Aztec: Governance Contracts
Security Review

Cantina Managed review by:

Xmxanuel, Lead Security Researcher
Cryptara, Security Researcher

November 12, 2025

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings are a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Aztec Labs was founded in 2017, and has a team of +50 leading zero-knowledge cryptographers, engineers, and business experts. Aztec Labs is developing its namesake products: AZTEC (a privacy-first L2 on Ethereum) and NOIR (the universal ZK language).

From Aug 5th to Aug 12th the Cantina team conducted a review of aztec-packages on commit hash 6496e59d. The team identified a total of **20** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	4	4	0
Low Risk	6	3	2
Gas Optimizations	1	1	0
Informational	9	7	2
Total	**20**	**15**	**4**

3 Findings

3.1 Medium Risk

3.1.1 Missing `instance` in signature digest in `EmpireBase.signal` can lead to signature replay attacks for different rollups

Severity: Medium Risk

Context: EmpireBase.sol#L300

Description: The `signature digest` from `getSignalSignatureDigest()` excludes the rollup instance address, allowing signatures from previous rollup instances to be replayed. If a signature transaction reverts (e.g., due to a proposer change), the signature remains valid and can be reused in a new rollup instance when the same proposer and round occur, provided the proposer has not increased the nonce with another signature. In the worst case, this would allow anyone to `signal` support. The contract is also used as a base contract by the `SlashingProposer`. The same signature replay attack can be used to signal support for a slashing as well.

Recommendation: Include the rollup instance address in the signature digest:

```
  function getSignalSignatureDigest(IPayload _payload, address _signaler, uint256 _round) public view returns
  ↪  (bytes32) {
-    return _hashTypedDataV4(keccak256(abi.encode(SIGNAL_TYPEHASH, _payload, nonces[_signaler], _round)));
+    return _hashTypedDataV4(keccak256(abi.encode(SIGNAL_TYPEHASH, _payload, nonces[_signaler], _round,
↪  getInstance())));
  }
```

Aztec: Fixed in 2e19271.

Cantina Managed: Fixed.

3.1.2 Governance deadlock when `registry.getCanonicalRollup() != GSE.getLatestRollup()` with `GovernanceProposer`

Severity: Medium Risk

Context: GSEPayload.sol#L79

Description: The GSE contract manages staking deposits from rollups, while `registry.getCanonicalRollup()` determines the cannoncial rollup. Although these typically align, the GSE acknowledges they may diverge:

```
// NB: The "latest" rollup in this contract does not technically need to be the "canonical" rollup
// according to the Registry, but in practice, it will be unless the new rollup does not use the GSE.
```

The Problem:

1. Proposal creation: Only block proposers from `registry.getCanonicalRollup()` can signal support via `GovernanceProposer`.
2. Proposal execution: `GSEPayload.amIValid()` requires 2/3 of total supply staked on `GSE.getLatestRollup()` in the post execution state:

```
function amIValid() external view override(IProposerPayload) returns (bool) {
  address latestRollup = GSE.getLatestRollup(); // Uses GSE, not Registry
  uint256 effectiveSupplyOfLatestRollup = GSE.supplyOf(latestRollup) + GSE.supplyOf(bonusInstance);
  require(effectiveSupplyOfLatestRollup > totalSupply * 2 / 3, Errors.GovernanceProposer__GSEPayloadInvalid());
}
```

Deadlock Scenario:

When `registry.getCanonicalRollup() != GSE.getLatestRollup()`:

- Canonical rollup proposers can create proposals.
- But proposals fail execution if 2/3 stake moved to canonical rollup (which isn't GSE's latest).
- Governance becomes blocked until stakers coordinate.
  - 2/3 would have to move back to the old rollup.

- Remaining stakers on the cannoncial are needed to create the proposal.

In our understanding, it would be required to replace the `GovernanceProposer` and `GSEPayload` as part of the governance proposal, which adds a new latest rollup which is not using the `GSE` system.

Because the existing `GSE` calls in the `GSEPayload` are not about the lastest anymore.

