OpenZeppelin | security

Ava Labs Validator Manager Incremental Audit

Ava Labs.

May 7, 2025

Table of Contents

Summary

Type	
Timeline	From 2025-03-10 To 2025-03-26

Languages

Total Issues	28 (27 resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	2 (2 resolved)
Medium Severity Issues	2 (2 resolved)
Low Severity Issues	13 (12 resolved)
Notes & Additional Information	11 (11 resolved)
Client Reported Issues	0 (0 resolved)

Scope

We performed a diff audit of the ava-labs/icm-contracts repository at commit 8a8f78d against commit 93920df

In scope were the following files:

```
contracts
└── validator-manager
    ├── ACP99Manager.sol
    ├── ERC20TokenStakingManager.sol
    ├── ExampleRewardCalculator.sol
    ├── NativeTokenStakingManager.sol
    ├── StakingManager.sol
    ├── ValidatorManager.sol
    ├── ValidatorMessages.sol
    └── interfaces
        ├── IERC20Mintable.sol
        ├── IERC20TokenStakingManager.sol
        ├── INativeTokenStakingManager.sol
        ├── IRewardCalculator.sol
        └── IStakingManager.sol
```

System Overview

This audit focuses on the updated version of the Validator Manager contracts that manage the lifecycle of subnet (now called L1) validators and delegators, as outlined in the ACP-77 specification. This system enables a transition from a Proof-of-Authority (PoA) model to a decentralized Proof-of-Stake (PoS) validator set for Avalanche L1 chains. The system uses Avalanche Interchain Messaging (ICM) to coordinate state and validator activity between the L1 and the P-Chain, which serves as the canonical source of truth for L1 validator registrations and terminations.

Since our previous audit of the Validator Manager contracts, the contract architecture has been refactored. Previously, the logic for validator management and staking was contained in a single contract. In this iteration, the Validator Manager has been split into two separately deployed contracts in the PoS configuration: the `ValidatorManager` contract and either the `ERC20TokenStakingManager` or `NativeTokenStakingManager` contract, depending on the staking token used.

This change was primarily motivated by the need to reduce contract size to stay within the 24kB limit and to better align the system with the ACP-99 modular contract design pattern. While the external interface and P-Chain messaging behavior remain unchanged, internal calls have been replaced by inter-contract communication. This audit focused on ensuring that these cross-contract interactions are safe and consistent with the protocol's intended behavior.

In addition, the updated system includes functionality to migrate from the original (v1) contracts to the new architecture for L1s governed through the PoA model. The validator registration and delegation flows continue to enforce restrictions around registration expiry, validator weight churn limits, reward accrual and disbursement, and locking/unlocking of stake, just as in the previous design.

Security Model and Trust Assumptions

This audit assumes that the following components and conditions function correctly:

- ICM messages are delivered securely and reliably, with off-chain relayers properly incentivized to forward messages between the P-Chain and the L1.
- Nodes are willing to produce signatures over any message that was passed to the ICM Precompile's `sendWarpMessage` method in an accepted transaction and will not censor any valid transactions.
- Validators only sign valid ICM messages, ensuring that messages reflect accurate and authorized validator activity.
- Validators possess the private keys corresponding to their registered BLS public keys and ownership is verified off-chain during `RegisterSubnetValidatorTx` submissions on the P-Chain.
- The uptime mechanism operates as intended, with accurate data delivered and verified across chains.
- After transitioning from PoA to PoS, the system allows anyone to disable PoA validators, enforcing decentralization and reducing reliance on trusted actors.
- The `uptimeBlockchainID` initialized in the `StakingManager` is correct and must be validated by the L1 validator set that the contract manages.
- The P-Chain enforces a nonce-based replay protection mechanism and will only accept `L1ValidatorWeightMessage` messages that contain a nonce equal to or greater than the current `minNonce` tracked per validator. This prevents replays and ensures that validator weight updates occur in a consistent and monotonic fashion.

Privileged Roles

The system includes multiple privileged roles that hold significant authority. These roles are foundational to the secure operation of the system, and any compromise or misbehavior by privileged actors could lead to security or liveness failures.

- The proxy owner of both the `ValidatorManager` and the associated `StakingManager` contracts (`ERC20TokenStakingManager` or `NativeTokenStakingManager`) retains upgradeability control. These proxy owners are trusted entities whose actions can directly impact the security and behavior of the validator and staking logic.

- In PoA mode, the owner also assumes the role of the ICM message relayer, tasked with submitting messages from the P-Chain on the `ValidatorManager` contract.
- The `ValidatorManager` contract may reside on a chain that is not validated by the L1 that the contract is managing. However, it is essential that the staking and reward tokens live on the same chain as the contract to ensure consistent and secure staking operations.

