Security Audit Report

Polygon Q1 2025: New key scheme, Cosmos SDK fork modules changes, Heimdall v2 ABCI++ & custom modules

Last revised 06.05.2025

Authors: Karolos Antoniadis, Mirel Dalčeković, Ivan Golubovic, Anton Kaliaev, Aleksandar Ljahović, Bernd Müller, Aleksandar Stojanović

Contents

Audit overview

The Project

In March 2025, Polygon retained Informal Systems to conduct a comprehensive security audit across four critical components of its blockchain infrastructure. First, the team reviewed Polygon's forks of CometBFT and the Cosmos SDK—each modified to introduce a new Ethereum-style uncompressed `secp256k1` key scheme—ensuring that key handling, signature verification, and consensus interactions remained robust and secure. Next, the auditors examined the Cosmos SDK fork, validating that core SDK modules (`x/bank`, `x/auth`, `x/gov`) adhered to best practices and posed no unforeseen vulnerabilities. The third focus was an in-depth evaluation of the ABCI++ implementation: Informal Systems assessed how vote extensions are generated and consumed, scrutinized the `PrepareProposal`, `ProcessProposal`, etc. workflows. Finally, the team audited Polygon's custom Heimdall v2 modules (`x/checkpoint`, `x/milestone`, `x/topup` and `x/stake`). Collectively, these coordinated audits bolster confidence in Polygon's layered architecture and advance its mission to deliver secure, high-performance Ethereum scaling.

Scope of this report

Ethereum-style `secp256k1` key scheme

The audit focused on evaluating the correctness and security properties of Polygon's forks of CometBFT and the Cosmos SDK to support an uncompressed, Ethereum-compatible `secp256k1` key format. The audit team:

- Compared the modified code against upstream repositories to isolate every change.
- Reviewed each pull request to confirm adherence to crypto-best-practices.
- Ran static analysis and fuzz-testing tools to uncover any edge-case or integration bugs.

Cosmos SDK fork `x/bank`, `x/auth` & `x/gov` changes

Focusing on the `x/bank`, `x/auth` & `x/gov` modules, the audit evaluated the correctness and security properties of the modifications introduced in Polygon's fork relative to Cosmos SDK v0.50.12, and their alignment with the intended design of the Polygon protocol with the goal of detecting security vulnerabilities. In addition to identifying potential vulnerabilities, the audit also aimed to provide recommendations for improving selected design and implementation approaches aligned with the best Cosmos SDK practices.

Heimdall v2 ABCI++ implementation

The audit focused on evaluating the correctness and security properties of Polygon's custom implementation of ABCI++, specifically how vote extensions are utilized, how voting takes place using vote extensions, how `ProcessProposal`, `PrepareProposal`, etc. operate.

Heimdall v2 custom modules

The audit focused on evaluating the correctness and security properties of Polygon's custom Heimdall v2 modules and their alignment with the intended protocol design. The primary goal was to identify vulnerabilities introduced by the upgrade to Cosmos SDK v0.50.12 and the integration of vote extensions, with particular attention to threats from potentially malicious actors (validators).

Although the core of the analysis centered on new or modified component parts, the audit also briefly reviewed migrated logic from Heimdall v1 modules, with lower priority.

The upgrade to CometBFT 0.38 and the integration of ABCI++ were reviewed as part of the audit; however, these are considered infrastructure-level concerns and are addressed in separate sections of the report, not within the scope of this section.

The scope border lines were defined in collaboration with the Polygon team during the threat modeling phase. The scope includes the Heimdall v2 bridge input/output processing but explicitly excludes downstream/usptream handling on Bor chain or Ethereum L1 layer i.e., any processing beyond the Heimdall v2 boundary is out of scope of this audit.

External dependencies, such as the merkletree library, were not audited. We recommend that such dependencies be regularly reviewed for updates and publicly disclosed issues, and that only well-maintained and audited libraries be used in production deployments to minimize systemic risk.

Audit plan

The audit was conducted between March 17, 2025 and April 30, 2025 by the following personnel:

- Karolos Antoniadis
- Mirel Dalčeković
- Ivan Golubovic
- Anton Kaliaev
- Aleksandar Ljahović
- Bernd Müller
- Aleksandar Stojanović

Conclusions

Ethereum-style `secp256k1` key scheme

The audit found that the changes were thoughtfully designed and implemented consistently across both codebases. Critical paths, including key generation, signing, and address derivation, were carefully modified to support the new key types while preserving backward compatibility where needed. No critical security issues were identified during the review. Some edge cases—such as panics on invalid keys or minor gaps in type handling—were analyzed and determined to be acceptable due to context (legacy paths, disabled functionality, or upstream consistency).

The overall quality of the codebase is strong, with careful attention to ensuring safe migration paths and avoiding regressions. Minor recommendations include optimizing key validation earlier during object initialization to improve performance and robustness. However, these suggestions are not blockers. Based on the findings, the code changes are suitable for production use following the planned migration, with legacy code isolated safely and marked for future removal.

Cosmos SDK fork `x/bank`, `x/auth` & `x/gov` changes

The audit of the `x/bank`, `x/auth` and `x/gov` modules was performed based on the documented changes to the default Cosmos SDK implementation. Heimdall specific requirements were addressed in the threat model. Potential threats were analysed in the context of a standalone module and in the context of how Heimdall application is using it.

As a result 7 findings are documented in the *Findings* section. One is classified with `High`, one with `Medium` and one with `Low` severity. The remaining ones are classified as `Informational` and contain improvements and suggestions to the current implementation.

Heimdall v2 ABCI++ implementation

The code is well-written, well-documented, and well-thought out and the ABCI++ requirements are satisfied.

During the course of the audit, the team found two issues where a malicious validator could halt or slow down the chain. Those issues stemmed from missing validation or slight validation discrepancies. To avoid possible issues in the future, we recommend carefully crafting the validation performed in message handlers, `ProcessProposal`, `ExtendVote`, etc. by for example abstracting the always-needed validation in one part so that it is easier to verify that the same validation takes place across different methods.

Heimdall v2 custom modules

The security review of the Heimdall v2 custom modules concluded that no major vulnerabilities were identified, assuming that the core protocol assumptions on which the design is built remain valid.

- The use of vote extensions enforces that any bridged event originating from Ethereum L1 or the Bor chain must be confirmed by at least two-thirds of the validator power before being accepted by Heimdall v2 consensus layer. This consensus mechanism mitigates the risk of a malicious proposer injecting forged events, as Heimdall v2 independently queries source chain nodes to validate the authenticity of each event (side-tx triggered on Heimdall v2).
- Although alternative mechanisms exist for manually triggering cross-chain actions, they are subject to the same consensus rules. Additionally, the Heimdall v2 consensus layer is designed to be exclusively operated by validators, further minimizing the attack surface.
- The audit identified missing or incomplete validations resulting from the migration to Heimdall v2 - most notably, issues related to genesis export and chain initialization functions due to ABCI++ changes. Some of these gaps stem from heavy reliance on the correctness of Ethereum L1 - emitted events and the associated smart contracts. However, none of these findings were considered critical.
- Other findings include protocol inconsistencies, such as the handling of checkpoint acknowledgments (e.g., processing both acknowledgment and non-acknowledgment at the same block height), and the possibility for a malicious validator to inject invalid milestones due to ABCI++ milestone processing. Missing milestone validations across the entire ABCI++ processing phases. These issues were assessed to have minimal or no serious security implications.
- Several additional findings address best practices, including improvements to query validation, input checks, code structure, and minor optimizations as well as and queries module-safe annotation.

While the assessment was conducted under a set of assumptions - particularly concerning the correctness and behavior of the Ethereum L1 contracts and the Bor chain - the audit team also recommended strengthening the Heimdall's v2 resilience by implementing additional safety checks that would mitigate the impact of these assumptions being violated.

These assumptions are explicitly detailed in the *Threat Model* section.

All identified findings are documented in the *Findings* section of this report.

Further Increasing Confidence

As a potential direction for future collaboration, the auditing team suggests considering an expanded scope that includes:

- Review of the Ethereum L1 smart contracts and Bor chain logic. This would help validate the full cross-chain bridging mechanism and confirm that its underlying assumptions hold in practice.
- Additionally, reviewing the bridge processor logic - particularly the components responsible for transferring data **from** the Heimdall v2 chain **to L1 and the Bor** chain, including the retry mechanisms. This could provide further assurance of robustness across the system and confirm no possibility of Heimdall finalized data loss.

The current Heimdall v2 implementation does not include the `x/upgrade` module, which would typically be used to coordinate live chain upgrades. As part of their future plans, the Polygon team intends to integrate the `x/upgrade` module to enable live chain upgrades, although this has been postponed until after the current migration is completed. This could also be potential audit scope in the future.

Audit Dashboard

Target Summary

- **Type: Protocol & Implementation**
- **Platform: Go**
- **Artifacts:**
  - thereum-style `secp256k1` key scheme:
    - crypto folder of CometBFT fork
    - crypto folder of Cosmos SDK fork
  - Cosmos SDK fork `x/bank`, `x/auth` & `x/gov` changes:
    - x/auth
    - x/bank
    - x/gov
  - Heimdall v2 ABCI++ implementation:
    - app/abci.go
    - app/vote_ext_utils.go
    - x/milestone/abci/abci.go
    - sidetxs/side_handler.go
    - sidetxs/ante_decorator.go
    - side_msg_server.go and msg_server.go files of all modules
  - Heimdall v2 custom modules:
    - x/checkpoint
    - x/stake
    - x/topup
    - x/milestone

Engagement Summary

- **Dates**: March 17, 2025 - April 30, 2025
- **Method**: Manual code review, protocol analysis

Severity Summary

Finding Severity	Number
Critical	0
High	6
Medium	3
Low	12
Informational	25
Total	46

System Overview

Ethereum-style `secp256k1` key scheme

The changes in the CometBFT and Cosmos SDK forks were focused on introducing support for a new Ethereum-style uncompressed `secp256k1` key scheme. This update involved adjustments to key serialization formats, address encoding (moving from bech32 to Hex), and signature handling to better align with Ethereum ecosystems. In CometBFT, changes primarily affected the `crypto/` directory, while in the Cosmos SDK fork, they were concentrated in the `crypto` module.

Two parallel key types were introduced: `PrivKey` / `PubKey` (the new Ethereum-compatible types) and `PrivKeyOld` / `PubKeyOld` (the legacy types maintained for Heimdall v1 compatibility and migration support). The modifications ensure that networks upgrading from older systems can migrate smoothly without breaking existing functionality.

Cosmos SDK fork `x/bank`, `x/auth` & `x/gov` changes

The `x/auth`, `x/bank`, and `x/gov` modules of the Cosmos SDK were modified to support Polygon's POL infrastructure, which introduces a separation of concerns across Heimdall consensus, Bor execution, and the Ethereum layer, including staking, rewards, and checkpointing contracts.

This security audit, conducted by the Informal Systems Security team, focused on evaluating these modifications and assessing their potential impact on the Polygon protocol. The specific changes reviewed are detailed in the following sections.

`x/auth` module

To ensure compatibility with the EVM ecosystem, the usage of bech32 addresses was switched to `0x` EVM-compatible addresses.

Method of applying transaction fees was simplified deducting same fee from each transaction, hence new `MaxTxGas uint64` and `TxFees string` module parameters were introduced.

Vesting account support was dropped (they are treated as base accounts) as well as multisignature functionality (`DefaultTxSigLimit` set to 1).

Some of the ante handler decorators were added or disabled due to Heimdall's specific requirements. `SignModeHandler` option was added to `PubKeyDecorator` , `ConsumeGas` was removed from basic decorator and a check that there is only one signer was added.

`x/bank` module

Since staking is handled on Ethereum layer, (un)delegation (also vesting) is not supported and because of that `DelegateCoins` is not supported.

Since POL is the only denom used `msgSetSendEnabled` is not supported.

`x/gov` module

The burn functionality is not supported by Heimdall, so in any case, the deposit is refunded to the proposer. Additionally, Heimdall does not support delegation, meaning inheritance is not applicable.

In the Polygon PoS network, participants are validators, and other holders and users do not have the right to participate in governance. Voting power for participants is calculated solely from their L1 POL stakes. Weighted votes are not supported, therefore, there are checks to ensure the length of weighted vote options equals one.

A dedicated threat model has been created and analyzed for each of the modules described in this overview.

Heimdall v2 ABCI++ implementation

Heimdall v2 uses the ABCI++ interface. By using ABCI++, Heimdall v2 allows validators to perform external calls (e.g., Bor) during `ExtendVote` and hence enables validators to vote on the proposed messages/transactions that reside on a block. This way, only messages that have been voted upon by at least 2/3 of the total voting power can be executed in the `PreBlocker`. The execution of the message is performed through the use of the manually-devised post handlers. Milestones are also automatically generated during `ExtendVote` so that milestones can be generated in a guaranteed cadence, because otherwise, if milestones were to be submitted through messages, then it could take multiple blocks until a milestone goes through.

Heimdall v2 custom modules

Heimdall is a critical component of the Polygon architecture, serving as the consensus and staking layer within its multi-layered design. It is responsible for validating and aggregating validator signatures, producing checkpoints that are submitted to Ethereum, and securely bridging events between Ethereum Layer 1 and Bor, Polygon's block production layer. Built on the Cosmos SDK and Tendermint consensus engine (now CometBFT), Heimdall ensures integrity, finality, and cross-chain communication across the system.

Heimdall v2 represents a major upgrade to the consensus layer. It integrates newer versions of CometBFT and the Cosmos SDK, and adopts ABCI++ and vote extensions to improve cross-chain event verification and state synchronization. The system design now allows certain messages to be processed through so-called *side transactions* (side-tx), enhancing protocol efficiency.

This audit focused on the set of custom Heimdall v2 modules that were adapted to leverage vote extensions and the updated Cosmos SDK. These modules play a vital role in ensuring the correctness and security of bridging operations under the new architecture.

Checkpoints (`x/checkpoint`)

Checkpoint Proposal

A checkpoint proposal is initiated by a proposer, a validator with POL tokens staked on the L1 Ethereum root chain. The checkpointing process is managed by the bridge processor which generates a `MsgCheckpoint` and broadcasts it as a transaction.

