessment Overview](#).

At the end of this report section we have added a changelog for each of the changes accordingly 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.

Sky implements PAS (Parallelized Allocation System), a governance framework for managing rate limited operations through authorized actors called *cBEAMs* (Configurator BEAMs, Bounded External Access Modules that modify the guardrails of external systems). PAS provides timelocked proposal execution, configurable rate limits, and role-based access control for *cBEAMs* to interact with external controllers (e.g. ALM controllers, rate limiters). It is intended to be deployed on both Ethereum mainnet and L2s.

The system consists of the following core contracts:

- **BeamState**: Central state registry holding authorization rules, role based access control, and the configuration parameters that govern the Configurator.
- **Configurator**: Operational interface through which *cBEAMs* can modify rate limits and execute pre-approved controller actions, subject to ceilings and cooldown periods.
- **Timelock**: An extended OpenZeppelin TimelockController with pausing support, permissionless execution, operation tracking, and disabled self calls.
- **PASMom**: Emergency circuit breaker enabling authorized parties to halt both the BeamState and the Timelock.

Sky also provides deployment and initialization scripts for these components.

2.2.1 BeamState

BeamState is the central configuration and authorization registry for the system. It stores all whitelisted entities (rate limiters, controllers, *cBEAMs*), their pairings, default rate limit configurations, pre-approved controller action hashes, and global parameters governing the Configurator's behavior.

The contract uses two access control mechanisms: a ward-based authorization (*wards*) for administrative operations and a role-based bitmask system (*userRoles* / *actionsRoles*) for operational functions.

Admin Functions (ward-only):

- `rely()` / `deny()`: Grant or revoke ward authorization.
- `setUserRole()`: Assign or revoke a role from a user via bitmask manipulation.
- `setRoleAction()`: Assign or revoke a function selector from a role.

Role-Authed Functions:



These functions require the caller to either be a ward or to have a role whitelisted for the function selector being called.

Operational controls:

- `stop()` / `start()`: Set or clear the emergency stop flag. When stopped, the Configurator will reject all operations.
- `setHop()`: Set the minimum time (in seconds) between rate limit increases for a given rate limiter. Supports a general fallback via `address(0)`.
- `setMaxChange()`: Set the maximum change multiplier (in WAD) for rate limit increases for a given rate limiter. Must be at least 1 WAD (or 0 since **Version 2**). Supports a general fallback via `address(0)`.

Entity management:

- `addRateLimits()` / `delRateLimits()`: Whitelist or delist a rate limiter contract. Deletion is a soft deprecation that prevents new cBEAM pairings.
- `addController()` / `delController()`: Whitelist or delist a controller contract. Deletion is a soft deprecation that prevents new cBEAM pairings.
- `addCBeam()` / `delCBeam()`: Whitelist or delist a cBEAM. Deletion is a soft deprecation that prevents new pairings with rate limiters and controllers.
- `setCBeamForRateLimits()` / `unsetCBeamForRateLimits()`: Pair or unpair a cBEAM with a rate limiter. Pairing requires both entities to be whitelisted.
- `setCBeamForController()` / `unsetCBeamForController()`: Pair or unpair a cBEAM with a controller. Pairing requires both entities to be whitelisted.

Default configurations:

- `addInitRateLimits()` / `delInitRateLimits()`: Set or delete default rate limit configurations (maxAmount, slope) per key and rate limiter. Supports a general fallback via `address(0)`.
- `addInitControllerActions()` / `delInitControllerActions()`: Pre-approve or revoke controller action hashes. The key is derived as `keccak256(data)` where data is the full calldata to be executed on the controller. Supports a general fallback via `address(0)`.

View Functions:

- `hasUserRole()` / `isActionInRole()`: Query role membership.
- `getHop()` / `getMaxChange()`: Query parameters with fallback to the general `address(0)` configuration.
- `getInitRateLimits()`: Query default rate limits with fallback to general configuration.
- `isControllerActionEnabled()`: Check if a controller action hash is pre-approved for a specific controller or generally.