High Severity

H-01 `ValidatorManager` and P-Chain Can Go Out of Sync

In the `initiateValidatorRegistration` function of the `ValidatorManager` contract, `validationId` is created using the SHA-256 hash of the packed `registerL1ValidatorMessage` message to be sent to the P-Chain. Since ICM messages are directly linked to `validationId`, it is expected that `validationId` is always different. However, a malicious validator in a particular timeframe can generate the same `validationId` again by reusing the `ValidationPeriod` parameters which, in turn, allows for reusing the ICM messages later for their own benefit.

An attacker can successfully register through `initiateValidatorRegistration` with a certain expiry, resulting in a validator with `validationId` `A`. Next, the P-chain confirms the registration and sends `L1ValidatorRegistrationMessage` (`M1`) which can be used in `completeValidatorRegistration` to complete the registration on the validation manager contract side. Then, the malicious validator calls `initiateValidatorRemoval`, once again the P-chain confirms this and sends a `L1ValidatorRegistrationMessage` (`M2`), which can be used in `completeValidatorRemoval`. Basically, the validator went through the whole lifecycle of registration and removal.

Now, before the expiry elapses, the attacker calls `initiateValidatorRegistration` again with the same parameters, resulting in the same `validationId` `A`. The P-chain will not confirm this message, as a certain `validationId` can only be used once from the P-chain's point of view. However, the attacker can reuse the `L1ValidatorRegistrationMessage` (`M1`) of the previous validator lifecycle to successfully complete the registration of the validator. Similarly, the attacker can reuse `L1ValidatorRegistrationMessage` (`M2`) to complete its removal in line with their own benefits, without requiring confirmation from the P-chain.

This results in the `ValidatorManager` contract having a different view of the validator set compared to that of the P-chain. Additionally, it introduces complexities such as resetting `sentNonce` and recreating `delegationId` using `sentNonce`.

It is worth noting that this attack is only feasible if the configured minimum stake duration is less than the maximum registration expiry window as the validator needs to go through the whole validation lifecycle before the registration expiry window is reached.

Consider adding checks to ensure that a `validationId` cannot be reused during `initiateValidatorRegistration`.

Update: *Resolved in pull request #771. The team stated:*

> *The linked PR addresses this vulnerability by removing the registration expiry as a parameter to `initiateValidatorRegistration`, and instead always using the maximum expiry, calculated from the block's timestamp. It also adds a redundant defensive check that the calculated validation ID is unique. An earlier version of ACP-77 required the validator's signature over the expiry (along with some other data), which required the expiry to be determined ahead of the call to `initiateValidatorRegistration`. That requirement was removed prior to ACP-77 being finalized, so we can instead calculate the expiry internally.*

H-02 Validators Can Steal Delegator Rewards

A validator has the ability to initiate the removal of delegators via the `initiateDelegatorRemoval` or `forceInitiateDelegatorRemoval` functions. This capability allows validators to remove delegators associated with their own `validationID` as this may be necessary in scenarios where a validator needs to reduce stake weight to comply with churn constraints. However, it also presents an opportunity for abuse.

Specifically, both removal functions have a version that accepts a `rewardRecipient` parameter, determining the address that will receive the delegation rewards upon completion of the removal via the `completeDelegatorRemoval` function. Since the validator owner is authorized to initiate the removal, they can specify any address - potentially their own - as the reward recipient, thereby diverting rewards away from the original delegator.

This behavior enables validators to unilaterally remove delegators and redirect their rewards to an arbitrary address. This may result in unexpected or unfair loss of rewards for delegators and can undermine trust in the staking mechanism, especially if this behavior is not transparently communicated or constrained by the protocol.

Consider restricting the ability of validators to set or override the `rewardRecipient` when removing a delegator. One possible approach could be to enforce a condition that only a delegator can define or update their own `rewardRecipient`, defaulting to the delegator's owner address if none is provided.

Update: *Resolved in [pull request #3](). The team stated:*

> *The linked PR in the internal fork of `icm-contracts` addresses the immediate issue of validators being able to set or override the delegator's reward recipient, and simplifies the interface for specifying reward recipients for both validators and delegators. With this change, the reward recipient may only be set when initiating registration, or by calling the function that specifically updates the reward recipient. Setting the reward recipient when removing a validator or delegator is no longer supported. On removal, the current reward recipient will be used. For validators, these functions are: - `initiateValidatorRegistration` - the reward recipient is provided as a required argument. - `changeValidatorRewardRecipient` - update the reward recipient. Callable only by the validator owner - `initiateValidatorRemoval` / `forceInitiateValidatorRemoval` - the overloads that accept `rewardRecipient` as an argument are removed For delegators, these functions are: - `initiateDelegatorRegistration` - the reward recipient is provided as a required argument. - `changeDelegatorRewardRecipient` - update the reward recipient. Callable only by the delegator owner - `initiateDelegatorRemoval` / `forceInitiateDelegatorRemoval` - the overloads that accept `rewardRecipient` as an argument are removed*

Medium Severity

M-01 Churn Tracker May Be Ineffective

In the `ValidatorManager` contract, the [`_initiateValidatorRegistration` function]() increases the total weight tracked in the churn tracker via [`_checkAndUpdateChurnTracker`](). Similarly, the [`_initiateValidatorRemoval` function]() reduces the total weight as part of the validator lifecycle. This ensures accurate tracking of active validator weight relative to the P-Chain.