However, technically this issue could still occur. For example, if the initial `addRollup` proposal doesn't include the update of the `GovernanceProposer`, the deadlock would occur if the stakers move to the new rollup.

Moving the staking to the new cannonical rollup, would mean it needs to be done without the `moveWithLatestRollup` feature by withdrawing first.

Recommendation: The governance process needs to replace `GovernanceProposer` and `GSEPayload` contract, if the `GSE` shouldn't be used anymore. In addition to prevent the worst case scenario on a technical level. Modify `amIValid()` to skip the 2/3 check when canonical and GSE latest diverge:

```
  function amIValid() external view override(IProposerPayload) returns (bool) {
+   // Skip supply validation if canonical rollup differs from GSE latest
+   // In this case, the new canonical rollup likely doesn't use GSE and no check can be performed
+   if (REGISTRY.getCanonicalRollup() != GSE.getLatestRollup()) {
+     return true;
+   }
+
    uint256 totalSupply = GSE.totalSupply();
    address latestRollup = GSE.getLatestRollup();
    address bonusInstance = GSE.getBonusInstanceAddress();
    uint256 effectiveSupplyOfLatestRollup = GSE.supplyOf(latestRollup) + GSE.supplyOf(bonusInstance);
    require(effectiveSupplyOfLatestRollup > totalSupply * 2 / 3,
    ↪  Errors.GovernanceProposer__GSEPayloadInvalid());
    return true;
  }
```

Aztec: Fixed in 07b9496.

Cantina Managed: Fixed.

3.1.3 Data Desynchronization and Fund Recovery Risk

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: The current implementation creates a critical data synchronization issue between the `Registry` contract and the `RollupCore` contract that can lead to permanent fund loss. While the `Registry` maintains a reference to the reward distributor, the actual reward distribution logic in `RollupCore` operates through its internal configuration via `RewardLib.setConfig()`. This creates two separate sources of truth for the reward distributor address.

When `RollupCore.setRewardConfig()` is called to update the reward distributor configuration, this change is not automatically reflected in the registry. The canonical rollup can only claim rewards from the distributor specified in the rollup's core config, not from the one registered in the registry. If the registry's distributor is updated while the rollup still points to the old one, or vice versa, funds become inaccessible.

The `RewardDistributor.claim()` function restricts access to only the canonical rollup, but if there's a mismatch between which distributor the rollup is configured to use and which one holds the funds, those funds become permanently unrecoverable. The rollup cannot claim from a distributor it's not configured to use, and no other entity has permission to withdraw the funds.

This architectural flaw means that updating the reward distributor configuration in either contract without proper synchronization can orphan funds, as there's no mechanism to ensure both contracts reference the same distributor instance or to recover funds from abandoned distributors.

Recommendation: Implement a unified reward distributor management system that ensures both contracts always reference the same distributor instance. This could be achieved by making the `RollupCore` always fetch the distributor address from the registry, or by implementing a two-step update process that requires both contracts to be updated atomically.

Add validation checks in `RollupCore.setRewardConfig()` to verify that the new distributor's registry reference matches the current registry address, preventing configuration of misaligned distributors. Consider implementing a migration mechanism that automatically transfers remaining funds from old distributors to new ones before allowing configuration updates.

Alternatively, create a fund recovery function in the `RewardDistributor` contract that allows the registry owner (governance) to withdraw remaining assets when distributors are being deprecated. This function should only be callable by the registry owner and should transfer all remaining assets to a designated recovery address.

Aztec: Fixed in 77fedd0.

Cantina Managed: Fixed.

3.1.4 Memory bounds validation in `SignatureLib.reconstructCommitteeFromSigners`

Severity: Medium Risk

Context: SignatureLib.sol#L153

Description: The function reads signature/address data from memory based on bitmap flags without validating bounds, potentially reading beyond the `signaturesOrAddresses` array into adjacent memory from the other memory parameter.