2.2.2 Configurator

Configurator is the operational interface through which cBEAMs interact with rate limiters and controllers. It references BeamState (set as an immutable at deployment) for all authorization checks and parameter lookups. The contract enforces ceilings and cooldown periods on rate limit changes.

cBEAM Functions:

- `setRateLimit()`: Allows an authorized cBEAM to set the rate limit for a given key on a given rate limiter. The function enforces the following constraints:
 - A key is locked as unlimited if either the default configuration in BeamState is unlimited (`maxAmount == type(uint256).max` and `slope == 0`), or the key is currently set to

(`type(uint256).max, 0`) in the rate limiter while no default is registered for it in `BeamState`, neither specific nor general (since [Version 4](#)). In both cases only unlimited values are accepted and `setUnlimitedRateLimitData()` is called, so a *cBEAM* cannot lower such a key.

- Otherwise, the new `maxAmount` must not exceed `current.maxAmount * maxChange / WAD, defaultMaxAmount, or current.maxAmount`. The new `slope` is checked in a similar way.
- Any increase in `maxAmount` or `slope` requires that the hop duration has elapsed since the last increase for this key. The hop must be configured (non-zero).
- After the update, the `lastAmount` is set to the minimum of the new `maxAmount` and the current rate limit value. Note that the initial capacity for a new key will be 0.

To make a key that is locked as unlimited adjustable by a *cBEAM*, a role-authed caller must register a bounded default for it via `addInitRateLimits()`. Note that this default must be non-zero in at least one field, as `(0, 0)` is indistinguishable from an unregistered key. Registering it hands control of the key to the *cBEAM*, which can then immediately lower it down to 0. This is assumed to be a deliberate governance decision, using a default specific to the key and rate limiter rather than the general `address(0)` fallback.

- `callControllerAction()`: Allows an authorized *cBEAM* to execute a pre-approved action on a controller. The full calldata hash must be enabled in `BeamState` for the given controller. The function performs a low-level `call()` on the controller and reverts if the call fails.

Both functions require the system to not be stopped (`notStopped` modifier).

2.2.3 Timelock

Timelock extends OpenZeppelin's `TimelockController` with `Pausable` and introduces the following modifications:

- **Self-calls are disabled:** The contract revokes `DEFAULT_ADMIN_ROLE` from itself in the constructor, preventing proposers from scheduling calls that modify the Timelock's own admin-only configurations (e.g., role management). The `scheduleBatch()` function additionally checks that no target in a batch is the Timelock itself.
- **Permissionless execution:** `EXECUTOR_ROLE` is granted to `address(0)` in the constructor, allowing anyone to execute ready operations.
- **Batch-only operations:** `schedule()` and `execute()` for single operations are overridden to always revert, forcing the use of `scheduleBatch()` and `executeBatch()`.
- **Pausing:** A new `PAUSER_ROLE` is introduced. Holders can call `pause()` to block scheduling, cancellation, and execution. Only the *admin* (`DEFAULT_ADMIN_ROLE`) can call `unpause()`. Note that cancellation is also blocked while paused; if needed, the admin can atomically cancel proposals right after unpausing.
- **Immediate delay update:** `updateDelayImmediately()` allows the *admin* to change the `minDelay` without going through the timelock delay. The inherited `updateDelay()` is called externally to change the `msg.sender`.
- **Operation tracking:** Scheduled operations are stored in a doubly-linked list (`Bytes32LinkedList`) alongside their full parameters (targets, values, payloads, predecessor, salt). Executed and cancelled operations are removed from the list.

Getter Functions (for keeper jobs):

- `getFirstOperationId()` / `getLastOperationId()` / `getOperationsCount()`: List boundaries and count.
- `getPrevOperationId()` / `getNextOperationId()`: Linked list navigation.

- `getOperationExists()`: Check operation membership.
- `getOperation()` / `getOperationLength()` / `getOperationTarget()` / `getOperationValue()` / `getOperationPayload()` / `getOperationPredecessor()` / `getOperationSalt()`: Access individual operation details.

The Timelock calldata is expected to be generated through the [PAS Timelock Calldata Generator](#).

2.2.4 PASMom

PASMom is the emergency circuit breaker for the system. It provides two emergency functions:

- `stop()`: Calls `BeamState.stop()` to activate the emergency stop flag, which blocks all Configurator operations.
- `pause()`: Calls `Timelock.pause()` to pause all scheduling, cancellation, and execution of timelock operations.