However, in certain specific cases, the validator's registration may be rejected on the P-Chain. For example, the `L1ValidatorRegistrationMessage` may not be signed by a sufficient amount of validator weight. As the validator is not successfully registered, it will not be able to use the [`initiateValidatorRemoval` function]() to exit, as this requires the validator to be active. Instead, the validator can unlock their funds using the `completeValidatorRemoval` function, requiring a signed `L1ValidatorRegistrationMessage` from the P-Chain with the `registered` boolean set to `false`. However, this `completeValidatorRemoval` function does not update the total weight tracked by the churn tracker.

As a result, invalidated validators retain their weight in the churn tracker indefinitely. Over time, this could lead to a desynchronization of the total weight tracked by the P-Chain and the `ValidatorManager` contract. In case the total weight becomes significantly inflated compared to the actual total weight on the P-Chain, it may cause the churn percentage check to pass for new validators who exceed the maximum churn threshold of the true P-Chain total weight, unintentionally allowing oversized validators to register.

In addition, this behavior introduces a stalling vector. As the minimum stake duration is only enforced while exiting an active validator, an attacker could repeatedly register validators that are intentionally invalidated on the P-Chain. Afterwards, they could call `completeValidatorRemoval` to unlock their stake and immediately register again within the same churn window. This allows them to rapidly inflate the churn tracker and block legitimate validators from registering due to the artificial weight cap enforced by the churn tracker, effectively stalling the `ValidatorManager` contract.

Consider updating the churn tracker to properly track the total weight corresponding to the total weight registered on the P-Chain. Additionally, consider implementing restrictions that prevent the abuse of the churn tracker which could lead to stalling the `ValidatorManager` contract.

Update: *Resolved in pull request #776 and pull request #784. The team stated:*

> *The linked PR updates `ValidatorManager` so that an invalidated validator's weight is correctly subtracted from the L1's total weight as tracked by the contract. The stalling vector is already mitigated against due to the fact that the 24 hour expiry must elapse before the P-Chain is willing to sign a negative L1ValidatorRegistration message, which is required to release funds staked in the call to initiateValidatorRegistration. This imposes a 24 hour bonding period for the stake, discouraging the type of rapid churn inflation described above. This could be further mitigated against by requiring that the minimum staking duration also elapse before an invalidated validator's funds can be released. This, however, may be overly punative for honest validators that do not register on the P-Chain in time, and the 24 hour bond may be sufficient on its own to discourage this kind of abuse. As such, we elected to keep the implementation as-is.*

M-02 PoA to PoS Transition May Be Slow

During transitioning from Proof-of-Authority (PoA) to Proof-of-Stake (PoS) mechanism, an ERC20TokenStakingManager or NativeTokenStakingManager contract is deployed with a `weightToValueFactor` and `minimumStakeAmount` alongside other parameters and afterwards set as `owner` of the `ValidatorManager` contract. This transition allows functionalities such as validator registration, removal of PoA validators and delegator registration to be exposed in a permission-less manner. Both the `weightToValueFactor` and `minimumStakeAmount` parameters should be carefully chosen depending upon the decimals and value of the staking token and plays a crucial role in determining the amount of weight assigned to a specific amount of staked assets.

During PoA mode, the owner assigns arbitrary weights to new validators through the `initialValidatorRegistration` function. For example, some PoA-L1s are configured with validator weights of 100 (i.e. AIBTRUSTMAINNET, LAMINAL1). This total weight is tracked in the churn tracking mechanism and is preserved during the transition from PoA to PoS. This protects against direct removal of all PoA validators by the the first (malicious) PoS validator, effectively taking control over the L1 and blocking other PoS from joining as described here. However, this churn tracking mechanism could also prevent PoS validators from registering depending on the configured `weightToValueFactor` and `minimumStakeAmount`, as it only allows to stake a percent of the current total weight.

This may lead to unexpected scenarios such as extremely slow transitions from PoA to PoS, only allowing PoS validators to register with a small amount of staked assets. Moreover, `weightToValueFactor` and `minimumStakeAmount` may have to be chosen to accommodate the low PoA total weight, which could result in precision loss converting the staked assets to weight or overflow as the weight is limited to `uint64`.

Consider adding functionality to scale up the weights of PoA validators before transitioning to PoS to ensure the total weight is compatible with the weights added during PoS. Alternatively, consider documenting the dependency between `weightToValueFactor`, `minimumStakeAmount` and the current total weight for L1s considering transitioning from PoA to PoS.