- The proposer derives the root hash from the Bor chain contract.
- Due to Bor's finality time of 60-90 seconds, the root hash may not always reflect the latest Bor tip.

Checkpoint Processing in Heimdall

Once the checkpoint message is included in a Heimdall block, it undergoes processing through the message handling system. Each validator node independently verifies the checkpoint by checking the Bor root hash provided in the message against its local Bor chain.

ABCI++ Processing Flow for the checkpoint submission on Heimdall

- Prepare Proposal: During the proposal phase, the checkpoint message `MsgCheckpoint` is included in the proposed block only if dry-running this tx does not return any errors.
- Process Proposal: The proposal is validated to ensure correctness.
- Pre-Commit: As part of the voting process, validators execute a side transaction to verify the checkpoint against their local Bor data.
  - If the checkpoint is valid, validators include a vote extension confirming their approval.
- Verify Vote: Injected votes are verified.
- Next block - Finalize: In the next block, the finalized votes are processed, and the checkpoint is considered approved if a sufficient majority supports it. The pre-blocker triggers post-tx handlers performing the Heimdall state changes when the checkpoint is finally saved in the checkpoint buffer as the checkpoint that needs to be further bridged to the Ethereum L1 root chain.

Submission to Ethereum (L1)

Once approved, the checkpoint is added to a checkpoint buffer and an event is emitted. The bridge system, which listens for these events, submits the checkpoint data along with validator signatures to the Ethereum root chain.

Acknowledgment from Ethereum (L1)

After the checkpoint is successfully included on the Ethereum chain, an acknowledgment `MsgCpAck` is sent back to Heimdall from the bridge processor. This acknowledgment, once processed through the ABCI++ flow with side and post-tx handlers: updates the state, flushes processed checkpoints from the buffer, and increments number of ACK counters to track confirmations of checkpoints. Additionally, the selection of the next checkpoint proposer is adjusted based on the updated state.

Missing Checkpoint Acknowledgment from Ethereum (L1)

The `MsgCpNoAck` message is broadcast by the bridge processor to indicate that a checkpoint was potentially transferred to the Ethereum chain but has not received an acknowledgment. A background routine periodically checks for time elapsed and publishes the No-ACK signal. No-ACK is sent if a sufficient amount of time has passed since:

- the last checkpoint was created on the Heimdall v2 chain and
- the last No-ACK was issued.

To conclude, the No-ACKs are triggered only when a checkpoint acknowledgment is overdue, ensuring they are not sent too frequently. This message is broadcasted only by the proposer.

This entire flow ensures that checkpoints are securely proposed, verified, and finalized across the Heimdall and Ethereum chains in a decentralized manner.

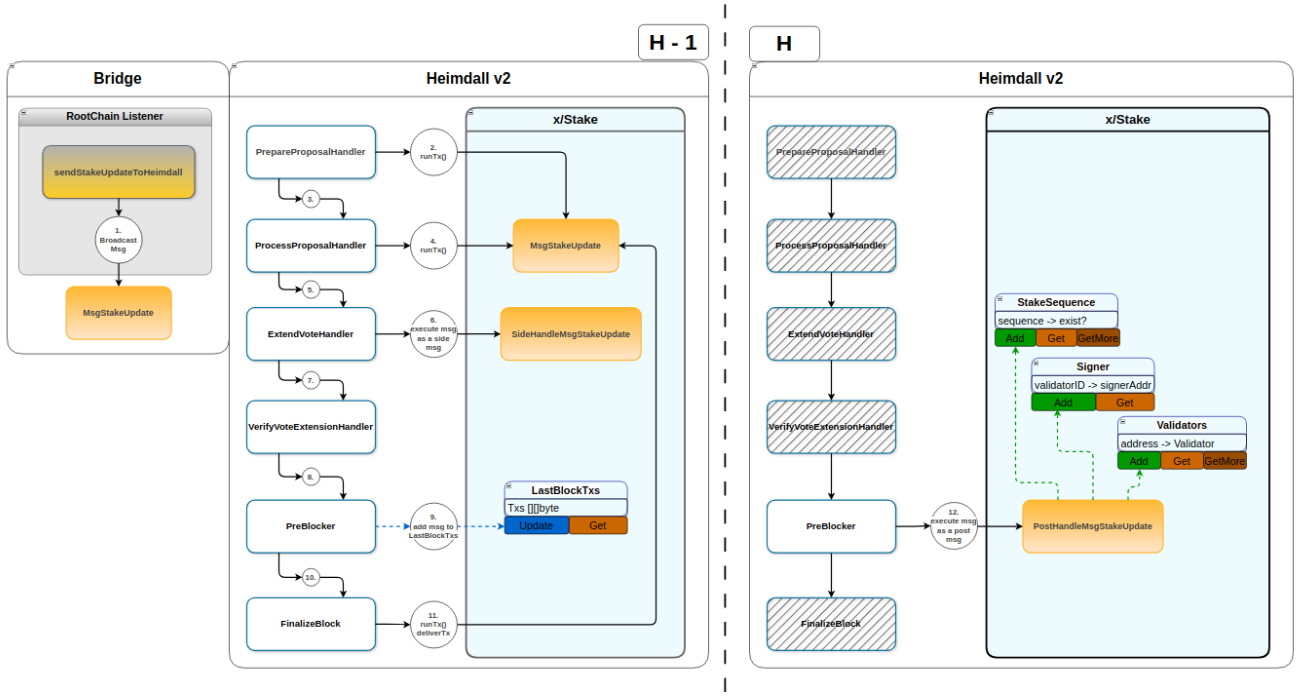

Figure 1: `x/stake` and `x/checkpoint` modules and bridging

Validator management (`x/stake`)

The `x/stake` module manages validator-related transactions and validator set management for Heimdall v2. Validators stake their tokens on the Ethereum chain to participate in consensus. To synchronize these changes with Heimdall, the bridge processor broadcasts the corresponding transaction for an Ethereum-emitted event, choosing from one of the following messages each with the necessary parameters:

- `MsgValidatorJoin`: This message is triggered when a new validator joins the system by interacting with `StakingManager.sol` on Ethereum. The action emits a `Staked` event to recognize and process the validator's participation.
- `MsgStakeUpdate`: Used to handle stake modifications, this message is sent when a validator re-stakes or receives additional delegation. Both scenarios trigger a `StakeUpdate` event on Ethereum, ensuring Heimdall v2 accurately updates the validator's stake information.
- `MsgValidatorExit`: When a validator decides to exit, they initiate the process on Ethereum, leading to the emission of a `UnstakeInit` event. This message ensures that Heimdall records the validator's departure accordingly.
- `MsgSignerUpdate`: This message is responsible for processing changes to a validator's signer key. When a validator updates their signer key on Ethereum, it emits a `SignerUpdate` event, prompting Heimdall v2 to reflect the new signer key in its records.

Each of these transactions in Heimdall v2 follows the same processing mechanisms, leveraging ABCI++ phases. During the `PreCommit` phase, side transaction handlers are triggered, and a vote is injected after validating the Ethereum-emitted event and ensuring its alignment with the data in the processed message. Once a majority of validators confirm that the action described in the message has occurred on Ethereum, the `x/stake` module updates the validator's state in Heimdall v2 during the `FinalizeBlock`'s `PreBlocker` execution.

Replay Prevention Mechanism

Heimdall v2 employs a replay prevention mechanism in the post-tx handler functions to ensure that validator update messages derived from Ethereum events are not processed multiple times. This mechanism prevents replay attacks by assigning a unique sequence number to each transaction and verifying whether it has already been processed. The sequence number is constructed using the Ethereum block number and log index, following the formula:

```
sequence = (block number × DefaultLogIndexUnit) + log index
```

where:

- `msg.BlockNumber` represents the Ethereum block where the event was emitted.
- `msg.LogIndex` is the position of the log entry within that block.
- `DefaultLogIndexUnit` (set to 100,000) ensures uniqueness when combining block numbers and log indexes.

Before processing a transaction, Heimdall checks its **stake keeper** to determine if the sequence number has been recorded. If the sequence is found, the transaction is rejected as a duplicate. Once the post-tx handler completes successfully, the sequence is stored, ensuring that any future message with the same sequence is recognized and ignored.

This approach guarantees that Heimdall only processes each valid Ethereum signer update once, preventing unintended state changes due to replayed messages.

Updating the Validator Set

In the `x/stake` `EndBlocker`, Heimdall v2 updates the validator set (through the `ApplyAndReturnValidatorSetUpdates` function), ensuring consensus reflects the latest validator changes. Before any updates, the current block's validator set is stored as the previous block's set. The system retrieves all existing validators, the current validator set, and the acknowledgment count from the `x/checkpoint` state. Using `GetUpdatedValidators`, a list of validators that require updates (`setUpdates`) is identified and applied through `UpdateWithChangeSet`, storing the new set under `CurrentMilestoneValidatorSetKey`.

To maintain fair block proposer selection, Heimdall implements a **proposer priority system**, ensuring all validators have an equitable chance to propose new blocks. The **proposer priority** is dynamically adjusted using `IncrementProposerPriority(times int)`, which prevents any validator from monopolizing block proposals. This function limits priority differences by re-scaling priorities (`RescalePriorities(diffMax)`) and shifting values based on the average proposer priority (`shiftByAvgProposerPriority()`). During each round, the validator with the highest priority is selected as the proposer, after which their priority is adjusted to prevent indefinite accumulation.

These mechanisms collectively ensure efficient and fair validator rotation, maintaining a balanced consensus process while preventing priority overflows and unfair selection biases.

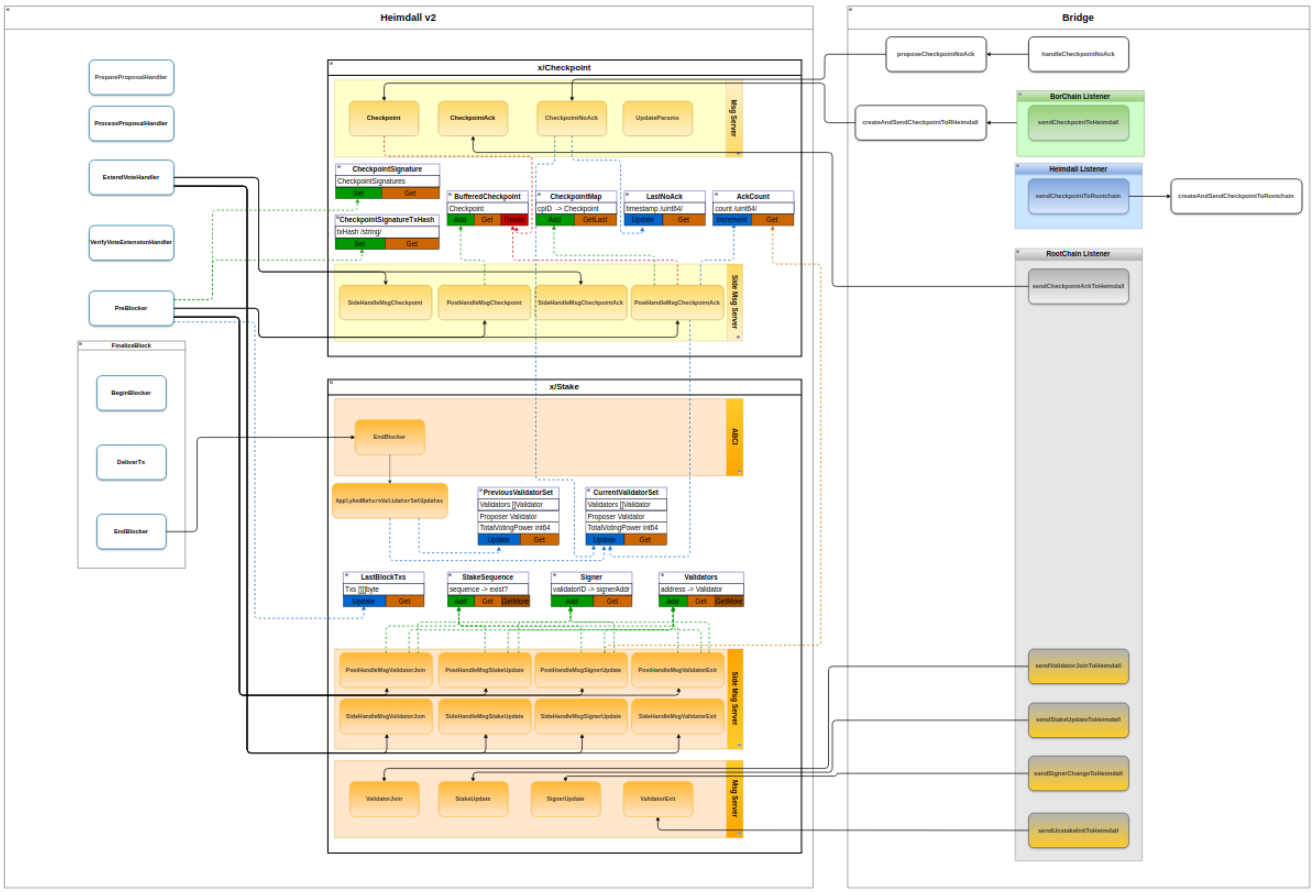

Figure 2: ABCI ++ processing flow

Top-up mechanism (`x/topup`)

The Heimdall Top-up Mechanism facilitates the management of validator fees on the Heimdall v2 chain by allowing deposits from the Ethereum (L1) root chain. This mechanism ensures validators have sufficient balance on Heimdall to cover operational fees. The system integrates staking smart contract on Ethereum with Heimdall custom `x/topup` and `x/checkpoint` module and a bridge components to enable seamless cross-chain fee management.

Top-Up Funding Methods

There are two primary ways to fund a validator's fee balance on Heimdall:

1. **During Validator Initialization**: When a new validator joins, they can specify a **top-up** amount in addition to the **staked** amount. This top-up is transferred to Heimdall as an initial balance.
2. **Direct Top-Up**: Any user can invoke the top-up function on the Ethereum staking smart contract to increase the validators' top-up balance on Heimdall.