The contract supports two authorization models: ownership (owner) and authority delegation. The owner can transfer ownership with `setOwner()` and configure an authority contract with `setAuthority()`. The auth modifier grants access if the caller is the owner or if the authority contract's `canCall()` returns true. The authority is expected to be set to `MCD_ADM`, allowing callers with the Chief's hat to trigger emergency actions.

2.2.5 Deployment Scripts

Deployment and initialization scripts are provided in the `deploy/` directory.

PASInstance

A struct holding the addresses of the three core contracts: `beamState`, `configurator`, and `timelock`.

PASDeploy

Library providing deployment functions:

- `deploy()`: Deploys `BeamState`, `Configurator` (referencing `BeamState`), and `Timelock` (with given `minDelay` and `admin`). Ownership of `BeamState` is transferred from deployer to the specified owner.
- `deployMom()`: Deploys `PASMom` (referencing `BeamState` and `Timelock`) and transfers ownership.

PASInit

Library providing initialization functions. It defines two roles as an enum: `DELAYED` (role 1) and `IMMEDIATE` (role 2).

- `init()`: Core initialization performing sanity checks and configuring both `BeamState` and the `Timelock`:
 - Verifies the `Configurator`'s `BeamState` reference and the `Timelock`'s `minDelay` match expectations.
 - Configures `BeamState` role-action mappings: operations that add or enable entities and configurations (`start`, `setHop`, `setMaxChange`, `addRateLimits`, `addController`, `addCBeam`, `addInitRateLimits`, `addInitControllerActions`) are assigned to the `DELAYED` role. Operations that remove or disable entities and configurations as well as manage `cBEAM` pairings (`stop`, `delRateLimits`, `delController`, `delCBeam`, `setCBeamForRateLimits`, `unsetCBeamForRateLimits`, `setCBeamForController`, `unsetCBeamForController`, `delInitRateLimits`, `delInitControllerActions`) are assigned to the `IMMEDIATE` role.

- The Timelock is assigned the DELAYED role and the *coreCouncil* is assigned the IMMEDIATE role on BeamState.
- The *coreCouncil* is granted the PROPOSER_ROLE and CANCELLER_ROLE on the Timelock. Additional cancellers and pausers are configured.

Note that it is known that delayed and immediate operations order is non-deterministic. The relevant operators of each role are assumed to communicate and track timelock and configurator executions.

- `pauseTimelock()`: Intended to be called after `init()` so that the system starts with the Timelock paused. It grants the PAUSER_ROLE to the given admin, calls `pause()`, and revokes the role again. Until the *admin* unpauses, no proposals can be scheduled or executed, so that DELAYED actions on BeamState are initially limited to what governance configures directly through spells. Note that the admin is assumed not to be part of the pausers array, so that a pre-existing role is not mistakenly revoked.
- `initExtras()`: Configures global hop and maxChange parameters (via `address(0)` fallback), whitelists initial rate limiters and controllers, and adds initial cBEAMs together with their pairings to rate limiters and controllers (potentially independent of the whitelisted ones).
- `initLimitsAndControllerData()`: Sets default rate limit configurations (via `addInitRateLimits`) and pre-approves controller action hashes (via `addInitControllerActions`) in batch.
- `addCoreToChainlog()`: Registers BeamState, Configurator, and Timelock in the Chainlog.
- `initMom()`: Initializes PASMom by performing sanity checks, relying PASMom on BeamState (so it can call `stop()`), granting PASMom the PAUSER_ROLE on the Timelock, and setting its authority to MCD_ADM. Registers PASMom in the Chainlog.
- `unpauseTimelock()`: (added in **Version 5**) serves as a helper to `unpause()` the timelock. Meanwhile, it allows configuring an array of pausers. Note idempotent role grant is allowed and there is no duplication check.
- `relyTimelockInBeacon()`: (added in **Version 5**) grants default admin role of timelock on PAU Beacon contract for integrations management (`setIntegration()` and `removeIntegration()`).

PASAuthorizeInPAU

Library providing an `authorize()` function that is the counterpart to `PASInit`: whereas `PASInit` configures the PAS-owned contracts, this one is expected to be run by a Star's governance (the current `DEFAULT_ADMIN_ROLE` holder of its PAU controller stack) to grant the Configurator the access it needs on the Star's own contracts. It grants `DEFAULT_ADMIN_ROLE` to the Configurator on both the Star's `AccessControls` (gating the facet admin setters dispatched through `callControllerAction()`) and its `RateLimits` (gating `setRateLimit()`).