Update: *Resolved in pull request #764 and pull request #783. The team stated:*

> *The linked PR adds a guide for migrating from PoA to PoS that discusses that various considerations when selecting weights for both PoA and PoS validators. Ultimately, once validators are registered with a given weight, churn limits must be respected regardless of whether the validator manager contract manages PoA or PoS validators.*

Low Severity

L-01 Validators May Not Be Able to Exit Due to Churn Limits

The `_initiateValidatorRemoval` function of the `ValidatorManager` contract updates the entire weight of the validator to 0. In the specific scenario where a validator's weight is higher than the maximum churn percentage of the L1 total validator weight, the transaction would revert due to churn limits being exceeded as per this check.

A validator with a weight higher than churn limits, trying to exit and unstake using `initiateValidatorRemoval` function in the `StakingManager` contract, will not succeed. This validator may have to wait until new validators or delegators join the network such that its own weight is below the churn limits. Only after this may the validator be able to call `initiateValidatorRemoval` again and subsequently call `completeValidatorRemoval` to withdraw its stake.

Furthermore, if the L1 only contains a small amount of validators and no new validators are joining the network, there could be a scenario where multiple validators with a weight higher than the churn limits may not be able to exit at all and their funds would be stuck in the `StakingManager` contract. Especially, if the L1 `ValidatorManager` and `StakingManager` contracts are residing on the C-Chain rather than on a chain that the L1 validators are validating. In this specific case, despite the validators having control of the chain validated by the L1, their staking funds may get stuck in the `StakingManager` contract that is on the C-Chain. Moreover, there will be always some dust weight present in the `StakingManager` contract due to this check, belonging to either the validators or the delegators.

This pitfall can be considered a design choice around the churn tracking mechanism. Thus, if redesigning is not possible, consider properly documenting this case so that validators are aware of the risk.

Update: *Resolved in pull request #780. The team stated:*

> *These restrictions on exiting a PoS L1's validator set are an intentional design choice. PoS L1s are expected to have sufficiently large and decentralized validator sets such that these edge cases are exceedingly uncommon. In order for a large validator to be able to exit without relying on other validators/delegators to join, we would need to*

> *implement a way for the validator to reduce its weight incrementally. The complexity that would add (especially with respect to rewards calculation and distribution) is not worth supporting this edge case. Rather, we document these edge cases and continue to promote large, decentralized PoS validator sets.*

L-02 Delegators May Not Earn Delegation Rewards

In the `StakingManager` contract, the `MAXIMUM_DELEGATION_FEE_BIPS` constant is responsible for ensuring that delegation fees are not more than 10_000 bps. However, a validator can set their delegation fees at 10_000 which is equal to the BIPS_CONVERSION_FACTOR (equivalent to setting 100% delegation fees), resulting in delegators not earning any delegation rewards for their staked amount.

Given the lack of visibility of important information in the `PoSValidatorInfo` and `Delegator` structs, there is a high chance that delegators may not be aware of the delegation fees imposed by a certain validator.

Consider setting `MAXIMUM_DELEGATION_FEE_BIPS` to a lower value to help ensure that delegators are able to earn meaningful rewards when choosing to delegate their funds to a validator.

Update: *Resolved in pull request #777. The team stated:*

> *Rather than imposing an upper limit on the delegation fee, the linked PR exposes getters for retreiving this information so that prospective delegators can choose validators with acceptable fees. Specifically, it adds: - `getStakingManagerSettings` to expose the contract's minimum delegation fee - `getStakingValidator` to expose a validator's delegation fee*

L-03 Maximum Registration Expiry Not Configurable

The `MAXIMUM_REGISTRATION_EXPIRY_LENGTH` constant in the `ValidatorManager` contract is currently hardcoded and cannot be updated without modifying the `ValidatorManager` implementation. The ACP-77 specification notes that this limit may need to be reduced dynamically by the P-Chain to mitigate potential memory pressure in cases where `RegisterL1ValidatorMessage` messages are spammed.

The inability to update the maximum expiry value without redeploying or upgrading the `ValidatorManager` limits the protocol's flexibility when it comes to responding to real-time constraints on the P-Chain. If the P-Chain enforces a stricter limit (e.g., lower than 24 hours), the smart contract may continue producing registration messages that are invalid under current network conditions, leading to unnecessary message rejections, wasted gas, and degraded system efficiency.

Consider refactoring the `ValidatorManager` contract to make `MAXIMUM_REGISTRATION_EXPIRY_LENGTH` configurable via an admin-controlled or governance-approved mechanism. This will allow dynamic adjustments to the expiry window to align with evolving requirements or constraints on the P-Chain without requiring a full contract upgrade.