Bridge Processing

Top-up events on the Ethereum layer trigger automated processing through the bridge process:

- The Root Chain Log Listener monitors for `StakinginfoTopUpFee` events.
- the Top-up Fee Processor task, upon detecting such an event, triggers the `sendTopUpFeeToHeimdall` execution, which:
  - Decodes the Ethereum log.
  - Verifies the event hasn't already been processed.
  - Broadcasts a `MsgTopupTx` to the Heimdall chain.

In addition to the automated broadcasting of `MsgTopupTx` transactions, it is also possible to manually craft and submit these transactions at the Heimdall v2 layer. This fallback mechanism is used in scenarios where issues arise in bridging or processing Ethereum events.

Although Heimdall primarily handles consensus and doesn't process typical transactions, there is an internal monitoring system in place to ensure validator operations run smoothly. A scheduled job, integrated with alerting tools, periodically checks validator accounts for low balances. If a validator is running low on funds - preventing them from submitting checkpoints to Ethereum L1 - the support team is alerted and notifies the validator directly. This proactive approach helps prevent missed checkpoints, loss of rewards, and potential disruptions to consensus.

Heimdall x/topup Module Implementation

Two core messages are defined in the `x/topup` module for fee management:

`MsgTopupTx`: Handles minting the top-up amount on Heimdall based on Ethereum events. Each top-up is uniquely identified by a sequence number built from `TxHash` and `LogIndex` to prevent duplicate processing. `MsgTopupTx` is a side-transaction, ensuring state changes only after successful pre-commit majority of the votes are collected and final validation and post-tx handler execution in the following block height. When broadcasting the `MsgTopupTx` - sender (proposer of the topup) must sign it, and additional user address must be sent. For the top-up to be accepted, the `MsgTopupTx.Fee` must be at least equal to the `DefaultFeeWantedPerTx` amount.

The top-up processing on Heimdall v2 involves:

- Minting the top-up amount of pol tokens to the top-up module account.
- Transferring the entire amount from the top-up module account to the user account.
- Transferring the `DefaultFeeWantedPerTx` amount from the user account to the proposer validator account. The remaining top- up amount stays on the user account.

`MsgWithdrawFeeTx`: Allows validators to withdraw fees from Heimdall back to Ethereum. The withdrawal process involves:

- Transferring the amount from the validator to the top-up module account.
- Burning the amount from the top-up module account.
- Updating the validator's **dividend "account"** with the withdrawn amount.
- No impact on user account used during the `MsgTopupTx`.

Dividend Account and Checkpoints

A **dividend account** is a specialized state entry (distinct from user/module accounts) that tracks:

- The validator's Heimdall address.
- The total withdrawn fee amount.

This data is included in Heimdall v2 checkpoints through the `AccountRootHash` and through bridging the accepted checkpoints the withdrawn amounts are transferred to the Ethereum root chain. The logic implemented in staking Ethereum smart contract is handling the correct processing of the dividend account state bridged with the checkpoints.

Assumptions Behind Protocol and Module Design

The design of the `x/topup` module in Heimdall v2 operates under a set of implicit assumptions derived from the broader protocol architecture. These assumptions are described in more detail in *Threat Model* chapter.

Key protocol assumptions influencing the `x/topup` module include:

- **Ethereum Event Integrity**: It is assumed Ethereum events is correct and complete and **Event Ordering**: Events are expected to be emitted in the same order as the actions they represent were processed. As a result, Heimdall v2 does not handle or attempt to reorder transactions related to bridged events (e.g., `MsgSignerUpdate`, `MsgTopupTx`).
- **Dividend Account Logic**: The module assumes that dividend withdrawals are correctly handled on the Ethereum side.

By design, the `x/topup` module assumes these invariants hold true and omits internal mechanisms to validate or enforce them. This makes it critical for upstream components and event emitters to maintain consistent and correct behavior to ensure the proper functioning of the system.

Milestones in Heimdall v2

Milestones are a lightweight alternative to checkpoints in Heimdall v2, used to finalize blocks more efficiently. With the introduction of milestones, finality is deterministic even before a checkpoint is submitted to L1. Unlike the original transaction-based design in Heimdall v1, the current design operates without transactions, relying entirely on ABCI++ flow and validator vote extensions. Each validator proposes a milestone independently. Milestones are proposed as series of recent, up to 10 block hashes, and a majority (2/3 of voting power) agreement on a consecutive sequence of these hashes is required to finalize a milestone. The system tolerates duplication to increase reliability and includes logic for resolving forks and ensuring milestone continuity.

Milestone Proposals and Duplication

Validators independently propose a sequence of block hashes, at most `MaxMilestonePropositionLength` (default value: 10) starting from the last finalized milestone:

```
message MilestoneProposition {
  option (gogoproto.equal) = true;
  option (gogoproto.goproto_getters) = true;

  repeated bytes block_hashes = 1 [ (amino.dont_omitempty) = true ];
  uint64 start_block_number = 2 [ (amino.dont_omitempty) = true ];
}
```

Proposals are handled in the `ExtendVoteHandler`, executed at each block (code ref1, ref2). A milestone proposed in block `N` is finalized in block `N+1` (or later), introducing acceptable duplication of proposed milestones to improve reliability. This duplication ensures that even if a milestone isn't finalized in `N+1`, it may succeed in `N+2` or later.

In cases of failed milestone proposition - the node still participates in the consensus (code ref).

Proposed Milestone validation checks

Proposed milestone validation is performed in `N` block with `ValidateMilestoneProposition` function in `ExtendVoteHandler`(code ref) and in `VerifyVoteExtensionHandler` (code ref):

- Length is validated - the milestone proposal, if created (code ref) should not contain more block hashes than `MaxMilestonePropositionLength` (code ref);
- Each block hash length is validated to be of the appropriate length (code ref) .

Majority Determination in the following block

Vote extensions from other validators are collected and unmarshaled. Duplicate vote extensions from the same validator are ignored. The algorithm uses data structures keyed by `(block_number, block_hash)` to handle forks (same block number may have different hashes), so that fork-resilience is achieved by:

- Separating vote data by hash and block number.
- Ensuring finalized milestones continue from the last one with no gaps.

The core algorithm looks for:

- The longest consecutive sequence of block hashes.
- Supported by >= 2/3 of the total voting power.

Milestones Validation and Finalization

Once a consensus over the milestone is reached - the **majority milestone** is validated with `ValidateMilestoneProposition` checks again for integrity in `PreBlocker` (code ref).

If the validation passes, the milestone is persisted (code ref).

A dedicated threat model has been created and analyzed for each of the modules described in this overview.

Threat Model (Ethereum-style `secp256k1` key scheme)

We conducted a focused review of changes to the `crypto/` directory of CometBFT fork (tagged audit release) and changes to `crypto` module of Cosmos SDK fork (tagged audit release) to add support of a new key scheme that aligns with Ethereum-style uncompressed `secp256k1` keys, featuring updated key serialization, address encoding (Hex instead of bech32), and signature generation/verification.

The new key scheme introduces two key types:

- `PrivKey` / `PubKey`: current key types based on CometBFT, using an Ethereum-compatible key scheme.
- `PrivKeyOld` / `PubKeyOld`: legacy types for Heimdall v1 compatibility, using the same keys but maintained for migration support.

The audit team used a comprehensive diff review to:

- Trace all code changes related to the new key scheme.
- Compare behavior between new and legacy key types.
- Identify any missing validations, unhandled edge cases, or inconsistent implementations.
- Evaluate test coverage for critical paths.

Inspection of `crypto` in Cosmos SDK fork

The table below summarizes the results of the file-by-file review:

File	Observations
`crypto/armor_test.go`	No issues identified.
`crypto/codec/amino.go`	No issues identified.
`crypto/codec/cmt.go`	Function `FromCmtProtoPublicKey` lacks a match arm for `*cmtprotocrypto.PublicKey_Secp256K1`; likely intentional.
`crypto/hd/fundraiser_test.go`	No issues identified.
`crypto/hd/hdpath_test.go`	No issues identified.
`crypto/keyring/doc.go`	No issues identified.
`crypto/keyring.go`	`SignWithLedger` is not covered by tests.
`crypto/keyring/keyring_linux.go`	No issues identified.
`crypto/keyring/keyring_linux_test.go`	`TestNewKeyctlKeyring` does not cover all possible cases.
`crypto/keyring/keyring_other.go`	No issues identified.
`crypto/keyring/types_test.go`	No issues identified.
`crypto/keys/bcrypt/bcrypt.go`	No issues identified.
`crypto/keys/ed25519/ed25519.go`	No issues identified.
`crypto/keys/multisig/multisig_test.go`	No issues identified.
`crypto/keys/secp256k1/internal/secp256k1/panic_cb.go`	No issues identified.
`crypto/keys/secp256k1/keys.pb.go`	Auto-generated.
`crypto/keys/secp256k1/secp256k1.go`	`PrivKeyOld.PubKey()` panics if the key is invalid. `UnmarshalAmino` only checks size, not key validity. Same for `PrivKey`. `PubKeyOld.Address()` checks `PubKeySize` during call, not during construction; rationale unclear.
`crypto/keys/secp256k1/secp256k1_nocgo.go`	No issues identified.
`crypto/keys/secp256k1/secp256k1_test.go`	No issues identified.
`crypto/keys/secp256r1/privkey.go`	`ecdsaSK.UnmarshalJSON` does not validate that `len(data) > 0`.
`crypto/keys/secp256r1/privkey_internal_test.go`	No issues identified.

File	Observations
crypto/keys/secp256r1/pubkey.go	ecdsaPK.UnmarshalJSON does not validate that len(data) > 0.
crypto/keys/secp256r1/pubkey_internal_test.go	No issues identified.
crypto/ledger/ledger_test.go	No issues identified.
crypto/types/compact_bit_array_test.go	No issues identified.

Several of the initially identified issues were reviewed and discussed with the development team. Based on these discussions, some concerns were determined to be non-issues due to specific contextual factors:

Panic in PrivKeyOld.PubKey() and PrivKey.PubKey()

These methods panic if ethCrypto.ToECDSA()fails, while UnmarshalAmino does not validate key correctness, only the length.

Resolution: This path is tied to legacy Amino usage, which the developers confirmed is no longer active. The code exists for compatibility with Heimdall v1 and is planned for eventual deprecation.

Use of secp256r1 JSON un-marshaling without size validation

Resolution: Developers confirmed that private keys are never received over the network, and the code matches upstream Cosmos SDK behavior. Additionally, secp256r1 is not actively used and may be disabled in the future.

Missing type support in FromCmtProtoPublicKey for Secp256K1

Resolution: This was likely intentional due to the exclusive use of custom key types, and does not affect current functionality.

Key Size Checks in PubKeyOld.Address()

Resolution: While key size could be validated earlier, the current implementation is safe and prevents misuse.

Overall, the changes introduced to support the new key scheme were implemented consistently, with no critical issues identified. Legacy code paths related to Heimdall v1 are maintained in a safe and isolated manner, with plans for future removal. The reviewed code aligns with its intended purpose, and the inspection confirms the changes are suitable for production use following the planned migration.

Inspection of crypto/ in CometBFT fork

The table below summarizes the results of the file-by-file review:

File	Observations
crypto/batch/batch.go	No issues identified.
crypto/encoding/codec.go	*pc.PublicKey_Secp256K1 match arm is gone; likely intentional
crypto/merkle/bench_test.go	No issues identified.
crypto/merkle/hash.go	No issues identified.
crypto/merkle/proof.go	No issues identified.
crypto/secp256k1/secp256k1.go	PrivKey.PubKey() panics if the key is invalid.
crypto/secp256k1/secp256k1_internal_test.go	No issues identified.
crypto/tmhash/bench_test.go	No issues identified.
crypto/tmhash/hash.go	No issues identified.

Several issues were initially identified and discussed with the development team. Through these discussions, certain concerns were determined to be non-issues due to specific context:

Panic in `PrivKey.PubKey()` and `PrivKeyOld.PubKey()`

These methods panic if `ethCrypto.ToECDSA()` fails.

Resolution: The recommendation is to call `ethCrypto.ToECDSA()` once during initialization and store the resulting `privateObject` inside `PrivKey(Old)` struct to:

1. prevent potential conversion errors,
2. improve performance (as `PrivKey.PubKey()` is called during each node network connection and every time the node signs something).

Note: There are no safety concerns when calling `ethCrypto.ToECDSA()` inside `PrivKey.PubKey()` and `PrivKeyOld.PubKey()` as long as the private key generation is correct.

Overall, the changes supporting the new key scheme were implemented consistently, with no critical issues found. Legacy code paths for Heimdall v1 are maintained safely and separately, with future removal planned. The reviewed code fulfills its intended purpose and is suitable for production after the planned migration.

Inspection of other changes in CometBFT fork

Additionally, we analyzed changes outside the `/crypto` folder. Most changes originate from the upstream CometBFT repository, with a few relating to the addition of a new key scheme.

File	Observations
`p2p/conn/secret_connection.go`	`MakeSecretConnection` doesn't have an arm for `secp256k1.PubKeyOld`

This issue was discussed with the development team. The absence of a match arm for `secp256k1.PubKeyOld` prevents old nodes from connecting. However, since all nodes will migrate to the new key scheme, the development team confirmed this is not a concern.

Verification using automated tooling

The audit team ran the existing set of e2e tests (`test/e2e`) to verify system stability and developed several fuzz tests to ensure `PrivKey.PubKey()` derivation remains reliable under all conditions. The fuzz tests can be found below:

```
func FuzzGenPrivKeySecp256k1_NoPanic(f *testing.F) {
  f.Fuzz(func(t *testing.T, in []byte) {
    privKey := secp256k1.GenPrivKey()
    if privKey == nil {
      t.Fatalf("%v: GenPrivKey: nil privKey", in)
    }
    pubKey := privKey.PubKey()
    if pubKey == nil {
      t.Fatalf("%v: PubKey: nil pubKey", in)
    }
  })
}

func FuzzGenPrivKeySecp256k1_NoPanic2(f *testing.F) {
  f.Fuzz(func(t *testing.T, in []byte) {
    privKey := secp256k1.GenPrivKeySecp256k1(in)
    if privKey == nil {
```

```
      t.Fatalf("%v: GenPrivKeySecp256k1: nil privKey", in)
    }
    pubKey := privKey.PubKey()
    if pubKey == nil {
      t.Fatalf("%v: PubKey: nil pubKey", in)
    }
  })
}
```

Threat Model (Cosmos SDK fork `x/bank`, `x/auth` & `x/gov` changes)

As part of our threat analysis, we begin by identifying a set of properties essential to the correctness of the audited scope - namely, the modifications to the `x/auth`, `x/bank`, and `x/gov` modules, as well as their interactions and impact on other components of the Polygon system. These properties are categorized into safety, liveness, and business-related non-functional concerns.