Note that `DEFAULT_ADMIN_ROLE` is the OpenZeppelin role `admin`, so each grant makes the Configurator a full role administrator of that contract, able to grant and revoke any role on it. This is intended, but implies that the security of these roles rests entirely on the Timelock, *cBEAM* and BeamState gating in front of the Configurator.

L2PASDeployer

A one-time deployer for L2s. In its constructor, it deploys and fully initializes a PAS instance (optionally with a paused Timelock), and then hands over all privileges to an admin (e.g., the L2 governance relay), keeping none itself. PASMom and the Chainlog are not available on L2s and are hence not configured. Granting the Timelock admin rights on the PAU Beacon is expected to be done separately by a governance spell. Note that the deployer does not prevent obviously broken configurations, such as setting the Timelock as the admin. Further, the constructor arguments count towards the `initcode` size limit (EIP-3860), hence a very large configuration may need to be split and completed by the admin afterwards.

2.2.6 Changelog

Since **Version 2**:

1. `setMaxChange()` allows resetting the `maxChange` to 0.
2. Configurator allows decreasing `maxAmount` and `slope` when they are `type(uint256).max`. Note when the new value increases, the computation of `current_value * maxChange` can still overflow though this is not expected to happen in practice.

Since **Version 3**:

1. `PASInit.initExtras()` signature changed: `cBEAMs` are now passed via an `InitCBeamConfig[]` array that bundles each `cBEAM` with its rate limiter and controller pairings, which are set during initialization (previously only the `cBEAMs` were whitelisted, with pairings configured separately).
2. New function `PASInit.initLimitsAndControllerData()` added, allowing spells to batch-register default rate limit configurations and pre-approve controller action hashes.

Since **Version 4**:

1. The Configurator now also locks a key as unlimited when it is currently unlimited in the rate limiter and no default is registered for it in `BeamState`. Previously such a key was treated as a bounded limit, allowing a `cBEAM` to lower a pre-existing unlimited key. Unlocking it requires registering a bounded default that is non-zero in at least one field.
2. `TimelockWrapperMainnet` and `TimelockWrapperForeign` were removed, along with `PASDeploy.deployTimelockWrapperMainnet()`, `PASDeploy.deployTimelockWrapperForeign()` and `PASInit.initTimelockWrapper()`. Proposals are now submitted directly to the Timelock by the authorized proposers.
3. New function `PASInit.pauseTimelock()` added, allowing the deployment spell to have the system start with a paused Timelock. `L2PASSpell.init()` takes a corresponding `startPaused` flag and no longer receives a wrapper address.
4. New library `PASAuthorizeInPAU` added, which is expected to be run by a Star's governance to grant the Configurator `DEFAULT_ADMIN_ROLE` on the Star's `AccessControls` and `RateLimits` contracts.

Since **Version 5**:

Two helper functions `unpauseTimelock()` and `relyTimelockInBeacon()` are added to the `init` library.

Since **Version 6**:

`L2PASSpell` was replaced by `L2PASDeployer`, which deploys, initializes and hands over a `PAS` instance on L2s in a single transaction.

2.3 Trust Model

- *Wards* (`BeamState`): Fully trusted. The wards can reconfigure all role-action mappings, user roles, and all operational parameters. Expected to be governance (e.g., `MCD_PAUSE_PROXY` or L2 governance relay). Incorrect actions can severely impact the system (e.g., assigning improper roles, whitelisting malicious entities).
- *DELAYED role* (`BeamState`): Expected to be the Timelock. This role can add or enable entities and configurations. Operations going through the Timelock are subject to a minimum delay before execution, providing a window for review.