Recommendation: Add bounds checking before memory reads:

```
  function getAddresses(...) {
+   uint256 maxDataPtr;
    assembly {
      dataPtr := add(signaturesOrAddresses, 0x20)
+     maxDataPtr := add(dataPtr, mload(signaturesOrAddresses))
    }

    for (uint256 i = 0; i < _length; ++i) {
      if (isSignatureFlag) {
+       require(dataPtr + SIGNATURE_LENGTH <= maxDataPtr, "Signature bounds exceeded");
        // read signature...
      } else {
+       require(dataPtr + ADDRESS_LENGTH <= maxDataPtr, "Address bounds exceeded");
        // read address...
      }
    }
  }
```

This prevents reading outside the `signaturesOrAddresses` array bounds when malformed bitmap data indicates more signatures than the array contains.

Aztec: Fixed in 7d1ec0d.

Cantina Managed: Fixed.

3.2 Low Risk

3.2.1 Potential reentrancy in `EmpireBase signal` casting

Severity: Low Risk

Context: EmpireBase.sol#L308

Description: The `signal` function calls `selection.getCurrentProposer()` which is not a `view` function before updating `round.lastSignalSlot`. If the rollup contract executes third-party code during `getCurrentProposer()`, reentrancy could allow multiple signals in the same slot because the `require` slot check could be passed.

```
require(currentSlot > round.lastSignalSlot.decompress(), ...);
```

It is still unlikely, since it would require the new rollup version to call third party code as a best practices we would still recommend to prevent it.

Recommendation:** Update `round.lastSignalSlot` immediately after the slot check and before external calls:

```
require(currentSlot > round.lastSignalSlot.decompress(), ...);
round.lastSignalSlot = currentSlot.compress(); // Move before external call
address signaler = selection.getCurrentProposer();
```

Aztec: Fixed in 2e19271.

Cantina Managed: Fixed.

3.2.2 Missing zero address validation in `AddressSnapshotLib.add`

Severity: Low Risk

Context: AddressSnapshotLib.sol#L54

Description: The `add` function does not validate that `_address` is not the zero address before adding it to the set. This could lead to unexpected behavior if zero address is accidentally added to attester sets and interpreted as empty when read with unsafe operations.

Recommendation: Add zero address validation:

```
  function add(SnapshottedAddressSet storage _self, address _address) internal returns (bool) {
+   require(_address != address(0), "AddressSnapshotLib: zero address");
    // Prevent against double insertion
    if (_self.addressToCurrentIndex[_address].exists) {
      return false;
    }
```

Aztec: Fixed in 07650da.

Cantina Managed: Fixed.

3.2.3 `ERC20` ASSET implementation without transfer-to-address-zero check can successfully `finaliseWithdraw` for non-existing ids

Severity: Low Risk

Context: Governance.sol#L353

Description: When `finaliseWithdraw` is called with a non-existing `_withdrawalId`, the function accesses `withdrawals[_withdrawalId]`, which returns a default `Withdrawal` struct with `recipient = address(0)`. The function passes all validation checks but fails during the ERC20 transfer to `address(0)` if the default OpenZeppelin implementation is used. The OpenZeppelin default ERC20 implementation has this check:

```
if (to == address(0)) {
    revert ERC20InvalidReceiver(address(0));
}
```

Other ERC20 implementations, such as Solmate, do not. However, an ERC20 implementation like Solmate would successfully execute the transfer, and `withdrawal.claimed = true` would be set. The `initiateWithdraw` or `proposeWithLock` functions initialize `claimed` with `false` again.

Otherwise, it would be possible to frontrun an `initiateWithdraw` call with a `finaliseWithdraw` using the next auto-incremented `id`, setting `claimed` to `true` and thereby blocking future `finaliseWithdraw` calls → resulting in stuck ASSET tokens.

In the current implementation, `claimed` is set to `false`, so it is not possible to frontrun and permanently lock funds. If an ERC20 implementation without the zero-address check is used, it would only result in an incorrect `claimed` state for a non-existing `_withdrawalId`, which would later be overwritten when actually used. To avoid any of these potential scenarios, add a check to ensure that a `withdrawal` for the corresponding `_withdrawalId` exists.