For each property, we define one or more associated threats. Each threat is then individually analyzed to determine whether it could be violated, with any violations resulting in the findings presented in the *Findings* section of this report.

The threat model presented here outlines the identified properties and related threats for the modified Cosmos SDK modules (`x/auth`, `x/bank`, and `x/gov`) in Polygon's fork of the Cosmos SDK (`v0.1.16-beta-polygon`), which is based on version `v0.50.12` of the upstream SDK.

The analysis and threat definitions were tailored to reflect the specific differences between the upstream `v0.50.12` release and Polygon's modified `v0.1.16-beta-polygon` version.

Property AUTH-1: Every Transaction Needs to be Authenticated

Category: Safety property

`x/auth` module ensures that only transactions with authenticated signatures are accepted (multi-sig transactions are not supported)

Violation consequences

- Potential vulnerabilities in transaction authentication mechanisms could allow unauthorized transactions being executed.

Threats

- Malicious transaction with invalid signature is not rejected.
- Transaction with valid signature is rejected.

Conclusion

The property holds.

Ante decorators chain related to signature verification was examined. Changes that were introduced are inclusion of `signModeHandler` into `SetPubKeyDecorator` and check for exclusively one signer in `SigGasConsumeDecorator` which are not introducing any transaction authentication problems.

`signModeHandler` except initialization in constructor (ref1) isn't utilized anywhere by `SetPubKeyDecorator` , neither in cosmosSDK fork nor Heimdall-v2 app. It is worth mentioning that `signModeHandler` is used inside `SigVerificationDecorator`.

Property AUTH-2: Transaction Encoding Validation

Category: Safety property

`x/auth` module accepts only valid transaction encodings (incl. address format and related encoding)

Violation consequences

- Exploitation of decoding or processing of transaction can impact system liveness.

Threats

- Invalid transaction encodings are not rejected.
- Valid transaction encodings are rejected.

Conclusion

The property holds.

Before ante handlers verification Tx is decoded. `DefaultTxDecoder` is used (ref1, ref2) which rejects bad Tx formats - no changes to it were introduced with code changes on the fork.

New `hex_codec` is introduced (ref). **External** `FromHex` and `Bytes2Hex` functions are used from Bor repositories `common` package (ref1, ref2).

Hex address verification is also imported from Bor repository. Prefix, length and is it valid hex string checks are done. The only missing check is checksum for Ethereum address (ERC-55) which doesn't pose a big issue since at this point user errors of manual address input should have been eliminated long ago.

Property AUTH-3: Correct Order of Transaction Validation

Category: Safety property

`x/auth` module performs critical checks on transactions in correct order

Violation consequences

- Insufficient upfront checks by ante handler can leverage unexpected message execution.

Threats

- Improper ordering or implementation of ante-decorators could allow bypass of critical checks

Conclusion

The property holds.

In the diff there were no changes in the ante decorators order, it follows the order stated in documentation. Only difference comparing to documentation is that `MempoolFeeDecorator` is omitted (it was done before diff). Other than decorator changes described in *Property AUTH-1: Every Transaction Needs to be Authenticated*, `ConsumeTxSizeGasDecorator` also checks for exclusively one signer. `DeductFeeDecorator` drops the support for fee granters and deducts fixed transaction fees (detailed in *Property AUTH-4: Fee Validation*).

None of the changes above allow bypass of critical checks.

Property AUTH-4: Fee Validation

Findings: Ante Decorator Deducting Signature Verification Fees

Category: Safety property

`x/auth` module rejects transactions with invalid fees.

Violation consequences

- Missing spam protections due to incorrect validation of transaction fees

Threats

- Manipulation of fee calculation or gas meter settings could allow execution of transactions without proper fees.
- Bypass of minimum gas price enables spamming.

Conclusion

The property holds.

Gas consumption in `ConsumeTxSizeGasDecorator` is disabled, but `DeductFeeDecorator` deducts constant fee per transaction, that is calculated using modules parameters (ref1, ref2, ref3). `SigGasConsumeDecorator` consumes gas required for signature verification but since Heimdall currently uses (checked with Polygon team) only one signature verification algorithm (secp256k1), the fee is consistent across all transactions.

Property AUTH5: Account Verification

Category: Safety property

`x/auth` module verifies addresses for module and other accounts.

Violation consequences

- Invalid accounts are not rejected.

Threats

- Vulnerabilities in key management or account handling could lead to account takeover.

Conclusion

The property holds.

The address of an account is used as the primary key in the store where the auth module is storing the accounts. This address is a hex string which is received from `BytesToString()` of the used address codec. `hex-codec` of `x/auth` module is used as the address codec of the account keeper. It ensures that the account address can represent a valid hex encoded Ethereum address and returns the address in a normalized format (lowered , `0x`-prefix, fixed length hex-string), needed to guarantee unique address identifiers.

Property AUTH-6: Property: Authentication Parameter Verification

Findings: Improper Validation in x/auth Module Allows for Negative Transaction Fees, TxSigLimit Validation Does Not Enforce Heimdall's Lack of Multisig Support

Category: Safety property

`x/auth` module only accepts modifications to valid authentication parameters

Violation consequences

Unauthorized modification of authentication parameters could disrupt system operation

Threats

- Unauthorized update of an authentication parameter `max_tx_gas, tx_fees` is not rejected
- Incorrect or malicious initialization of auth parameter in genesis is not rejected

Conclusion

The property does not hold.

Parameters update has to be done through governance (ref1, ref2), governance module allows `MsgUpdateParams` message for `x/auth` module (ref).

Validation of all new parameters is done during update (ref1, ref2).

- `TxFees` could be set to a negative number (ref). Possibility for this is low because update is done through governance.

Validation of parameters during genesis is done by `ValidateGenesis()` called inside `InitChainer()` (ref) where in case of `x/auth` parameters same validation is done as when parameters are being updated.

Because Heimdall doesn't support multisig, validation that enforces `TxSigLimit` to be set to 1 could be added to the existing validations (ref).

Property AUTH-7: Tracking of Proposer

Findings: Block Proposer Tracking Failure in Heimdall v2 Causes Reward Distribution Malfunction

Category: Safety property

`x/auth` module stores and updates proposer correctly for each block

Violation consequences

- Incorrect proposer tracking per block would lead to impact fee model.

Threats

- Corruption/inconsistency in storage of block proposer.

Conclusion

The property does not hold.

Inside `x/auth` keeper new auxiliary `GetBlockProposer` , `SetBlockProposer` and `RemoveBlockProposer` functions are defined (ref) and they are used in `BeginBlocker` and `EndBlocker` of a Heimdall-v2 app (ref).

When comparing Heimdall v1 and v2 versions of `BeginBlocker` , there are few parameter and following logic changes that are causing **reward distribution malfunction**.

Because `req abci.RequestBeginBlock` parameter is no longer present, `GetBlockProposer` function is used to retrieve current block proposer from the app's KVStore. Problem is that nowhere before the call of the `GetBlockProposer` current block proposers address is being written to the KVStore, thus `GetBlockProposer` will always return `false` in place of its second return parameter causing the logic for setting block proposers address in `BeginBlocker` to always be skipped which then also affects `EndBlocker` in a way that it will always omit its reward distribution logic.

Since `BeginBlocker` no longer has `req` parameter, logic will have to be moved from the `BeginBlocker` to the `PreBlocker` where proposers address can be directly acquired from parameters (ref) and set to the KVStore.

Also, logic in the v2 `BeginBlocker` after retrieving proposers address (ref) appears redundant as it converts proposers address from `sdk.AccAddress` to `string` and then back to `sdk.AccAddress`. If this logic is written purely for the

sake of address verification that is done during type conversions (ref1, ref2, ref3), there should be more optimal way because in this case it is unnecessarily done twice.

Property AUTH-8: Account Data Integrity

Category: Safety property

`x/auth` module stores account information correctly.

Violation consequences

- Corruption or inconsistency in account information storage/retrieval could impact system operations

Threats

- Invalid account information are not rejected and impact proper account management

Conclusion

The property holds.

Accounts are stored as normalized account addresses (see also *Property AUTH-5: Account Verification*). This includes module accounts, base accounts and their derivates e.g. vesting accounts. Compared to Cosmos-SDK v0.50.12, only the content of account addresses have changed. Addresses for base- and module-accounts are validated on account creation `VerifyAddressFormat()`.

It is noted that unit test `DeterministicTestSuite.TestGRPCQueryAccount()` is passing and can be re-enabled.

Property BANK-1: Enabling bank send per denom is not supported

Category: Safety property

`x/bank` module does not support enabling bank send on a per denom basis.

Threats

- Transactions with `MsgSetSendEnabled` are not rejected.

Conclusion

The property holds.

Because function `SetSendEnabled` that has `MsgSetSendEnabled` as a parameter needs to go through governance proposal process (expects authority that is initially set to gov module account) `MsgSetSendEnabled` isn't accepted during proposal submission by the `ValidateGovMsgType` (ref) function.

Other parts related to `x/bank` parameters were examined (genesis, migrations), revealing that `DefaultParams` (ref) are configured with `SendEnabled` set to `nil` and `DefaultSendEnabled` set to `true`, meaning all denominations are sendable by default. There are no built-in restrictions during genesis to allow only `POL` denomination, and governance could potentially update `DefaultSendEnabled` from `false` to `true` (and vice versa), enabling/disabling all denominations to be sent.

Property BANK-2: Delegation / Undelegation actions are rejected

Category: Safety property

`x/bank` module rejects stake all actions related to delegation functionality.

Violation consequences

- Not rejecting delegation messages could lead to unwanted side effects as related logic is handled in PoS core contracts.

Threats

- Transactions related to delegation are not rejected

Conclusion

The property holds.

`x/bank` `Keeper` interface defines functions for a coin (un)delegations (ref1, ref2), but each of earlier implementations is commented out and returns error instead as soon it is called (ref1, ref2, ref3, ref4), with no alternative implementations present in the `x/bank` module.

Property GOV-1: Deposit Management Safety

Findings: Governance Refund of Deposits, Governance Expedited Proposal Deposits

Category: Safety property

`x/gov` module must ensure that deposits are correctly refunded to depositors in valid scenarios (proposal passes/fails, never enters voting, cancellation, etc).

Violation consequences

- Misbehaviour of refund functionality could lead to drop of liquidity and impact system operations and liveness.

Threats

- Governance refunds deposits incorrectly (multiple times, not at all, partially, to wrong destination).
- Refunded deposits are not removed from the state.
- Governance does not refund deposits in a deterministic way.

Conclusion

The property does not hold.

Deposits are refunded to depositors on proposal which are passing or cancelled. Possible drain of deposits can occur in the context of proposal cancellation where the default value (empty string) of the related parameter `proposal_cancel_dest` is used. In that case the cancellation ratio amount stays in the balance of the governance module and is not refunded.

Deposits are not refunded to its depositors but equally shared across validators in the case of "dead" or rejected proposals. This does not hold in case of an invalid signer address received received from external iterator across the validator set. In this situation the deposits would just be distributed among a subset of the current validator set. It is suggested to just ignore the validator with the invalid address and continue iteration. In `DistributeAndDeleteDeposits()` the shared part is calculated on a per deposit base whereas calculation on the total amount of deposits would benefit of performance improvement and reduce the amount of the factional part.

Deposits are removed from the state whenever a refund is performed.

There is a potential threat related to non-determinism in the case of `IterateCurrentValidatorsAndApplyFn()` not iterating validators in a deterministic way. This is an external module dependency to the staking module in `DistributeAndDeleteDeposits()` which could lead to a chain halt in case of non-deterministic iteration of validators. The chain halt is caused by sending remaining amount of deposits to a random validator due to the random order of the list of addresses of validators. In current implementation of heimdall's staking module the implementation of this iterator function is deterministic. Nevertheless it is suggested to enforce determinism on `x/gov` side for that part.

A random number generator is used to select the validator which is receiving the amount of remaining deposits. The seed use for this randomness is the amount of validators which does not vary too much and might be replaced by a simpler approach or providing more random distribution as a result.

Property GOV-2: Validator Authorization

Category: Safety property

`x/gov` module ensures that only valid validators are allowed to participated in governance (vote on proposals)

Violation consequences

- Unauthorized user can submit votes and influence proposal results. Authorized participants are rejected and cannot vote.

Threats

- Governance accepts votes from unauthorized users (users not part of the active validator set when tallying).
- Governance rejects votes from authorized participants (active validators when tallying).

Conclusion

The property holds.

Every valid user can submit votes on a proposal. No impersonation on the level of submitting a vote is possible as the voter is the one signing the transaction.

Only votes from validators provided by the staking module are taken into account when tallying. Staking module needs to ensure to provide the active validator set at the time when tallying is taking place.

Property GOV-3: User Authorization

Category: Safety property

`x/gov` module accepts proposals and deposits only from valid users (incl. validators).

Violation consequences

- Invalid authorization checks can prevent submission of proposals and deposits.

Threats

- Governance accepts unauthorized proposal submission or deposits
- Authorized users are prevented to submit proposals and deposits

Conclusion

The property holds.

Any user with a valid account can submit governance proposals and add deposit on them. Users without a valid account are not authorized and prevented to submit proposals and deposits.

Property GOV-4: Vote Integrity

Category: Safety property

`x/gov` module ensures that votes are counted accurately and only once per validator

Violation consequences

- Incorrect tally results impacting system operations.

Threats

- Duplicate votes are not rejected by governance module.
- Votes are not counted correctly in tally.

Conclusion

The property holds.

Votes are stored in a map with key pair of proposal ID and address of the voter. No duplication of votes possible due to data structure used. Sending multiple votes for a proposal from same voter results in an update of the existing vote.