- *IMMEDIATE role* (BeamState): Expected to be the *coreCouncil*. This role can remove or disable entities and configurations as well as manage cBEAM pairings without going through the Timelock. This allows for rapid response to issues (e.g., delisting a compromised cBEAM or controller, or reassigning cBEAM pairings).
- *cBEAMs*: Semi-trusted. cBEAMs (Configurator BEAMs) interact through the Configurator to modify the guardrails of external systems. They can only modify rate limits within the bounds set by the default configurations, the maxChange multiplier, and the hop cooldown. They cannot lower a key that is locked as unlimited, either by registration in BeamState or by already being unlimited in the rate limiter without a registered default. They can only execute controller actions that have been explicitly pre-approved in BeamState. Once cBEAMs are whitelisted, they can be assigned to controllers and rate limiters (potentially without delay), so theoretically they can be positioned to interfere with each other. This is known and assumed to be monitored.
- *Admin* (Timelock, DEFAULT_ADMIN_ROLE): Fully trusted. Can grant and revoke all roles on the Timelock, unpause, and update the minimum delay immediately. Expected to be governance.
- *Proposers* (Timelock, PROPOSER_ROLE): Trusted to submit valid proposals. Expected to be the *coreCouncil*.
- *Cancellers* (Timelock, CANCELLER_ROLE): Can cancel any pending proposal. Expected to include the *coreCouncil* and potentially additional trusted entities.
- *Pausers* (Timelock, PAUSER_ROLE): Can pause the Timelock, blocking all scheduling, cancellation, and execution. The *admin* is required to unpause. Expected to include PASMom and potentially additional trusted entities.
- *Executors*: Untrusted. Execution is permissionless (EXECUTOR_ROLE granted to address(0)), meaning anyone can execute a ready operation.
- *PASMom owner / authority*: Fully trusted. Can halt both the BeamState (via stop()) and the Timelock (via pause()). Expected to be governance with authority set to MCD_ADM.
- *Users / keepers*: Untrusted. Can permissionlessly execute ready operations through the Timelock and use the TimelockHelper to discover executable operations.

Note that since proposals scheduled through the Timelock configure BeamState and the actual downstream changes only take effect when cBEAMs consume these configurations, atomicity in configurations cannot be assumed.

To summarize, the Governance (MCD_PAUSE_PROXY or L2 governance relay) are fully trusted, the Core Council and the cBeams are semi-trusted.

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:

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	0
Low -Severity Findings	0
Informational Findings	2

- [Insufficient Sanity Checks in Deployment Scripts](#) **Acknowledged** **Code Corrected**
- [maxChange Cannot Be Below WAD](#) **Code Corrected**

6.1 Insufficient Sanity Checks in Deployment Scripts

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

CS-SKYPAS-002

- `PASInit.init()` implicitly verifies that the `PauseProxy` has the admin role of `PROPOSER_ROLE` and `CANCELLER_ROLE` on `Timelock` by successfully calling `grantRole()`. However, this is not sufficient. A malicious deployer could deploy the `Timelock` with itself as the default admin, then grant `DEFAULT_ADMIN_ROLE` to the `PauseProxy` while retaining it for itself. As a consequence, the initialization spell will still succeed, but the deployer retains hidden admin privileges. Note that `initMom()` is similarly affected for `PAUSER_ROLE`.
- `PASInit.initExtras()` does not perform a sanity check on the `hop` parameter. A `hop` of 0 set as the fallback would block all rate limit increases for `rateLimits` without a specific `hop` configured.

Since **Version 2**:

1. Acknowledged.
2. `hop` is now checked to be greater than 0.

6.2 maxChange Cannot Be Below WAD

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

CS-SKYPAS-003

In `BeamState`, `setMaxChange()` requires the new value to be at least `WAD`. This leads to the following consequences:

1. It is impossible to reset a specific rateLimits maxChange (including the fallback maxChange). Namely this specific rateLimits cannot use the fallback value and the fallback value cannot be changed back to 0 once set.
 2. Having a maxChange below WAD might make sense, since this requires the maxAmount to decrease and eventually restricts it to be less than defMaxAmount. However, due to the check, this feature cannot be enabled.
-

Code corrected:

setMaxChange () now additionally allows setting the new value to 0.

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 Timelock Can Potentially Change Beacon Admins

Informational **Version 5**

CS-SKYPAS-004

In `PASInit` the timelock is granted `DELAYED` role over the `BeamState`, hence it can only invoke the role-authed functions and has no privilege to change the `BeamState`'s admin wards.

However, there is no such access isolation on PAU Beacon contract. Namely a default admin can not only manage the integration, but also manage the default admin roles.

`relyTimelockInBeacon()` grants timelock the default admin role on the PAU Beacon contract. Consequently, if a malicious proposal passed the time delay and got executed, it can:

1. add malicious default admins for immediate Beacon integrations change.
2. revoke the default admin role from Sky governance (Pause proxy) hence Sky lose control over the Beacon contract.