Update: *Acknowledged, not resolved. The team stated:*

> *ACP-77 notes "A future ACP can reduce the window of expiry if 24 hours proves to be a problem." This would require a network upgrade to activate this ACP, which may contain other backwards incompatible changes that would need to be addressed in the validator manager contracts. Instead, we elect to document the contracts as compatible with ACP-77 and defer handling any potential future changes to later. This documentation is already present, so no changes are needed.*

The team did opt to leave the window of expiry as constant and, if necessary, address the changes in a future version of the validator manager contract.

L-04 `claimDelegationFees` Does Not Take the Reward Recipient into Account

The `claimDelegationFees` function does not take into account the reward recipient that may have been configured through the `changeValidatorRewardRecipient` function. Instead, it directly transfers the rewards to the owner of the validator.

This behavior may lead to confusion for validators who expect rewards to be sent to the designated rewards recipient, as they will instead be transferred to the owner. It could also introduce unexpected integration challenges for protocols that assume that the rewards recipient will receive all associated fees.

If the validator has configured a reward recipient, consider transferring the delegation fees earned by this validator to the reward recipient, similar to the `completeValidatorRemoval` function.

Update: *Resolved in [pull request #772](). The team stated:*

> *The linked PR updates `claimDelegationFees` to withdraw rewards to the validator reward recipient set via `changeValidatorRewardRecipient`. If the reward recipient is unset, it falls back to the validation owner. It also removes the deletion of the reward recipient in `completeValidatorRemoval`, since it is now used in `claimDelegationFees`.*

L-05 Missing Check for Duplicate Addresses in `_validatePChainOwner`

The [`_validatePChainOwner` function]() ensures that `PChainOwner` meets the validation criteria outlined in the [ACP-77 specification](). It correctly checks that if the threshold is 0, the address list is empty, and that the threshold does not exceed the number of addresses. It also verifies that addresses are sorted in ascending order. However, it does not explicitly check for duplicate addresses, which is required by the ACP-77 specification.

If duplicate addresses are allowed, it could lead to unintended behavior when validating ownership thresholds, potentially enabling improper authorization or quorum calculations. This could pose security risks, as the system might incorrectly count duplicate addresses towards the threshold, leading to unintended validator permissions or incorrect staking assignments.

Consider adding an explicit check to ensure that all addresses of the `PChainOwner` struct are unique.

Update: *Resolved in [pull request #765](). The team stated:*

> *The linked PR adds a validation check that the list of addresses in the `PChainOwner`s passed to the validator manager contracts do not contain duplicate addresses.*

L-06 `_validatePChainOwner` Does Not Check for Zero Addresses

The [`_validatePChainOwner` function]() performs several checks to ensure that a `PChainOwner` struct complies with the [ACP-77 specification](). It validates that the threshold is consistent with the number of addresses and confirms that the address list is sorted. However, it does not check whether any of the addresses in the list are zero addresses.

Allowing zero addresses in the `PChainOwner`'s list could result in unusable ownership configurations. Specifically, if an address is zero, the corresponding owner may be unable to disable their validator on the P-Chain using a `DisableL1ValidatorTx`, or may be unable to collect their AVAX balance after disabling the validator. This could lead to locked funds or validators that cannot be properly managed, resulting in user-facing issues and operational inefficiencies.

Add a validation step in `_validatePChainOwner` to ensure that none of the addresses in the addresses array are zero addresses. Doing this would help ensure that all owners are capable of performing required operations on the P-Chain, preserving expected functionality.

Update: *Resolved in pull request #763. The team stated:*

> *The linked PR adds a validation check that prevents zero addresses from being used in the `PChainOwner` address list.*

L-07 Maximum Registration Expiry Exceeds Limit Defined in ACP-77

The `ValidatorManager` contract currently allows a `RegisterL1ValidatorMessage` to be created with a `registrationExpiry` of up to 48 hours in the future. However, according to the ACP-77 specification, the expiry must not exceed 24 hours from the time the transaction is issued on the P-Chain. Any message with an expiry beyond this window is considered invalid and will be discarded by the P-Chain.

By allowing a registration expiry of up to 48 hours, the contract permits validators to register with parameters that will be automatically rejected by the P-Chain after 24 hours. This discrepancy can lead to wasted ICM messages, increased gas costs, and validator registrations that silently fail without clear feedback, reducing the reliability and predictability of the registration process.

Consider updating the `ValidatorManager` contract's `MAXIMUM_REGISTRATION_EXPIRY_LENGTH` constant to enforce a 24-hour maximum, aligning it with the ACP-77 specification. This change will ensure that all `RegisterL1ValidatorMessage` messages conform to protocol expectations and are not prematurely discarded by the P-Chain.