Recommendation: Add explicit validation for withdrawal existence:

```
  function finaliseWithdraw(uint256 _withdrawalId) external override(IGovernance) {
    Withdrawal storage withdrawal = withdrawals[_withdrawalId];
+   require(withdrawal.recipient != address(0), Errors.Governance__InvalidWithdrawalId(_withdrawalId));
    require(!withdrawal.claimed, Errors.Governance__WithdrawalAlreadyClaimed());
```

```
    require(
      Timestamp.wrap(block.timestamp) >= withdrawal.unlocksAt,
      Errors.Governance__WithdrawalNotUnlockedYet(Timestamp.wrap(block.timestamp), withdrawal.unlocksAt)
    );
    // ...
    }
```

3.2.4 `Bonus` votes remain with `GSE` latest rollup when `registry.canonical` rollup diverges

Severity: Low Risk

Context: GSE.sol#L556

Description: The `GSE.voteWithBonus` function uses `GSE.getLatestRollup()` to authorize bonus voting power instead of verifying the caller is the canonical rollup from the Registry.

```
function voteWithBonus(uint256 _proposalId, uint256 _amount, bool _support) external {
    Timestamp ts = _pendingThrough(_proposalId);
    require(msg.sender == getLatestRollupAt(ts), Errors.GSE__NotLatestRollup(msg.sender));
    // ...
}
```

The bonus votes are a feature of the GSE, which allows attesters to move their stake automatically with the latest rollup. However, technically the edge case where the `GSE.getLatestRollup() != registry.getCanonicalRollup()`, is possible. This would mean the `registry` has a new canonical rollup and the `bonus` voting power stays in the old rollup. This edge case could lead to governance voting with outdated voting power distribution until attesters manually migrate.

Recommendation: In this edge case `GSE.getLatestRollup() != registry.getCanonicalRollup()`, it should be not possible for the `GSE.getLatestRollup()` to use the voting power.

```
function voteWithBonus(...) external {
// Block bonus votes if systems have diverged
require(getLatestRollup() == REGISTRY.getCanonicalRollup(), "Bonus votes disabled during system transition");
    // ... existing checks
}
```

Instead `attesters` should withdraw and deposit into the new rollup contract, which uses a different system to replace `GSE` contracts.

Aztec: The assets are deposited into the GSE for a specific instance, not into the registry, so it should be the GSE's view of things that is the truth. Similarly to if the instance was a user, it should be able to use all the funds delegated to it, not only if canonical from the pov of registry.

Cantina Managed: Acknowledged.

3.2.5 Ownership Control Risk

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: The `CoinIssuer` and `Registry` contracts inherit from OpenZeppelin's `Ownable` contract, which exposes critical ownership management functions that could lead to permanent loss of control. The `renounceOwnership()` function allows the current owner to permanently relinquish ownership to the zero address, while `transferOwnership()` enables immediate ownership transfer without confirmation from the recipient.

In the `CoinIssuer` contract, losing ownership would prevent any future minting operations since the `mint()` function is restricted to `onlyOwner`. This would effectively freeze the coin issuance mechanism, potentially disrupting the entire token economics of the system.

The `Registry` contract is even more critical as it manages the canonical rollup instances and reward distribution. Loss of ownership here would prevent:

- Adding new rollup instances to maintain system upgrades.
- Updating the reward distributor to maintain validator incentives.
- Managing the governance relationship that controls the entire system.

The current implementation relies on external governance contracts to manage these critical functions, but if ownership is lost, there would be no way to recover control or delegate it to new governance contracts.

Recommendation: Override the `renounceOwnership()` function to revert all calls, preventing accidental or malicious relinquishment of ownership. Implement a two-step ownership transfer mechanism by inheriting from `Ownable2Step` instead of `Ownable`, which requires the new owner to explicitly accept ownership before the transfer is completed.