Tally implementation panics in case of vote splitting or weighted votes. Safe guards are put in place when votes are added. It might be considered just to ignore the invalid votes as this is in `EndBlocker` and would cause a chain halt.

Property GOV-5: Vote Validity

Category: Safety property

`x/gov` module accepts only valid votes. A valid vote in heimdall does not have more than one vote option and its weight is required to be 1. Vote option needs to be one of the defined options (`Yes`, `Abstain`, `No`, `NoWithVeto`). It needs to be submitted during voting period of the related proposal and meta data should not exceed its configured maximum size.

Violation consequences

- Incorrect tally results impacting system operations.

Threats

- Invalid votes are not rejected.
- Valid votes are rejected.

Conclusion

The property holds.

Votes are validated in the keeper implementation of the governance module. Invalid votes are rejected and not added to the store.

Property GOV-6: Accurate Usage Of Voting Power

Findings: Governance Tally Results

Category: Safety property

`x/gov` module ensures that total voting power is calculated deterministically and accurately when tallying votes for a proposal. The total voting power is the sum of the bonded tokens of the active validators voting for a specific proposal

Violation consequences

- Voting power of validators impacts tally functionality and can lead to wrong tally results.

Threats

- Voting power of validators is not used correctly and causes incorrect tally results.

Conclusion

Property does not hold.

Total voting power is determined from the set of validators provided by the custom staking module of heimdall, which is the active validator set. When iterating of the validator set in the tally function, an error when decoding the validator address would stop the iteration at that point and omitting the remaining validators to be taken into account. This could be an issue never determining the correct total voting power and reaching quorum by just considering a subset of the current active validator set. The effect would be that this subset of validators, which can be even a single one, can control with its vote the outcome of the proposal as quorum is always reached.

Property GOV-7: Proposal Validity

Category: Safety property

`x/gov` module ensures that only valid proposals are accepted. A proposal is valid if the title and summary are present and the defined maximum length for meta data, summary and title of a proposal are not exceeded. The initial deposit must be provided and all messages of a proposal needs to have a handler registered and the governance module being the only signer. Only whitelisted message types as specified in `ValidateGovMsgType()` and`ValidateGovMsgContentType()` are accepted and each message needs to pass its related basic validation checks.

Violation consequences

- Invalid proposals accepted by governance or rejected valid proposals can impact system operations and liveness.

Threats

- Proposal submission accepts invalid proposals (min deposit on submission, messages of type `software-ugrade` , `cancel-sofware-upgrade` , `community-pool-spend` ,...)

Conclusion

The property holds.

Validation of submitted proposals are preformed in `SubmitProposal()` of message server and module keeper. Beside the existing SDK checks for initial deposit, meta data, summary and title, all messages of a proposals are checked against the whitelisted message types in `ValidateGovMsgType()`. Proposals with unsupported legacy proposal messages are rejected by `ValidateGovMsgContentType()`.

`ValidateGovMsgContentType()`checks validity based on `TypeUrl` of the message content. In the implementation we suggest to add a check for nil pointer dereference of `msg.Content`.

Property GOV-8: Authorization of Parameter Updates

Category: Safety property

Changes to governance parameters must follow proper authorization. All messages for parameter updates to be controlled by governance must contain governance as the only signer of the message.

Violation consequences

- Insufficient authorization measures of parameter changes can impact system operations and liveness.

Threats

- Unauthorized users can perform parameter changes.
- Authorized users are prevented to perform parameter changes.

Conclusion

The property holds.

`SubmitProposal()` checks for each message of a proposal that the only signer is the governance module.

The message handlers for parameter updates (`UpdateParams()`) in heimdall's `chainmanager`, `bor`, `checkpoint` modules checks that the authority of the related update message matches the authority set in the keeper of the module.

For the custom modules in heimdall the sanity check for `authority` of the keeper object ensures that no other addresses than the one of the governance module can be set in the keeper. However there's a possibility for a glitch in this check to accept all possible variants of governance module address e.g. `0x1beef`, `0xBEEF`, `0xBeef`, `beef` as it's using the result of `StringToBytes` for the comparison but storing the passed value (a mentioned variant) of it in the module keeper's authority (see `NewKeeper()` of all custom modules in heimdall). The authority check of a specific message run in `UpdateParameters()` might fail as it requires an exact match of the governance modules address returned by the auth module `authtypes.NewModuleAddress(govtypes.ModuleName).String()` . This is not an issue in heimdall's current implementation as its custom modules are initializing their keepers with the exact match of the governance module address.

Property GOV-9: Validation of Parameter Updates

Category: Safety property

Only valid parameter updates are accepted by governance. Valid parameters are those defined in update messages for cosmos' `governance`, `auth`, `bank` and `consensus` implementation and heimdall's `bor`, `chainmanager`, `checkpoint`, `clerk`, `topup`, `stake` and milestone module.

Violation consequences

- Insufficient validation of parameter updates can impacting system operations and liveness.

Threats

- Invalid parameters of governance proposals are not rejected.
- Governance rejects proposals with valid parameter updates.
- Valid parameters are not accessible and can't be updated on proposal execution.

Conclusion

The property holds.

On governance proposal submission the whitelisted parameter update messages are used to ensure that only these parameters are subject of an update. To be able to execute the parameter update the related message handler for the parameters needs to be registered in the base application which is outside the scope of this audit. The governance module ensures that on proposal submission a handler for each message of the proposal is registered.

It is noted that `staking`, `clerk`, and `topup` module are fully relying on cosmos' `x/params` module for parameter updates which is deprecated.

Threat Model (Heimdall v2 ABCI++ implementation)

Definitions

We say that a message `MsgXYZ` (e.g., `MsgStakeUpdate`) is ***executed*** when its corresponding post handler (e.g., `PostHandleMsgXYZ`) is called (irrespectively of whether the post handler returns `nil` or not).

We call the **voting** of a message, the execution of its side handler (e.g., `SideHandleMsgXYZ` for message `MsgXYZ`). The result of voting by a validator can either be `Vote_VOTE_YES` or `Vote_VOTE_NO`.

We say that a message `MsgXYZ` is ***voted-upon*** if $>2/3$ of the validators (i.e., talking about their voting power from the previous block) voted `Vote_VOTE_YES` for this message.

We call the **verification** of a message, the execution of its message handler. We say that a message is **successful** if its verification is successful (i.e., its message handler does not return an error whenever it is called from (e.g., during `ProcessProposal`, `PrepareProposal`, or `FinalizeBlock`). Otherwise we say the message is **failed**.

The following messages from each module are called **side** messages:

`x/bor`:

- `MsgProposeSpan`

`x/checkpoint`:

- `MsgCheckpoint`
- `MsgCpAck`

`x/clerk`:

- `MsgEventRecord`

`x/stake`:

- `MsgValidatorJoin`
- `MsgStakeUpdate`
- `MsgSignerUpdate`
- `MsgValidatorExit`

`x/topup`:

- `MsgTopupTx`

All other messages are non-side messages. In what follows, unless stated otherwise, we refer to side messages.

Property ABCI-01: Only Voted-upon Side Messages Can Be Executed

Category: Safety property

Threats

- A side message executed without being voted-upon can lead to cases where someone sets bogus voting powers (e.g., by calling `MsgStakeUpdate` with bogus data). Or a failed message (that passed initial verification but later on due to state changes fails execution in `FinalizeBlock`) is voted upon and it gets executed. This could lead to having a message getting executed although it was not verified.

Conclusion

The analysis focused on ensuring that only side messages that have been properly voted upon and successfully verified are ultimately executed through their respective post handlers. This property was evaluated across the full transaction lifecycle — spanning `ExtendVote`, `PrepareProposal`, `ProcessProposal`, and `FinalizeBlock` (via `PreBlocker`).

Execution Control Mechanism

- Voting Phase (`ExtendVote`):

  Each validator executes the side handler for each side message, resulting in a vote (`VOTE_YES`, `VOTE_NO`, or `UNSPECIFIED`). These votes are included in the `VoteExtension` for the current block.

- Aggregation and Approval (`PreBlocker`):

  Votes are aggregated in `tallyVotes`, which determines whether a transaction is approved (>2/3 voting power voted `VOTE_YES`). Only transactions included in `approvedTxsMap` are forwarded for post-handler execution.

- Execution Phase (`PreBlocker`):

  The `PostHandler` of a side message is called only if its transaction is present in `approvedTxsMap`. Inside each `PostHandler`, an additional guard (`sideTxResult != Vote_VOTE_YES`) prevents execution if the vote was not affirmative.

This layered mechanism ensures that execution is strictly limited to side messages with approval.

Verification Completeness

- Verification Phase (`PrepareProposal`, `ProcessProposal`, `FinalizeBlock`):

  The message handler is invoked via `runTx` in `app.Prepare(Process)ProposalVerifyTx()` or `FinalizeBlock`, ensuring that the message passes standard semantic validation (e.g., checking nonce, stake amount, replay protection). A failed message (i.e., one that returns an error during verification) is excluded from the final committed block unless the proposer is malicious.

- `PostHandler` Isolation:

  The `PostHandler` does not repeat full verification and assumes the message has been previously validated. This implies a dependency on the earlier verification phase not being bypassed — a key point in validating the execute-only-successful property.

The implementation satisfies the property under normal conditions.

All relevant `PostHandlers` respect the voting outcome before execution. However, message verification happens earlier, during `PrepareProposal`/`ProcessProposal`, and is not completely rechecked at `FinalizeBlock`. This is acceptable if state changes do not invalidate the earlier verification, which is found to be true.

Property ABCI-02: The Vote Extensions (in LocalLastCommit) of a Committed Block Need to Have Valid Signatures (i.e., Signed by the Respective Validator)

Category: Safety property

Threats

- Someone could introduce bogus vote extensions and take over Heimdall.

Conclusion

The signature verification logic was analyzed with respect to its enforcement across both `ValidateVoteExtensions` and `ValidateNonRpVoteExtensions`. These functions are invoked respectively within `PrepareProposal` (executed by the proposer) and `ProcessProposal` (executed by all validators), ensuring that signature verification occurs at multiple points in the consensus.

The following aspects were examined in detail:

- Signature Source and Validation:
  - The validator's public key is retrieved from the validator set (via `getPreviousBlockValidatorSet`).
  - The `ExtensionSignature` field in the vote is verified against canonical sign bytes (which include the height, round, and chain ID).
  - This process ensures that each vote extension is provably linked to the expected validator identity.
- **Standard and Non-RP Vote Extensions**:
  - `ValidateVoteExtensions` covers standard vote extensions attached to transactions.
  - `ValidateNonRpVoteExtensions` performs a similar check for non-RP extensions, including verification of milestone and checkpoint data.
  - Both functions independently validate signatures and reject malformed or unauthorized inputs.
- **Threat Scenarios Considered**:
  - A forged vote extension from an unauthorized entity.
  - A malformed vote extension with manipulated fields.
  - A proposal containing bogus vote extensions introduced by a malicious proposer.

Across all paths, the application rejects vote extensions with invalid signatures before accepting or finalizing blocks. The logic adheres to CometBFT's expectations and ensures that only authorized and correctly signed data influences consensus outcomes. No inconsistencies, bypasses, or failure scenarios were identified in the signature verification mechanism.

Property ABCI-03: The Vote Extensions (in LocalLastCommit) of a Committed Block Cannot Contain the Vote of a Single Validator More Than Once or from Someone That is Not a Validator

Category: Safety property

Threats

- A validator could introduce multiple votes to vote on messages that are not supposed to pass.

Conclusion

The implementation enforces at-most-once voting by:

- Ensuring each validator appears only once in `LocalLastCommit`.
- Verifying that a validator votes only once per side transaction.
- Confirming all votes originate from known validators from the previous block.

These checks are performed during both `PrepareProposal` and `ProcessProposal` phases. The vote aggregation and validation logic (`aggregateVotes`, `validateSideTxResponses`) are consistent and sufficient to prevent amplification or injection attacks. No violations of the at-most-once voting property were identified.

The at-most-once voting property was assessed by analyzing how vote extensions are processed and validated during the `PrepareProposal` and `ProcessProposal` phases. The analysis focused on two key threat scenarios:

1. Double Voting by a Validator:
   - This could occur if a validator:
     - Submits multiple votes within a single vote extension (i.e., multiple `SideTxResponses` referencing the same `TxHash`).

  - Appears more than once in `LocalLastCommit`.
- The application defends against both:
  - In `ValidateVoteExtension`, the function `validateSideTxResponses` maintains a `txVoteMap` to detect if a validator has already voted for a specific transaction.
  - A `seenValidators` map ensures each validator contributes only once to `LocalLastCommit`.
  - These checks are applied consistently in both `PrepareProposal` and `ProcessProposal`, guaranteeing that no validator can influence the outcome more than once per block.

2. Votes from Non-Validators:
   - An explicit validation step ensures that all entries in `LocalLastCommit` originate from recognized validators of the previous block.
   - In `ValidateVoteExtension`, this is enforced by checking the address against the validator set.
   - A similar check exists in `ValidateNonRpVoteExtension`

No inconsistencies or loopholes were found that would allow unauthorized votes or multiple votes from the same validator. The design adheres to the expected CometBFT semantics and properly enforces at-most-once voting across all voting paths.

Property ABCI-04: A Transaction in a Committed Block Cannot Contain More Than One Message That Has a Side Handler

Category: Safety property

Threats

- A malicious validator manages to commit a block that we are not able to execute.

Conclusion

The implementation enforces that each transaction contains at most one side-handled message by:

- Filtering such transactions out during `PrepareProposal`.
- Explicitly rejecting proposals that include them during `ProcessProposal`.

This prevents ambiguous or unexecutable transactions from entering the committed block, preserving consensus safety. No violations of this property were found in the current design.

Property ABCI-05: If Events e_1, e_2, and e_3 Take Place on Ethereum (or Bor), Then Heimdall Executes All the Corresponding Messages in This Exact Same Order

Category: Safety property

Conclusion

Looking into this property, we realize that some properties need to respect ordering such as `MsgStakeUpdate`, while others do not (e.g., `MsgTopupTx`). As we see in the following table, the only side messages that need to respect ordering are the `MsgStakeUpdate`, `MsgSignerUpdate`, and `MsgValidatorExit` and indeed they respect it. Hence, the property holds.