Update: *Resolved in pull request #771. The team stated:*

> *The linked PR updates the `MAXIMUM_REGISTRATION_EXPIRY_LENGTH` constant to match ACP-77.*

L-08 Lack of Event Emissions

Throughout the codebase, multiple instances of missing event emissions were identified:

1. Inside the `claimDelegationFees` function of the `StakingManager` contract.
2. Inside the `changeValidatorRewardRecipient` function of the `StakingManager` contract.
3. Inside the `changeDelegatorRewardRecipient` function of the `StakingManager` contract.
4. Inside the `_withdrawDelegationRewards` function of the `StakingManager` contract.
5. Inside the `_withdrawValidationRewards` function of the `StakingManager` contract.

The aforementioned functions are responsible for distributing rewards/fees and updating important storage slots. Therefore, consider emitting events whenever state changes are performed in these functions for improved transparency and better monitoring capabilities.

Update: *Resolved in pull request #766. The team stated:*

> *The linked PR adds events to track delegation and validation reward recipient changes, and delegation and validation reward withdrawals.*

L-09 Duplicate Code

Within the `StakingManager` contract, one instance of duplicate code was identified. This check to ensure that the ICM message to update the uptime of a specific validator has originated from a validator node has been implemented twice.

Consider removing any instances of duplicate code to improve the clarity and maintainability of the codebase.

Update: *Resolved in pull request #762. The team stated:*

> *The linked PR removes the duplicate code.*

L-10 Lack of Input Validation

Throughout the codebase, multiple instances of missing input validation were identified:

- The `churnPeriodSeconds` variable initialized inside the `__ValidatorManager_init` function of `ValidatorManager` contract lacks input validation allowing the deployer to set the variable as high as `max(uint64)`.
- Both the `_manager` and `_rewardCalculator` addresses initialized inside the `__StakingManager_init_unchained` function of the `StakingManager` contract are missing checks against `address(0)`.

Consider validating all constructor inputs to ensure that the contracts cannot be deployed with invalid arguments.

Update: *Resolved in pull request #762. The team stated:*

> *The linked PR adds validation checks to ensure the zero address is not passed to the `StakingManager` and `ValidatorManager` initialization functions. Also adds a check that only active PoA validators can update their weight. It does not add the suggested validation check to `churnPeriodSeconds`, as there is no meaningful upper bound on the churn period duration. Instead, we leave it to the deployer to set this value appropriately.*

L-11 Wrong Usage of `reinitializer` Modifier

Both `NativeTokenStakingManager` and `ERC20TokenStakingManager` contracts have an `initialize` function with the `reinitializer(2)` modifier. However, these are the first versions of the contracts and no previous version has been deployed yet. Therefore, these `initialize` functions should use the `initializer` modifier instead.

Consider changing the `reinitializer(2)` modifier to `initializer` on the `initialize` functions of both `NativeTokenStakingManager` and `ERC20TokenStakingManager`.

Update: *Resolved in pull request #767. The team stated:*

> *The linked PR updates the `NativeTokenStakingManager` and `ERC20TokenStakingManager` initializer functions to use the correct modifier, as suggested. The use of `reinitializer(2)` was a vertige of the V1 architecture that related the `ValidatorManager` and `PoSValidatorManager` via inheritance in a*

> *single contract. To upgrade from PoA to PoS, the contract needed to be allowed to be initialized again after being upgraded.*

L-12 Inefficient Order of Validation in `initializeValidatorSet`

The `initializeValidatorSet` function verifies the SHA-256 hash of the L1 conversion data (`conversionID`) only after performing multiple iterations and storage operations. If the hash verification fails, all previous computations and state modifications become unnecessary, leading to wasted gas.

The function should prioritize this validation at the beginning to enforce a fail-fast approach. Delaying the validation of `conversionID` increases gas consumption when processing invalid data. This inefficiency leads to higher transaction costs for users and unnecessary state modifications.

To optimize for gas efficiency and follow best practices, the `conversionID` verification should be moved to the beginning of the function, immediately after checking the `_initializedValidatorSet` flag. This early validation ensures that the function exits before performing unnecessary operations in case of invalid input.

Update: *Resolved in pull request #769. The team stated:*

> *The linked PR reorders the `conversionID` check to occur after the relatively cheap checks against the address and blockchain ID checks, but before the relatively expensive iteration over the validator list. We elected to place this check after the address and blockchain ID checks, rather than at the top of the function as suggested, so that those cheaper checks don't incur the cost of the more expensive `conversionID` check if those parameters are invalid.*

L-13 Inconsistent Imports

Throughout the codebase, multiple instances of inconsistent or redundant imports were identified:

- The `ValidatorManager` contract imports from itself.
- The contracts rely on version `1.2.0` of the `subnet-evm-contracts` library used, while the newest version is `1.2.2`.

- The `PChainOwner` struct is imported from the `IERC20TokenStakingManager` and `INativeTokenStakingManager` interfaces rather than from the `ACP99Manager`.

Consider addressing the above-mentioned instances of inconsistent and/or redundant imports in order to improve the clarity and maintainability of the codebase.