For the `Registry` contract specifically, ensure that the governance contract has a mechanism to call `acceptOwnership()` when ownership is transferred to it. Consider implementing additional access controls or timelock mechanisms for critical functions like `addRollup()` and `updateRewardDistributor()` to provide additional layers of security beyond simple ownership checks.

Alternative approaches include implementing a multi-signature requirement for ownership changes or creating a separate admin role with limited permissions for routine operations while keeping ownership for emergency functions only.

Aztec: Part of the reasoning that we are using just Ownable and not Ownable2Step is to avoid having an initial setup as part of the Governance.sol that ”needs” to know about the CoinIssuer and Registry.

If we are to have them take ownership, we would either need to run a proposal to get ownership, inject knowledge of them into the Governance itself about taking it over or some kind privileged role.

Catina Managed: Acknowledged.

3.2.6 Signature Validation Logic Flaw

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: The `isEmpty()` function in the `SignatureLib` library has a logical flaw that could prevent proper signature validation and committee reconstruction. The current implementation requires both `signatureIndices` and `signaturesOrAddresses` to be empty to return true, but this creates a problematic edge case where one field could be empty while the other contains data.

If `signatureIndices` is empty but `signaturesOrAddresses` contains data, the function would incorrectly return false, indicating the attestations are not empty. However, without the bitmap information from `signatureIndices`, it becomes impossible to determine which bytes in `signaturesOrAddresses` represent signatures versus addresses. This would make the `getSignature()` and `reconstructCommitteeFromSigners()` functions fail or produce incorrect results.

Similarly, if `signaturesOrAddresses` is empty but `signatureIndices` contains bitmap data, the function would return false, but there would be no actual signature or address data to process. The bitmap would be meaningless without corresponding data to interpret.

This flaw could lead to runtime errors when trying to extract signatures from malformed attestation data, potentially causing transaction failures in critical functions like block proposal validation and committee attestation processing within the rollup system.

Recommendation: Modify the `isEmpty()` function to return true when either `signatureIndices` OR `signaturesOrAddresses` is empty, as both fields are required for valid attestation processing. This ensures that incomplete or malformed attestation data is properly identified as empty and prevents attempts to process invalid data.

Consider adding additional validation in functions that consume `CommitteeAttestations` to verify that both fields have consistent lengths and that the bitmap size in `signatureIndices` corresponds to the expected number of committee members. This would provide an additional layer of protection against malformed attestation data.

Aztec: Fixed in 83353d3.

Cantina Managed: Fixed.

3.3 Gas Optimization

3.3.1 Store configuration ID instead of full configuration in `Governance` proposals

Severity: Gas Optimization

Context: Governance.sol#L727

Description: Each proposal stores a complete copy of the `Configuration` struct (9 storage slots) to preserve the configuration at proposal time. This is expensive for every new proposal. This is needed for having a snapshot of the configuration when the proposal has been created.

Recommendation: Store configurations in a mapping with auto-incrementing IDs:

```
mapping(uint256 => Configuration) public configurations;
uint256 public currentConfigId;

function updateConfiguration(Configuration memory _configuration) external onlySelf {
    currentConfigId++;
    configurations[currentConfigId] = _configuration;
}

// In _propose
proposals[proposalId] = Proposal({
    configId: currentConfigId,  // Single storage slot instead of 9
    // ... other fields
});
```

This reduces proposal creation cost from ~9 SSTORE operations to 1.

Aztec: Fixed in 83c6572.

Cantina Managed: Fixed.

3.4 Informational

3.4.1 Consider simpler `IF` statements instead of require with logical `OR` in `GSE.deposit`

Severity: Informational

Context: GSE.sol#L311

Description: The deposit function uses negated conditional logic with OR conditions that make the code harder to understand and maintain. This pattern appears in two different require statements.