Message (Module)	Do we need to respect order and not miss events?	Can it be re-ordered? (e.g., have events e_1 and e_2 and Heimdall executes message for e_2 and then for e_1)	Can it be missed? (e.g., have events e_1, e_2, and e_3 and we only see e_1 and e_3)
`MsgProposeSpan` (`x/bor`)	**No**.		
`MsgCheckpoint` (`x/checkout`)	**No**.		
`MsgCpAck` (`x/checkout`)	**No**.		
`MsgEventRecord` (`x/clerk`)	**No**.		
`MsgValidatorJoin` (`x/stake`)	**No.**		
`MsgStakeUpdate` (`x/stake`)	**Yes.**	**No**. Has checkpoints. Assuming each event has a specific `Nonce` and due to the `msg.Nonce != validator.Nonce + 1` this message cannot be reordered.	**No.** Due to the use of `Nonce`.
`MsgSignerUpdate` (`x/stake`)	**Yes**. A signer probably changes key for security reason so we do not want to keep using the old one.	**No.** Uses `Nonce`.	**No.** Due to the use of `Nonce`.
`MsgValidatorExit` (`x/stake`)	**Yes**.	**No**. Uses `Nonce`.	**No.** Due to the use of `Nonce`.
`MsgTopupTx` (`x/topup`)	**No**.		

Property ABCI-06: A Block Proposed by a Correct Validator Only Contains Transactions that Have Only Successful Messages

Category: Safety property

Conclusion

This property is trivially satisfied because during `PrepareProposal` for a transaction to be added to a block, the transaction has to be executed (as part of `PrepareProposalVerifyTx` that calls `runTx` and subsequently `runMsgs`).

Property ABCI-07: A Correct Validator Rejects a Block During ProcessProposal that Contains At Least One Failed Message

Category: Safety property

Conclusion

This property is trivially satisfied because during `ProcessProposal` a transaction is only added to a block, if the transaction can be executed as part of `ProcessProposalVerifyTx` that calls `runTx` and subsequently `runMsgs`.

However, this property raises an interesting question: What happens if a message is successful during verification (i.e., message handler succeeds) but later during the execution (i.e., post handler execution) if we were to execute the verification again, the verification fails? This can happen because `ProcessProposalVerifyTx` is executed on a clean slate every time without seeing the side effects of the other transactions in the same block. For example, consider a block `B` at height `H + 1` with two transactions t_1 and t_2 with side messages where both t_1 and t_2 pass `ProcessProposalVerifyTx` because they run starting from the state up to height `H`. Additionally consider that both t_1 and t_2 are voted upon by $> 2/3$ of validators. However, when we actually execute those transactions, transaction t_2 executes after transaction t_1 has completed and hence t_2 executes in a potentially different state than the one it used when running `ProcessProposalVerifyTx`. In such a scenario, t_2 executes its post handler although it should not have. This is what we investigate below for each side message.

Note that the message handler of a message is executed only once and its post handler is executed in the next block as part of the `PreBlocker` but for the last block's transactions.

In the following table, we go through every side message and check whether the aforementioned scenario is possible.

Message (Module)	verification (i.e., message handler)	voting (i.e., side handler)	execution (i.e., post handler)
`MsgProposeSpan` (`x/bor`)	Checks that `lastSpan.Id+1 != msg.SpanId` so if two messages pass verification at the same block they need to have the exact same `msg.SpanId`, as well as `msg.ChainId`.	Checks in relation to externally fetched `currentBlock` as well as `SpanId`.	Checks `SpanId` for replay attacks. Updates `SpanId` as part of `FreezeSet`. Therefore, a second `MsgProposaSpan` in the same block fails execution.
`MsgCheckpoint` (`x/checkout`)	Checks that `lastCheckpoint.EndBlock + 1 != msg.StartBlock`, so if two messages pass verification at the same block they have the exact same `msg.StartBlock`.	Calls `IsValidCheckpoint` that requests data from Bor.	Checks that `lastCheckpoint.EndBlock + 1 != msg.StartBlock` and checks that checkpoint in buffer in which case it fails. If one passes, the other one would fail.
`MsgCpAck` (`x/checkout`)	Checks `msg.StartBlock`, etc. correspond to what is on buffer.	External call to verify message matches contract data.	Verifies in relation to checkpoint buffer and flushes buffer). The checks here are stricter than the verification.
`MsgEventRecord` (`x/clerk`)	Checks whether an event record with this `msg.Id` already exists, as well as sequence (based on `blockNumber` and `logIndex`) and checks `msg.ChainId`.	External call to verify message that includes the `msg.Id`.	Checks message's `Id` but does **not** check the sequence. Although, you cannot overwrite the record though due to the `Id` check.

Message (Module)	verification (i.e., message handler)	voting (i.e., side handler)	execution (i.e., post handler)
`MsgValidatorJoin` (`x/stake`)	Checks sequence (based on `blockNumber` and `logIndex`) as well as whether `msg.ValId` already exists.	Compares `msg` with data received from external calls (e.g., `Nonce`) that also includes `msg.ValId`.	Cannot succeed while verification fails because execution checks sequence as well. `msg.ValId` is not being checked but this is checked as part of voting. Additionally, `AddValidator` sets `msg.ValId` so assuming we cannot have two events with the exact same `ValId` emitted, there is no issue.
`MsgStakeUpdate` (`x/stake`)	Checks sequence (based on `blockNumber` and `logIndex`) . Also checks that `msg.Nonce != validator.Nonce+1`.	Compares the `msg` with data received from external calls including `msg.Nonce`.	Cannot succeed while verification fails because execution checks sequence as well. The `msg.Nonce != validator.Nonce+1` is not checked. However, if we are to assume that all `MsgStakeUpdate` messages that are voted upon have a different `msg.Nonce`, then this implies that this `MsgStakeUpdate` is the exact same message as the one that got executed.
`MsgSignerUpdate` (`x/stake`)	Checks sequence (based on `blockNumber` and `logIndex`) as well as `msg.Nonce != validator.Nonce+1`. Additionally checks that indeed we have a `newSigner`.	Compares the `msg` with data received from external calls including `msg.Nonce`.	Similar to what we have above, if we assume that `Nonce` cannot be re-used, then, this cannot succeed while verification fails because execution checks sequence and that we have a new signer.
`MsgValidatorExit` (`x/stake`)	Checks sequence (based on `blockNumber` and `logIndex`) as well as `msg.Nonce != validator.Nonce+1`.	Compares the `msg` with data received from external calls including `msg.Nonce`.	Again, if we assume that `Nonce` cannot be re-used from the external side, then, cannot succeed while verification fails because execution checks sequence.

Message (Module)	verification (i.e., message handler)	voting (i.e., side handler)	execution (i.e., post handler)
`MsgTopupTx` (`x/topup`)	Checks sequence (based on `blockNumber` and `logIndex`) as well as feasibility of top up and whether the denom is enabled.	Compares the `msg` with data received from external calls (e.g., the fee is the same).	For this to execute successfully, the sequence should not exist already. However, it could be the case that the denom is disabled during execution. For this to happen, a `x/gov` proposal needs to execute at the same block that disables the denom but even then the proposal executes at `EndBlocker` and hence succeeds the `PreBlocker` where the execution takes place. Hence there is no issue.

From the table we can see that if a message is successful during verification (i.e., message handler succeeds) but later during the execution (i.e., post handler execution) if we were to execute the verification again, the verification would not fail.

Recommendation

The crux of the verification for each side message can move to a new method and can then be used by both the message and side handlers, as well as the post handler. This way, it is easy to see that we cannot have a case where a post handler executes even though the verification fails.

Property ABCI-08: The PreBlocker Should Not Fail Due to User's Input (e.g., a Problematic Transaction)

Category: Safety property

`PreBlocker` can potentially fail if the underlying key-value storage is corrupted and we cannot read or set from it, etc. However, we introduce this property because we want to avoid exceptional cases where the chain might halt due to a malicious transaction, etc.

Let us consider all the cases where `PreBlocker` can return an `error`:

- Assuming that `voteExtensionsEnableHeight > 0` and that `req.Height + 1 >= voteExtensionsEnableHeight` then `checkIfVoteExtensionsDisabled` and `req.Height` check do not throw an error.
- Because of the way we process blocks, `req.Txs[0]` contains `abci.ExtendedCommitInfo` so an error is not thrown here and here. Additionally, note that because `req. Height + 1 >= voteExtensionsEnableHeight` this implies that vote extensions are included (`req.Txs[0]` is populated during `PrepareProposal`):

  > *Otherwise, at all heights greater than the configured height `H` vote extensions must be present (even if empty). When the configured height `H` is reached, `PrepareProposal` will not include vote extensions yet, but `ExtendVote` and `VerifyVoteExtension` will be called. `PrepareProposal` will include the vote extensions from height `H`.*

- `GetLastBlockTxs` does not throw an error because last block transactions are set during genesis and again set here.

- getPreviousBlockValidatorSetdoes not throw an error because the previous block validator set is set during genesis and is also updated at the end of every block. Whether that validator set can be empty, etc. is out of scope of this audit (is part of the `x/stake` module and the corresponding Ethereum contracts).
- HasMilestone performs basic key-value store checks and GetLastMilestone is called only if `hasMilestone` evaluates to `true`, so `GetLastMilestone` finds a milestone and does not return an `ErrNoMilestoneFound` error.
- We investigate whether GetMajorityMilestoneProposition can throw an error. This can happen in the following cases:
  - Fails to Unmarshal vote extension. This cannot happen because we already check this in ProcessProposal as part of ValidateVoteExtensions.
  - Fails to transform val address to string. This cannot happen because we already check this in ProcessProposal as part of ValidateVoteExtensions.
  - Fails because a validator does not belong in thevalidatorSet. We already check this as part of ValidateVoteExtensions. Specifically, note that `validatorSet` stems from the previous block validator set and the previous block validator set is updated at the ApplyAndReturnValidatorSetUpdates that is called from EndBlocker. The `PreBlocker` executes before `EndBlocker` so the check on the `validatorSet` performed in `ValidateVoteExtensions` applies here as well.
  - Fails because the supportingValidatorList is empty. However, if `supportingValidatorList` was empty then totalSupportingPower is 0 and `GetMajorityMilestoneProposition` returns without an `error` here.
- AddMilestone and SetMilestoneBlockNumber can throw an error. However, they both perform basic read/write on the key-value storage, so unless storage is corrupted everything operates as expected.
- tallyVotes can throw an error here in the following cases:
  - In aggregateVotes, but for `aggregateVotes` to return an `error` one of the following things needs to happen:
    - The vote does not have a valid BlockIdFlag but this cannot happen because this is checked in ValidateVoteExtensions during `ProcessProposal`. Similarly, the Unmarshal of vote extensions is checked in ValidateVoteExtensions.
    - `ve.Height != currentHeight - 1` cannot occur because it is checked in ValidateVoteExtensions in `ProcessProposal`.
    - The block hashes are not equal. As a matter of fact this can happen if a malicious validator adds a bogus vote extensions during proposal (see findings).
    - The validator address to string transformation (as well as here) cannot fail because we already check in ValidateVoteExtensions in `ProcessProposal`.
    - We have a duplicate vote. Again, this is already checked in ValidateVoteExtensions.
    - We have multiple votes and whether a vote is valid are already checked in here validateSideTxResponses in ValidateVoteExtensions.
  - Voting power exceeds total voting power. A single validator cannot vote differently for a transaction due to the multiple-votes check. Hence a specific `txHash` gets at most `vote.Validator.Power` per validator. Additionally, because we check that no validator votes twice (in all of the vote extensions), the power sum cannot exceed `totalVotingPower`.
- getMajorityNonRpVoteExtension can throw an error:
  - If it cannot transform the validator address to string. But this cannot happen because we already check in ValidateVoteExtensions in `ProcessProposal`
  - If it cannot find a validator. Cannot happen because we check that the validator exists in ValidateVoteExtensions (see argument above for`GetMajorityMilestoneProposition`).
- SetCheckpointSignaturesTxHash or SetCheckpointSignatures can throw an error but these perform basic read/write on the key-value storage, so unless storage is corrupted everything should operated as expected.
- Decoding cannot fail because this is already checked during `ProcessProposal` in ValidateVoteExtensions.

Conclusion

From the above we can see that there is a case where `PreBlocker` can return an `error`, that when a malicious validator adds a bogus vote extension during proposal (see findings).

Property ABCI-09: A Voted-upon Side (except MsgCheckpoint) Message Eventually Executes

Category: Liveness property

This means, that there should be no way for a malicious actor to prevent a message (e.g., `MsgStakeUpdate`) from being executed.

Conclusion

Consider a successful side message `m`, `m` eventually gets picked up by a correct validator, added to a block and voted upon. This means that eventually the transaction `t` of this message reaches this part of the `PreBlocker` where it is about to execute its post handler. The only way for message `m` to not execute is if:

- `m` has no corresponding post handler: This is not the case because all side messages have a corresponding post handler, `MsgProposeSpan`, `MsgCheckpoint` and `MsgCpAck`, `MsgEventRecord`, `MsgValidatorJoin`, `MsgStakeUpdate`, `MsgSignerUpdate`, and `MsgValidatorExit`, and `MsgTopupTx`.
- `t` contains at least 2 messages with post handlers (where `m` appears later in `t`) and hence exits early here and does not execute message `m`: This cannot happen because i) in `ProcessProposal` we check that a transaction that does not have more than 1 side handler, and ii) a message has the exact same number of post handlers as side handlers. The second case can occur only if for a message we do not have a side handler but we do have a post handler in which case `sideHandler == nil` and `postHandler != nil`. This second case cannot occur because there is no such side message, all side messages have set up a corresponding side handler (`MsgProposeSpan`, `MsgCheckpoint` and `MsgCpAck`, `MsgEventRecord`, `MsgValidatorJoin`, `MsgStakeUpdate`, `MsgSignerUpdate`, and `MsgValidatorExit`, and `MsgTopupTx`), as well as a post handler (see above).