***Update:** Resolved in pull request #768. The team stated:*

> *The linked PR updates the imports for consistency, as suggested.*

Notes & Additional Information

N-01 Incorrect SPDX License Identifier

The contract files in scope currently use `SPDX-License-Identifier: Ecosystem` to indicate that the files are under the custom Ecosystem license. However, this is not a recognized SPDX license identifier. According to the SPDX specification, custom or non-listed licenses should be prefixed with `LicenseRef-` to comply with the standard.

Using a non-standard SPDX license identifier without the `LicenseRef-` prefix may lead to issues with tooling that relies on SPDX compliance, such as license checkers, open-source scanners, or continuous integration systems. This could result in inaccurate license reporting or failure to detect the license entirely.

To align with SPDX standards and ensure compatibility with automated tooling, consider updating the license identifier to `SPDX-License-Identifier: LicenseRef-Ecosystem` or to using an SPDX-approved license if applicable. This will help maintain clarity and improve ecosystem-wide interoperability.

***Update:** Resolved in pull request #773. The team stated:*

> *The linked PR updates the license descriptions for all validator manager contracts to `SPDX-License-Identifier: LicenseRef-Ecosystem` as suggested. A separate PR was also merged to apply this change to all other contracts in the repository.*

N-02 Misleading Documentation

Throughout the codebase, multiple instances of misleading documentation were identified:

- According to the inline documentation in the `initializeValidatorSet` function, the `validationID` is calculated by taking the sha256 hash of the `ConvertSubnetToL1Tx` transaction ID and the index of the initial validator. However, it is actually calculated using the sha256 hash of the `subnetID` transaction ID and the index of the initial validator.
- The `IStakingManager` interface refers to a `completeEndValidation` function. However, this function does not exist.

Consider correcting the aforementioned instances to improve the overall clarity and readability of the codebase.

Update: *Resolved in pull request #775. The team stated:*

> *The linked PR corrects the misleading comments as suggested.*

N-03 Missing Getter Functions

The `StakingManager` contract contains state variables and mappings, leveraging the EIP-7201 namespaced storage layout. While this data is accessible by off-chain components, accessing this data without getter functions is cumbersome if the contract relies on EIP-7201. Especially, the state variables and mappings related to tracking the PoS validator information and delegator information might benefit from getter functions as these will likely be used by the frontend to show the status of certain validators and delegators.

Consider adding getter functions for key state variables and mappings to facilitate easier access by off-chain components. If contract size limits are a concern, an alternative approach would be to deploy a separate helper contract that provides these getter functions, including a universal `getValueAtSlot` function for accessing raw storage slots of the `StakingManager` contract.

Update: *Resolved in pull request #777. The team stated:*

> *The linked PR adds getters to `StakingManager` and `ValidatorManager` for all configuration parameters and active validator and delegator information. Specifically, it adds: - `StakingManager.getStakingManagerSettings` - `StakingManager.getStakingValidator` - `StakingManager.getValidatorRewardInfo` -*

> *`StakingManager.getDelegatorInfo` - `StakingManager.getDelegatorRewardInfo` - `ValidatorManager.getChurnTracker` - `ValidatorManager.getNodeValidationID` - `ValidatorManager.isValidatorSetInitialized`*

N-04 Code Quality and Readability Suggestions

Throughout the codebase, multiple opportunities for improving code quality and readability were identified:

- Rename the `resendEndValidatorMessage` function to be consistent with `initiateValidatorRemoval` and `completeValidatorRemoval`.
- Consider placing the `completeValidatorWeightUpdate` function after the `initiateValidatorWeightUpdate` function to be consistent with the ordering of the other "initiate" and "complete" functions.

Consider implementing the above recommendations to improve the clarity and readability of the codebase.

Update: *Resolved in pull request #756. The team stated:*

> *The linked PR makes the suggested renaming and function ordering changes to improve code quality and readability.*

N-05 Variable Initialized With Its Default Value

Within `ValidatorMessages.sol`, the `index` variable is being initialized with its default value.

To avoid wasting gas, consider not initializing variables with their default values.

Update: *Resolved in pull request #757. The team stated:*

> *The linked PR removes the redundant initialization as suggested.*

N-06 Unused Error

In `ValidatorManager.sol`, the `UnauthorizedCaller` error is unused.

To improve the overall clarity, intentionality, and readability of the codebase, consider either using or removing any currently unused errors.

Update: *Resolved in [pull request #758](). The team stated:*

> *The linked PR removes the unused error as suggested.*

N-07 Unused Named Return Variable

Named return variables are a way to declare variables that are meant to be used within a function's body for the purpose of being returned as that function's output. They are an alternative to explicit in-line `return` statements.

In `ERC20TokenStakingManager.sol`, the [`validationID` return variable]() of the `initiateValidatorRegistration` function is unused.

Consider either using or removing any unused named return variables.