Recommendation: Refactor the conditions to use positive logic with if-statements for clarity:

```
//  Current logic
require(!_moveWithLatestRollup || isMsgSenderLatestRollup, Errors.GSE__NotLatestRollup(msg.sender));

// Suggested refactor
if (_moveWithLatestRollup) {
    require(isMsgSenderLatestRollup, Errors.GSE__NotLatestRollup(msg.sender));
}

// Current logic
require(
    !isMsgSenderLatestRollup || !isRegistered(BONUS_INSTANCE_ADDRESS, _attester),
    ↪  Errors.GSE__AlreadyRegistered(BONUS_INSTANCE_ADDRESS, _attester)
);

// Suggested refactor
if (isMsgSenderLatestRollup) {
    require(!isRegistered(BONUS_INSTANCE_ADDRESS, _attester),
    ↪  Errors.GSE__AlreadyRegistered(BONUS_INSTANCE_ADDRESS, _attester));
}
```

This refactoring maintains the exact same logic while significantly improving readability.

Aztec: Fixed in 9a92666.

Cantina Managed: Fixed.

3.4.2 Test-specific code in production `GSE` contract

Severity: Informational

Context: GSE.sol#L335

Description: The GSE contract contains a `checkProofOfPossession` flag explicitly marked `for testing only` that enables critical cryptographic validation for the `deposit` function.

```
// @note  Always true, exists to override to false for testing only.
bool public checkProofOfPossession = true;

// GSE.deposit
if (checkProofOfPossession) {
  // ...
}
```

Recommendation: Remove the `checkProofOfPossession` flag. Use inheritance-based testing approach:

`GSE.sol` - Production contract:

```
- bool public checkProofOfPossession = true;

+ function _validateProofOfPossession(...) internal virtual {
+   // validation logic here
+ }

-  if (checkProofOfPossession) {
-   // validation logic
+   _validateProofOfPossession(...);
  }
```

`TestGSE.sol` - Test contract:

```
contract TestGSE is GSE {
  function _validateProofOfPossession(...) internal override {
    // Skip validation for testing - no op
  }
}
```

Aztec: Fixed in efd6a3e.

Cantina Managed: Fixed.

3.4.3 Unnecessary addition in `GSE.effectiveBalanceOf` due to mutual exclusion

Severity: Informational

Context: GSE.sol#L676

Description: The `effectiveBalanceOf` function adds balances from both the rollup instance and bonus instance, but the `deposit` function's require statements ensure an attester cannot be registered in both locations simultaneously:

```
// Ensure that we are not already attesting on the rollup
require(!isRegistered(msg.sender, _attester), ...);

// Ensure that if we are the latest rollup, we are not already attesting on the bonus instance.
require(!isMsgSenderLatestRollup || !isRegistered(BONUS_INSTANCE_ADDRESS, _attester), ...);
```

Since an attester can only be in either the rollup OR the bonus instance (never both), the addition will always have one operand as zero.

Recommendation: Leverage the mutual exclusion invariant to avoid unnecessary storage reads. This would also make the mutual exclusion more clear.

```
  function effectiveBalanceOf(address _instance, address _attester) external view returns (uint256) {
    uint256 balance = delegation.getBalanceOf(_instance, _attester);
-   if (getLatestRollup() == _instance) {
-     balance += delegation.getBalanceOf(BONUS_INSTANCE_ADDRESS, _attester);
-   }
+   if (balance == 0 && getLatestRollup() == _instance) {
+     return delegation.getBalanceOf(BONUS_INSTANCE_ADDRESS, _attester);
```

```
+   }
    return balance;
  }
```

Aztec: Fixed in 262591f.

Cantina Managed: Fixed.

3.4.4 Inconsistent checkpoint behavior for `indexToAddressHistory` in `AddressSnapshotLib.remove`

Severity: Informational

Context: AddressSnapshotLib.sol#L135

Description: The `AddressSnapshotLib` library maintains historical snapshots of attester sets, allowing the `GSE` to query which attesters were active at any point in time. This is crucial for governance voting power calculations based on past states.

The `remove` function handles the last index differently than other indices for the `indexToAddressHistory` removal, creating an inconsistency in checkpoint updates. The `indexToAddressHistory` mapping stores the historical address values for each index position, enabling time-based lookups of who occupied each slot. When removing the last element (highest index), it pushes `address(0)` to the `indexToAddressHistory` checkpoint.