Recommendation

To guarantee this is the case in the future as well in case new messages are added, an additional check can be introduced here for the registration to fail if `(sideHandler == nil || postHandler == nil)`.

Property ABCI-10 (milestones): Only Voted Milestones Get Added

Category: Safety property

`AddMilestone` is called for a milestone `m` with `m.StartBlock` and `m.EndBlock` only if at $> 2/3$ of the validators support this milestone, meaning that at $> 2/3$ validators have the same Bor block hash for all the block heights from `m.StartBlock` to `m.EndBlock`.

Conclusion

`AddMilestone` is called only here if `isValidMilestone` evaluates to `true`. This can only happen if the `ValidateMilestoneProposition` is successful for the given `majorityMilestone` returned by `GetMajorityMilestoneProposition`. In `GetMajorityMilestoneProposition` the `blockHashes` from `startBlock` to `endBlock` are returned where all those blocks correspond to `majorityBlocks`. For a block to be in `majorityBlocks`, the blocks needs to have been voted upon by at $> 2/3$ of the validators and hence the property holds.

Property ABCI-11 (milestones): Milestones are Contiguous

Category: Safety property

`AddMilestone` is only called for a milestone `m` with `m.StartBlock` if for the last milestone `m'` we have `m'.EndBlock = m.StartBlock - 1`.

Conclusion

We propose a milestone in `ExtendVoteHanlder` using `GenMilestoneProposition`. `GenMilestoneProposition` generates the next milestone proposition based on the last milestone's `EndBlock`. Nevertheless, a malicious validator can potentially generate a bogus milestone proposition that does not satisfy this milestone-contiguity property. However, even in this case, when `GetMajorityMilestoneProposition` is called before `AddMilestone` it gets as a parameter the `EndBlock` of the last milestone, so `GetMajorityMilestoneProposition` creates a majority proposition starting from `EndBlock + 1` and hence the property is satisfied.

Property ABCI-12 (milestones): Milestones Eventually Get Added

Category: Liveness property

Milestones are periodically introduced and are added (i.e., `AddMilestone` is called).

Conclusion

For this property to hold, besides the given liveness assumptions, we have to assume that no fork has taken place on Bor (from the perspective of correct validators). In this case, eventually >2/3 of validators propose at least one block hash during`GenMilestoneProposition` and then the longest possible prefix is being picked up by`GetMajorityMilestoneProposition` that is used when `AddMilestone` is called.

Property ABCI-13 (checkpoint): Checkpoint tx Stems from a Correct Process

Category: Safety property

The method `getMajorityNonRpVoteExtension` always returns the non-replay-protected vote extension of a correct validator. Additionally, if `getMajorityNonRpVoteExtension` returns a non-`error`, the returned `[]byte` slice should contain at least 1 element otherwise.

Conclusion

- If `getMajorityNonRpVoteExtension` return a non-`error`, the returned `[]byte` slice contains at least one element. `getDummyNonRpVoteExtension` always returns a `[]byte` slice with at least one element. So, if there is no `MsgCheckpoint` in a block, there is at least one `nonRpVoteExt` (the dummy one) that is voted upon and is returned by `getMajorityNonRpVoteExtension`. Note that in case of a correct `nonRpVoteExt`extension, `packExtensionWithVote` always returns an extension with at least one byte.
- The part of the property that states that "*always returns the non-replay-protected vote extension of a correct validator*" does not hold. This is because you can have a vote extension returned by `getMajorityNonRpVoteExtension` that does not correspond to any correct validator. For example, this can happen in the following scenario:
  - `2f + 1` validators vote `pre-commit` for the same block;
  - `f` of those validators are malicious and choose a different (than the last) non-replay-protected vote extension;

– remaining `f+1` validators are correct but some of them might timeout when requesting from Bor (see here) and hence their side handlers vote `Vote_VOTE_NO`.

As a result, the vote extension returned by `getMajorityNonRpVoteExtension` can stem from a malicious validator.

Nevertheless, in hindsight, this is not an issue because at the end, the checkpoint that is executed (i.e., calls `CheckpointSignatures`, etc.) is one that has to be approved by at least `2f + 1` validators and hence the `MsgCheckpoint` that is getting executed is a successful one.

- We need to verify that there is no case where, for example, a bogus vote is being added by a malicious validator that forces `getMajorityNonRpVoteExtension` to not return something (e.g., always return an `error`). This can occur if `getPreviousBlockValidatorSet` returns an `error` or the validator address to string transformation fails or `validatorSet.GetByAddress(valAddr)` returns `nil` for the `validatorSet` returned by `getPreviousBlockValidatorSet`. The validator address to string transformation cannot fail because of `ProcessProposal` as part of `ValidateVoteExtensions`. Furthermore, `getPreviousBlockValidatorSet` returns the previous validator set, that is updated in `ApplyAndReturnValidatorSetUpdates` during `EndBlocker` similarly to what is depicted in the figure below:

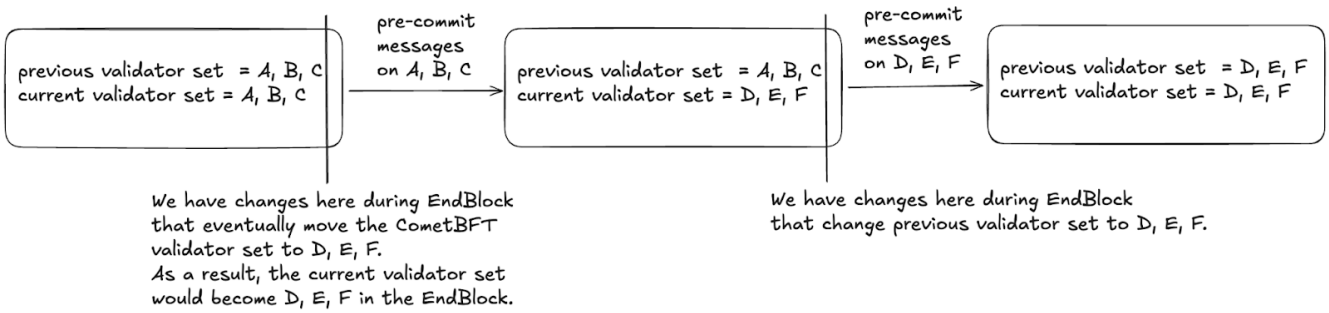

Figure 3: Validator set change

Note in the figure that the CometBFT validator set (used in `pre-commit` messages) takes two blocks to be applied as described here:

> *Note the updates returned in block `H` will only take effect at block `H+2`.*

As a result, when `getPreviousBlockValidatorSet` is called as part of `getMajorityNonRpVoteExtension` during the `PreBlocker` the previous or current validator sets have not been yet updated, so `validatorSet.GetByAddress(valAddr)` cannot return `nil`. Also, note that for `getPreviousBlockValidatorSet` to return an `error`, this implies that `ValidateVoteExtensions` failed before the `PreBlocker`call, hence the chain would have halted in the previous block (because no block would have been able to be proposed or processed).

Property ABCI-14 (checkpoint): Checkpoints are Contiguous

Category: Safety property

The verification of a `MsgCheckpoint` message `m` should guarantee that `m.StartBlock` starts from the previously stored-in-state checkpoint.

Conclusion

Checkpoints are indeed contiguous due to this check here that `lastCheckpoint.EndBlock+1 != msg.StartBlock`. Therefore, any `MsgCheckpoint` message that starts from a different block number fails in `ProcessProposalVerifyTx` and is not included in any block.

Property ABCI-15 (checkpoint): Checkpoints Eventually are Applied

Category: Liveness property

There should always be a way to submit a `MsgCheckpoint` that gets voted upon and is executed that leads to updating the `CheckpointSignaturesTxHash` and `CheckpointSignatures`.

Conclusion

Consider a successful `MsgCheckpoint` (i.e., it passes verification) as well as it is getting voted upon. Is there a way to prevent this `MsgCheckpoint` from being executed? Indeed, there is, because only the last `MsgCheckpoint` is set here in `nonRpVoteExt` and hence `findCheckpointTx` returns that last transaction that contains this message. Therefore, a `MsgCheckpoint` that for example resides at the beginning of a block with multiple `MsgCheckpoints` might **not** execute.

In any case, a `MsgCheckpoint` can only be successful if it has the right `proposer`. So, in a block, only `MsgCheckpoints` from a single validator can be successful. Eventually, a correct validator proposes a valid `MsgCheckpoint`, this message gets voted upon and is eventually executed. So, even though messages might be skipped, a `MsgCheckpoint` is eventually executed.

Property ABCI-16 (CometBFT): Requirements for the Application

R1 [PrepareProposal, timeliness]

`PrepareProposal` does not take too much time

Conclusion

The `PrepareProposal` logic includes several loops over validators, vote extensions, and transactions. The overall time complexity can be expressed as $\mathbf{O(V \times S + V + T \times M)}$, where:

- $\mathbf{V}$ = number of validators,
- $\mathbf{S}$ = number of side transactions (vote extensions) per validator,
- $\mathbf{T}$ = number of transactions in the block proposal,
- $\mathbf{M}$ = number of messages per transaction.

This complexity is manageable under typical network conditions, especially when the number of validators and messages is within expected operational limits. Time is primarily dependent on the number of vote extensions and total transactions being proposed.

During the inspection of the threat model, a finding was uncovered that demonstrates a scenario where a malicious validator can include an arbitrary number of `SideTxResponses` within vote extensions. This behavior may inflate the computation time beyond expected bounds and violates the timeliness requirement.

R2 [PrepareProposal, tx-size]

Prepare proposal does not take too much space.

Conclusion

The total size of the proposal is bounded by the `MaxTxBytes` parameter, which caps the size of included transactions (`Sum(Tx_i)`). However, the first transaction — the marshaled `LocalLastCommit` — is not size-limited, and it includes vote extensions and validator metadata. The size of the proposal can be approximated as:

`Size(PrepareProposal) = V × S + T × M`, where:

- `V` = number of validators,
- `S` = average size of a single validator's vote extension (including side tx responses and signatures),
- `T` = number of included transactions,

- `M` = average transaction size.

While `T × M` is bounded by `MaxTxBytes`, the `V × S` component can grow without strict limits, especially if vote extensions include a large number of side transaction responses.

During threat model inspection, a finding was identified where a malicious validator could inject an arbitrary number of side transaction responses into vote extensions. This violates this requirement.

R3 [`PrepareProposal`, `ProcessProposal`, coherence]

On those blocks prepared by a correct node, any other correct node's ProcessProposal will return true as well.

Conclusion

`ProcessProposal` re-validates all critical conditions established in `PrepareProposal`, including vote extension correctness, transaction filtering, and proposal structure. Although a rejection may occur due to a failed Bor query (external dependency), this is explicitly permitted under CometBFT RFC-105. Thus, the implementation satisfies R3 [coherence] in all deterministic paths and adheres to accepted protocol behavior for non-deterministic inputs.

R4 [`ProcessProposal`, determinism-1]

`ProcessProposal` is a deterministic function of the application's current state and the proposed block.

Conclusion

`ProcessProposal()` is deterministic with respect to the application's state and the block contents, except for a well-defined, external dependency on the Bor contract. This dependency is handled according to CometBFT RFC-105, which explicitly allows a proposal to be rejected if such an external query fails. Thus, the function **satisfies R4 [determinism-1]**, under the accepted semantics of optional external reads.

R5 [`ProcessProposal`, determinism-2]

For any two correct processes, the output of their ProcessProposal on an arbitrary block is equivalent.

Conclusion

The implementation of `ProcessProposal` in Heimdall is equivalent across correct nodes, provided they share the same application state and receive the same proposal. All logic is deterministic and based on local state and block input, except for a well-scoped dependency on Bor (external RPC), which is handled in compliance with CometBFT's RFC-105. Therefore, the implementation satisfies R5 [determinism-2] under the accepted semantics of optional external queries and deterministic state-driven validation.

R6 [`ExtendVote`, `VerifyVoteExtension`, coherence]

A vote extension created by a correct process always passes the VerifyVoteExtension check performed by a correct process.

Conclusion

The implementation ensures that vote extensions produced by a correct validator are accepted by other correct validators. Shared validation functions and explicit checks (height, hash, vote format) uphold this coherence. No divergence risks were found in the current logic, and the requirement R6 [coherence] is satisfied.

R7 [`VerifyVoteExtension`, determinism-1]

Verification depends only on vote extension, proposed block at the height and application's current state.

Conclusion

Every check inside `VerifyVoteExtensionHandler` relies exclusively on:

- The vote extension content.
- The block height and hash (passed in `req`).
- The application state at the time of the vote (accessed through the context).

There is no dependency on external input, ordering, or time-based behavior, and no mutation of state occurs during this phase. The implementation satisfies the determinism requirement. `VerifyVoteExtension` is fully deterministic as per R7.

R8 [`VerifyVoteExtension`, determinism-2]

For any two correct processes, the output of their `VerifyVoteExtension` on an arbitrary vote extension is equivalent.

Conclusion

The `VerifyVoteExtensionHandler`, together with `ValidateNonRpVoteExtension`, performs deterministic verification based on the vote extension input, application state, and external contract data. As long as the nodes:

- Are at the same block height,
- Share the same keeper state,
- And have consistent access to Ethereum contract views,

then the output of `VerifyVoteExtension` is guaranteed to be identical across all correct nodes. The implementation satisfies R8 [determinism-2], assuming synchronized state and consistent contract access across nodes.

R9 [all, no-side-effects]

Execution of `PrepareProposal`, `ProcessProposal`, `ExtendVote`, and `VerifyVoteExtension` do not modify the application's state.

Conclusion

`ExtendVoteHandler()` is designed to be side-effect free. It uses cache-wrapped contexts for side transaction execution, avoids direct store writes, and performs only deterministic vote extension construction and validation. Combined with previous analysis of `PrepareProposal`, `ProcessProposal`, and vote verification functions, the codebase satisfies R9 [no-side-effects].

R10 [`ExtendVote`, `FinalizeBlock`, non-dependency]

For any correct process p, and any vote extension e that p received at height h, the application state at height h does not depend on e.

Conclusion

The implementation satisfies R10 [non-dependency]. Although vote extensions are processed during `FinalizeBlock(h)`, they only affect the application state of height h+1 by influencing the post-processing logic in `PreBlocker()`. No state changes at height `h` depend on vote extensions received at that same height.