Update: *Resolved in [pull request #759](). The team stated:*

> *The linked PR removes the named return parameter in all declarations and definitions of `initiateValidatorRegistration`, with the exception of `ACP99Manager`, which we keep so that that contract can be as self-documenting of an interface as possible.*

N-08 Unused State Variables

Throughout the codebase, multiple instances of unused state variables were identified:

- In `StakingManager.sol`, the [`P_CHAIN_BLOCKCHAIN_ID` state variable]()
- In `ValidatorManager.sol`, the [`ADDRESS_LENGTH` state variable]()
- In `ERC20TokenStakingManager.sol`, the [`_tokenDecimals` state variable]()

To improve the overall clarity and intent of the codebase, consider removing any unused state variables.

Update: *Resolved in [pull request #760](). The team stated:*

> *The linked PR removes these unused state variables as suggested.*

N-09 Suboptimal Struct Packing

The fields in the `StakingManagerStorage` struct could be re-ordered for more efficient packing, resulting in saving one storage slot. Specifically, `_manager`, `_minimumStakeDuration`, and `_minimumDelegationFeeBips` can be grouped together, as well as `_rewardCalculator` and `_maximumStakeMultiplier`.

Consider reordering the fields in the `StakingManagerStorage` struct to reduce the amount of storage used along with the gas costs associated with storing and accessing data.

Update: *Resolved. The team stated:*

> *We acknowledge that the fields in `StakingMangerStorage` are not optimally packed. However, updating this struct would introduce potential storage collisions with existing deployed contracts, which would need to migrate from the old layout to the new one. Considering this added complexity, we will not change this struct definition.*

N-10 Potential Storage Collision

The `ValidatorManagerStorage` struct has been updated to include a modified version of the `ValidatorChurnPeriod` struct. In the previous version, `ValidatorChurnPeriod` occupied 4 storage slots. On the other hand, the current version occupies 2 storage slots, effectively shifting the layout of subsequent storage variables in the `ValidatorManagerStorage` struct upward by 2 slots.

This change introduces a risk of storage collision in legacy instances of the `ValidatorManager` contract, which might be upgraded via the `migrateFromV1` function, as described in the documentation. However, it has been confirmed that all known deployed instances are using validator-manager-v1.0.0, which includes the updated layout of the `ValidatorChurnPeriod` struct. As a result, the risk of storage collision is significantly reduced under the current deployment landscape.

Consider adding information in the migration documentation for the older instances that are unaware of the latest changes.

Update: *Resolved in pull request #779. The team stated:*

> _ Migration from the v1 validator manager contracts only applies to commits at or after the tag `validator-manager-v1.0.0`. The linked PR explicitly notes this requirement, as suggested._

N-11 Inconsistent Usage of Interfaces

The `ValidatorManager` contract currently defines its functions, events, and key structs directly within the contract file. This contrasts with the `StakingManager` contract, which follows a clearer separation of concerns by organizing functions and struct definitions into dedicated interfaces.

For consistency and improved maintainability, consider adopting a similar structure for the `ValidatorManager` contract by moving function declarations, events, and relevant structs into a dedicated interface file. This would align with existing practices in the codebase and enhance modularity going forward.

Update: *Resolved in pull request #778. The team stated:*

> *The linked PR refactors the inheritance hierarchy to be more self consistent. It adds `IACPManager` and `IValidatorManager`, which contain error, struct, event, and public/external function declarations that were previously declared in `ACPManager` and `ValidatorManager`, respectively.*

Conclusion

The updated version of the Validator Manager contracts continues to provide a comprehensive and adaptable framework for managing validator and delegator lifecycles within the Avalanche L1 chains, in accordance with ACP-77. By leveraging Avalanche Interchain Messaging (ICM) for cross-chain communication with the P-Chain, the system enables smooth coordination of validator registration, deregistration, uptime reporting, and state transitions. The recent architectural refactor into separate `ValidatorManager` and `StakingManager` contracts (supporting both ERC-20 and native tokens) improves modularity, extends code space headroom, and aligns with the ACP-99 design standard—positioning the system for more scalable, long-term maintenance.

Throughout the audit, the primary focus was on validating the integrity of cross-contract interactions introduced by the new split-contract design, while confirming the continued correctness of validator lifecycle flows and ICM messaging behavior. The codebase reflects a solid architectural approach, with well-structured logic and an emphasis on maintainability.

Two high-severity issues were identified. One could result in the `ValidatorManager` contract and the P-Chain going out of sync, potentially leading to discrepancies in validator state between the two systems. The other exposes a risk where validators can unilaterally remove delegators and redirect their rewards to arbitrary addresses, undermining the fairness of the staking process. Addressing these risks is critical to preserving the correctness, integrity, and trustworthiness of validator lifecycle operations and delegation mechanics.

Collaboration with the Ava Labs team was smooth and highly effective. Their responsiveness and contextual clarity were instrumental in understanding the broader protocol and reasoning behind the design choices.