7.2 Reentrancy in Proposal Execution

Informational **Version 1** **Acknowledged**

CS-SKYPAS-001

In `Timelock`, the proposal execution can reenter since there is no reentrancy guard. Consequently, during the execution of a proposal:

1. It can cancel other proposals if one has `CANCELLER_ROLE`.
2. It can reenter `executeBatch()` to break the atomicity of a proposal. Consequently there could be intermediate states in the `BeamState` when not all configs in the proposal are set.
3. It can schedule new proposals.

In general, reentrancy is not expected in the intended workflow, since the target is intended to be the `BeamState` for setting configurations only. Additionally, there is a check in `scheduleBatch()` that target cannot be the contract itself to prevent self-changing the `minDelay`.

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 Configuration Soft Deprecation

Note **Version 1**

The configurator delegates the access control to BeamState. Namely it checks cBeam's privilege over rateLimits and controllers by querying the mappings controllersCBeams and rateLimitsCBeams of BeamState respectively. Note soft deprecation is implemented by design:

1. If a cBeam is deleted (`delCBeam()`) however its privilege for rateLimits and controllers is not reset, it can still adjust rate limits or execute pre-approved controller actions.
2. If a rateLimits or controller is deleted (`delController()` and `delRateLimits()`) however the cBeam's privilege over it is not deleted, the cBeam can still trigger calls to the now unauthorized rateLimits or controller contract.

Note it is also documented in the comments in BeamState that when deprecating rateLimits, controller, or cBeam, the existing relationships need to be individually unset for full removal.

8.2 Controller Init Action Considerations

Note **Version 1**

By design of the Configurator, rate limit adjustments respect hop and maxChange, while whitelisted controller actions bypass these. The following should be considered when approving init controller actions:

1. Ensure redundant calls to the same init action are idempotent or revert.
2. Configurator does not support `msg.value` in init controller action calls.

8.3 Core Council's Proposals Should Be Carefully Inspected

Note **Version 1**

The Core Council can propose init rateLimits and controller actions, after the delay in the timelock, the proposals can be executed by the cBeam with the configurator. In the current design of the rateLimits and controllers, the configurator should be granted the `DEFAULT_ADMIN_ROLE` to perform the required calls. Consequently, a compromised Core Council may be able to take over the rateLimits and controllers if the malicious proposal is not cancelled during the delay. For instance, it can whitelist a call to drop PauseProxy's `DEFAULT_ADMIN_ROLE` with `revokeRole()`, or to grant `DEFAULT_ADMIN_ROLE` to another account under control with `grantRole()`.

Governance, cancellers, and pausers should carefully inspect the Core Council's proposals to prevent malicious arbitrary calls from being executed.

8.4 Fallback Configs Considerations

Note **Version 1**

BeamState allows using fallback configurations, the following risks should be considered when using a fallback value:

1. `initRateLimits`: The fallback configuration for a key applies for all `rateLimits` contracts. However, different `rateLimits` contracts may be consumed by different controllers (e.g., Spark ALM vs Grove ALM) with different implementations and risk profiles. A ceiling that is appropriate for one controller may be too permissive or restrictive for another.
2. `initControllerActions`: The fallback action enables a specific calldata hash across all controllers. However, different controllers may not share the same implementation for that function. For instance, the calldata may match a different function signature or fall into a controller's fallback function.
3. `maxChange`: This should be set according to the most risky `rateLimits`.

Similarly, when deleting the fallback value with `delInitRateLimits()` or `delInitControllerActions()`, it will be applied across all `rateLimits` and controllers without explicit configurations.

To summarize, when changing the fallback value, the effects on `rateLimits` and controllers without explicit configurations should be carefully considered.

8.5 Inconsistent Configs May Exist in BeamState

Note **Version 1**

Inconsistent configs may exist in BeamState.

1. The `rateLimits`, controller, or the `cBeam` may be deleted from the whitelist, while respective configs may still exist in `rateLimitsCBeams`, `controllersCBeams`, `initRateLimits`, and etc.
2. A fallback configuration may be returned though the controller or `rateLimits` are not whitelisted at all.

Contracts dependent on BeamState should be careful of the inconsistencies and validate all the related configs. Note this is already extensively documented in the codebase.