When removing any other element which requires a swap with the last element before the removal, it doesn't clear the removed element's checkpoint history.

This means the actual removed value of the index would be still accessed with `unsafe` read operations which don't perform size check.

Currently, there is no functional impact due to `size` checks in calling code.

- The `unsafeGetRecentAddressFromIndexAtTimestamp` could return stale addresses for removed indices.

The `GSE` contract properly validates indices before calling unsafe functions, preventing any actual issues. The ”unsafe” naming convention appropriately signals the need for external validation.

Recommendation: For consistency, push `address(0)` to the removed index's checkpoint history in both cases:

```
  if (lastIndex == _index) {
    _self.indexToAddressHistory[_index].push(key, uint224(0));
  } else {
    // Swap the last item with the item we are removing
    _self.addressToCurrentIndex[lastValidator] = Index({exists: true, index: _index.toUint224()});
    _self.indexToAddressHistory[_index].push(key, uint160(lastValidator).toUint224());
+   // Clear the old position of the moved validator
+   _self.indexToAddressHistory[lastIndex].push(key, uint224(0));
  }
```

This ensures consistent behavior: all removed indices get `address(0)` in their checkpoint history.

Aztec: Fixed in 1cec310.

Cantina Managed: Fixed.

3.4.5 Timestamp Comparison Inconsistency

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The `powerAt()` and `totalPowerAt()` functions in the `Governance` contract have a problematic timestamp comparison that could lead to unexpected behavior and potential user confusion. The current implementation checks if the provided timestamp exactly equals `block.timestamp` to determine whether to return the current power value or a historical checkpoint value.

This exact timestamp matching approach is problematic because `block.timestamp` represents the timestamp of the current block, which may not always align perfectly with user expectations. Users calling these functions with the current block's timestamp might receive different results depending on when

exactly their transaction is included in a block, especially in scenarios with varying block times or network congestion.

Additionally, the requirement for exact timestamp matching makes it difficult for external systems or frontends to reliably query current power values, as they would need to know the exact `block.timestamp` value to get current power instead of historical values.

Recommendation: Modify the `powerAt()` and `totalPowerAt()` functions to accept a special value (such as 0) to indicate the caller wants current power values, making the API more intuitive and reliable. This would eliminate the need for exact timestamp matching and provide a stable way to query current power.

Alternatively, create separate view functions like `powerNow()` and `totalPowerNow()` that always return current power values, while keeping the timestamp-based functions for historical queries. This approach maintains backward compatibility while providing clear, stable interfaces for current power queries.

Aztec: Fixed in 884cd9b.

Cantina Managed: Fixed.

3.4.6 Attesters using `_moveWithLatestRollup` might avoid slashing during rollup transitions

Severity: Informational

Context: GSE.sol#L306

Description: When attesters deposit with `_moveWithLatestRollup = true`, their stake moves automatically to new rollups via the bonus instance. This creates a slashing vulnerability during rollup transitions.

The Attack Scenario:

1. Attester deposits on rollup A with `_moveWithLatestRollup = true` (stake goes to bonus instance).
2. Attester produces invalid blocks on rollup A.
3. Before slashing can be executed (which requires time for verification), governance adds rollup B.
4. The attester's stake automatically moves to rollup B (via bonus instance).
5. Rollup A attempts to slash, but the attester is no longer in its effective set.

The Problem:

- Slashing uses `effectiveBalanceOf` which checks both direct and bonus attesters.
- But when a new rollup becomes latest, the bonus attesters immediately move.
- The old rollup loses the ability to slash bonus attesters who misbehaved.

Impact:

- Attesters can produce invalid blocks without consequences if they time it before rollup upgrades.
- Undermines the security model where economic stake ensures honest behavior.
- Creates an incentive to misbehave when rollup transitions are anticipated.

The Aztec team is aware of such potential attacks in their design and communicated it to the Cantina team, therefore this issue is only informational.

Recommendation: Consider implementing a slashing grace period where the previous rollup retains slashing rights over bonus attesters for a defined period after losing ”latest” status. Otherwise, document such potential attacks and the related risk during the transition period clearly to users and block producers.

Aztec: This is acceptable and part of the design.

Cantina Managed: Acknowledged.

3.4.7 Redundant zero check for `minimumVotes` in `ProposalLib`

Severity: Informational

Context: ProposalLib.sol#L105

Description: The function checks if `_self.config.minimumVotes == 0`, but this condition can never be true. The configuration is validated in `ConfigurationLib.assertValid()` which requires:

```
require(_self.minimumVotes >= VOTES_LOWER, /*...*/);  // VOTES_LOWER = 1
```

Recommendation: Remove the redundant check:

```
- if (_self.config.minimumVotes == 0) {
-   return (VoteTabulationReturn.Invalid, VoteTabulationInfo.MinimumEqZero);
- }
```

This simplifies the code and saves gas by removing an impossible branch.

Aztec: Fixed in 744fcf6.

Cantina Managed: Fixed.

3.4.8 Code quality improvements

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: Several minor code quality issues were identified:

1. Missing event emission in `dropProposal` (`Governance.sol:500`).

```
self.cachedState = ProposalState.Dropped;
// Missing: emit ProposalDropped(_proposalId);
```

2. Incomplete comment (`Governance.sol#L395`):

```
// Current: "...since if the msg.sender does not have sufficient balance, the ."
// Should be: "...the _initiateWithdraw would revert with an underflow."
```

3. Incorrect comment about stake check (`GovernanceProposer.sol#L20`):

```
// Says: "...if 2/3 of all stake in the GSE are not staked on the canonical rollup"
// Reality: GSEPayload checks GSE.getLatestRollup(), not canonical rollup
```

4. Misleading error message (`StakingLib.sol#L141`):

```
require(exit.isRecipient, Errors.Staking__NotExiting(_attester));
// Better: Errors.Staking__InitiateWithdrawNeeded(_attester)
// Occurs when attester was partially slashed and needs withdrawer to call initiateWithdraw
```

5. Repeated assembly computation in `SignatureLib.packAttestations` (`SignatureLib.sol#L226,L233,L241`):

```
// Current: Recalculates add(signaturesOrAddresses, 0x20) multiple times
assembly {
  mstore(add(add(signaturesOrAddresses, 0x20), dataIndex), r)
}

// Optimized: Calculate once and reuse
uint256 dataPtrStart;
assembly {
  dataPtrStart := add(signaturesOrAddresses, 0x20)
}
assembly {
  mstore(add(dataPtrStart, dataIndex), r)
}
```

Recommendation:

- Add `ProposalDropped` event for consistency with other state changes.
- Complete the unfinished comment.
- Update the comment to reflect that the check uses GSE's latest rollup, not Registry's canonical.
- Use more descriptive error message that guides users to the correct action.

- Cache assembly pointer calculation to avoid redundant computation.

Aztec: Fixed in a3da5c9.

Cantina Managed: Fixed.

3.4.9 Implicit veto power for 33.4% staker coalition in `GSEPayload.amIValid`

Severity: Informational

Context: GSEPayload.sol#L85

Description: The `GSEPayload.amIValid` check requires > 2/3 of total stake on the canonical rollup at execution time. If current support is just above the 2/3 majority, even less than 33.4% of stakers can block governance proposals by withdrawing stake during the proposal lifecycle. This means individual attesters with sufficient stake can have effective veto power in some cases.

While there is a `withdrawalDelay` configured in `ConfigurationLib`, governance proposals are typically discussed off-chain in forums before submission, providing sufficient time for strategic veto withdrawals.

Recommendation: Document this veto mechanism clearly so all stakeholders understand that large stakers have implicit veto rights over governance proposals regardless of voting outcomes.

Aztec: A comment has been added to document implicit veto power in a3da5c9.

Cantina Managed: Acknowledged.