R11 [`FinalizeBlock`, determinism-1]

For any correct process the application state exclusively depends on the previous application state and the decided-upon block.

Conclusion

The `PreBlocker` logic used inside `FinalizeBlock` meets the R11 [determinism-1] requirement. The application state transitions are strictly determined by the prior state and the contents of the finalized block. All processing paths, including vote tallying, milestone aggregation, and post handler invocation, rely only on deterministic inputs shared across correct nodes.

R12 [`FinalizeBlock`, determinism-2]

For any correct process the results of the executed transactions from the block being finalized will exclusively depend on the previous state and the block.

Conclusion

All logic within `FinalizeBlock`, including the execution of both regular and side transactions via post-handlers, is deterministic and entirely dependent on the previously committed application state and the contents of the finalized block. No use of external APIs, randomness, or mutable shared state was found. As such, requirement R12 [determinism-2] is satisfied, and no violations or inconsistencies were identified during inspection.

Threat Model (Heimdall v2 custom modules)

Our threat analysis begins by defining a set of properties essential for the correctness of the audited scope - specifically, the custom modules of Heimdall v2 and their correlation with other components in the Polygon protocol. These properties are grouped into the following categories: safety, liveness, and business-related non-functional properties. For each property, we identify one or more associated threats and analyze the conditions under which they might be violated.

The general threats labeled GT-1 through GT-5 are considered across all property analyses. They represent common issues and pitfalls related to Cosmos SDK application implementations.

The results of this analysis are presented in the *Findings* section.

Assumptions Influencing Heimdall v2 Design

The Heimdall v2 layer serves both as a consensus layer and as a bridge between the two other main components of the Polygon protocol: Ethereum L1 smart contracts and the Bor block production layer. Heimdall v1 and v2 protocol is designed under certain set of assumptions.

These assumptions shape the behavior of the modules under audit scope and highlight **the trust placed in the upstream data and processing order of events emitted by the Ethereum (L1) staking smart contract and Bor chain**. Notably, the module does not include internal validations to enforce these assumptions, instead relying on the correctness of L1 and Bor layers.

The following assumptions, derived from the intended protocol design, were taken into account during the threat analysis:

- **Ethereum L1 assumptions**:
  - **Ethereum Event Integrity and Ordering:** Events emitted by Ethereum smart contracts contain correct data and are correctly ordered, representing the sequence of actions that Heimdall v2 is expected to process. Heimdall is not responsible for reordering transactions related to bridged events (e.g., `MsgSignerUpdate`, `MsgTopupTx`) originating from L1 Ethereum smart contracts.
  - Top-up mechanisms specific assumptions: Dividend account withdrawal amounts are accurately handled within the Ethereum staking contract:
    - No reset of the dividend account fee value is expected upon successful processing of `MsgCpAck` on Heimdall v2.
    - No decrease in this value is assumed.
    - No further processing is expected for dividend account entries after a validator exits the validator set.
- Heimdall v2 consensus layer assumptions:
  - There are at least two-thirds plus one (2/3+) of honest validators.
- **Bor chain assumptions**:
  - At least two-thirds plus one (2/3+) of the Bor block-producing validators are assumed to be honest and correctly reporting produced blocks and block hashes to Heimdall v2 chain.
  - Bor chain is up and running (producing blocks, progressing).

These assumptions directly influence Heimdall v2's logic and its expected behavior. Where applicable, the audit highlights findings that address potential correctness or safety violations in scenarios where these assumptions might be broken.

Property CHP-1: Ensuring Input Validation for Correct Checkpoint Processing

Findings: Missing Validation for BorChainId Field in MsgCheckpoint, Insufficient Validation and Authorization in MsgCpAck Handling

Category: Safety property

Existing validation of inputs on `x/checkpoint` module must ensure correct tx processing and system behavior.

Violation consequences

- Malformed checkpoint transactions being processed could lead to inconsistent Heimdall v2 state, incorrect validator VE injection, potential Invalid checkpoint data could bridged to Ethereum.

Threats

- Validations are missing for the API exposed checkpoint messages.

Conclusion

The property does not hold.

A review of the messages defined in the `x/checkpoint` module revealed that some fields are either not validated at all, or only partially validated:

`MsgCheckpoint`

- The field `BorChainId` is not validated. It is used during side sign byte generation (`GetSideSignBytes`) and unpacking (`UnpackCheckpointSideSignBytes`), but no checks are in place to ensure it is a valid numeric string. If `BorChainId` is malformed (e.g., `"a-b-c"`), the call to `strconv.ParseUint` will silently fail and default the value to `0` without returning an error. The message will still be processed, and an incorrect value may propagate to other components of the system, potentially leading to inconsistent behavior.

`MsgCpAck`

- This message is signed using the `From` field, which is only validated to ensure it's a properly formatted address. This means any user can submit a valid `MsgCpAck`, regardless of whether they were the original proposer.
- Critical fields such as `StartBlock` and `EndBlock` are not validated locally (unlike `MsgCheckpoint`, which includes such checks — see *Property CHP-7: Enforcing Valid Block Range for Checkpoints*.
- The fields `Proposer` and `Number` are validated only by comparing them to the L1 root chain contract (see: code link). However, if the contract is misconfigured or manipulated (even by a malicious proposer), this could result in system corruption.

  For example:

  - If a malicious user submits a `Number` that is not `+1` relative to the current checkpoint ID, it could block future checkpoint submission and negatively impact liveness.
- The `MsgCpAck` struct also contains two unused fields — `TxHash` and `LogIndex` — which are neither validated nor referenced in the code. These fields should either be validated or removed to prevent potential misuse or confusion.

`MsgCheckpointNoAck`

- No missing or risky validations were identified in this message. All relevant checks (sender authorization, timing, proposer role) are correctly enforced.

Property CHP-2: Checkpoint Module Queries Correctness

Findings: GetCurrentProposer and GetProposers are Implemented as x/checkpoint Queries

Category: Safety property

The `x/checkpoint` module queries deliver accurate and reliable data while ensuring no sensitive security information is exposed.

Violation consequences

- The system may return inaccurate or expose sensitive security information or misleading checkpoint data to external actors, leading to incorrect cross-chain behavior or decisions if used in bridging process.

Threats

- Query protobuf definition implementation issues.
- Query issues due to mismatch between query proto definitions and implementation.

Conclusion

The property holds.

All the queries are analyzed for listed issues related to types mismatch in protobuf definitions and keeper logic processing.

There were several informational issues noticed - not impacting the correctness of the property.

Property CHP-3: Ensuring Proper Error Handling in x/checkpoint Queries For Invalid Inputs

Findings: Missing Validations for x/checkpoint Query Inputs Before Processing

Category: Safety property

`x/checkpoint` queries should return appropriate errors for invalid inputs (e.g., wrong address or missing parameters).

Violation consequences

- The system may produce misleading responses, obscure bugs, or cause clients to misinterpret the system's state.

Threats

- Querying for invalid format input does not return error but returns default value.
- Queries do not validate the necessary parameters before processing.

Conclusion

The property does not hold.

Most of the queries do not query for a specific input message defined within the query.proto file. Requests do not contain any fields, except for:

- `QueryCheckpointSignaturesRequest` - `string tx_hash`
- `QueryCheckpointRequest` - `uint64 number`
- `QueryProposerRequest` - `uint64 times`
- `QueryNextCheckpointRequest` - `string bor_chain_id`

The checkpoint signature request and checkpoint request will return an error if the queried items do not exist in the store under the specified key.

However, for the following two queries, some checks are missing, which is why the issue has been reported.

Property CHP-4: Ensuring Signature Validation for x/checkpoint Messages

Category: Safety property

`x/checkpoint` messages must be signed by the corresponding private key of the sender's address.

Violation consequences

- Unauthorized entities could submit forged checkpoint messages, leading to fraudulent or malicious state transitions.

Threats

- Signatures are not checked through defining signer field in protobuf message definitions.

Conclusion

The property holds.

All `x/checkpoint` message types properly declare signer fields in their protobuf definitions.

We reviewed all three messages in the `x/checkpoint` module — `MsgCheckpoint`, `MsgCpAck`, and `MsgCpNoAck` — and verified that each of them **correctly defines the signer field** using this protobuf option.

```
option (cosmos.msg.v1.signer) = "proposer"; // MsgCheckpoint
option (cosmos.msg.v1.signer) = "from";     // MsgCpAck, MsgCpNoAck
```

Property CHP-5: Preventing Arbitrary Checkpoint Message Interference

Findings: Insufficient Validation and Authorization in MsgCpAck Handling

Category: Safety property

Checkpoint messages broadcasted by arbitrary addresses can not interfere with the regular checkpoint flow triggered by the bridge processor.

Violation consequences

- Unauthorized entities could submit forged checkpoint messages, leading to fraudulent or malicious state transitions or spamming.

Threats

- `MsgCpAck` message can be sent for any proposer (different than from address sending and signing the message).
- Any Heimdall v2 user might be able to submit checkpoints, leading to arbitrary or malicious proposals entering the system.

Conclusion

The property does not hold.

`MsgCpAck` is signed by the address specified in the `From` field, which is only validated for proper formatting - not for authorization. The `Proposer` field does not have to match the `From` address, meaning a user could attempt to spoof an acknowledgment for another validator's checkpoint. While `SideHandleMsgCheckpointAck` does verify that the checkpoint data matches the state on the L1 root chain contract, it does not validate whether the sender is an

authorized user to submit that checkpoint acknowledgment. For more details, see *Property CHP-1: Ensuring Input Validation for Correct Checkpoint Processing*.

Property CHP-6: Restricting Checkpoint Submission to Designated Proposer

Category: Safety property

Only the designated proposer from the current validator set can submit new checkpoints for Heimdall and further Ethereum processing.

Violation consequences

- Malicious checkpoint data being bridged to Ethereum, potentially causing on-chain contract failures or misrepresentation of Heimdall's state.

Threats

- The checkpoint proposer's address might be missing from the current validator set if it hasn't been updated, potentially blocking valid proposals.
- Validators who are not the designated proposer might be able to submit checkpoints.
- If no designated proposer exists in the validator set, any validator might be able to submit checkpoints, weakening control over checkpoint creation.
- Any Heimdall v2 user might be able to submit checkpoints, leading to arbitrary or malicious proposals entering the system.

Conclusion

The property holds.

The `MsgCheckpoint` message validation in the `MsgServer` implementation verifies that the `Proposer` field (also the message signer) matches the proposer in the current `ValidatorSet` maintained by the `x/stake` module. The code explicitly checks that a proposer exists in the `ValidatorSet`. This ensures that only the currently designated proposer is authorized to submit a new checkpoint, as intended by the protocol.

However, checkpoint messages are processed in the `PostHandle` phase - that is, in the next block. This creates a potential timing issue: if the proposer changes during this interval (through rotation or governance actions), the checkpoint will still be processed and stored, but the emitted event will reference an outdated proposer. The code doesn't handle or restrict this scenario. While this doesn't directly compromise state integrity, it could potentially confuse or mislead external components that depend on the emitted event to reflect the correct proposer at processing time.

Property CHP-7: Enforcing Valid Block Range for Checkpoints

Findings: Missing Checkpoint Length Validations in MsgCheckpoint, Insufficient Validation and Authorization in MsgCpAck Handling

Category: Safety property

Each checkpoint must have a defined start and end block, where:

- the end block is strictly greater than the start block
- checkpoint span is less than `MaxCheckpointLength`: `end_block - start_ block < MaxCheckpointLength`
- checkpoint span is at least `AvgCheckpointLength`: `end_block - start_ block >= MaxCheckpointLength`

Violation consequences

- Invalid or meaningless checkpoint data, causing Heimdall to track zero or negative progress, may break assumptions in the bridging logic or smart contracts that rely on consistent, forward-moving block intervals.

Threats

- A proposer might submit a checkpoint where:`end_block <= start_ block`
- A proposer might submit a checkpoint greater than expected max number of blocks `MaxCheckpointLength`
- A proposer might submit a short checkpoint, spamming the system

Conclusion

The property does not hold.

`MsgCheckpoint` implements sufficient validation to enforce strictly increasing block ranges.

The `MsgCheckpoint` flow includes multiple safeguards to enforce a valid block range:

- The `ValidateBasic` method enforces `start_block < end_block`, rejecting any zero or negative ranges early.
- In both `MsgServer` and `PostHandleMsgCheckpoint`, additional continuity checks ensure that the new `start_block` equals `lastCheckpoint.EndBlock + 1`, maintaining a strictly increasing checkpoint sequence.

There are some **redundant** validations, as they overlap in functionality. These redundancies are documented separately in the finding *`x/checkpoint` Various Minor Code Improvements* for potential cleanup or refactor.

However, for `MsgCpAck`, **these block range validations are missing**.

- In `MsgServer` and `PostHandleMsgCheckpointAck`, only the equality of `start_block` between the message and the buffered checkpoint is enforced.
- The `SideHandleMsgCheckpointAck` does validate that `start_block` and `end_block` match the values from the L1 root chain contract.

While this provides some assurance, it also introduces a **trust dependency on the correctness of L1 data**. Due to the potential presence of bugs or malicious behavior by external users (*this part is not covered in the scope of the audit and therefore is not analyzed*), invalid `end_block` values (e.g., `end_block == 0` or `end_block <= start_block`) could still be accepted and stored. For more details, see finding *Insufficient Validation and Authorization in `MsgCpAck` Handling*.

Analysis of the `MaxCheckpointLength` and `AvgCheckpointLength` validations shows that **the bridge processor** automatically prepares checkpoints using the `nextExpectedCheckpoint` logic (ref1, ref2), which ensures these length conditions are met. However, since these checks are absent at the message server and side-tx handler levels, a proposer could potentially violate these conditions. This issue has been reported to highlight the need for consistent validation across all levels.

Since it is possible to propose and process the checkpoint of unexpected length - the property does not hold.

Property CHP-8: Ensuring Continuous Block Sequence Between Checkpoints

Category: Safety property

A checkpoint's start block must be exactly one greater than the previous checkpoint's end block, ensuring continuity.

Violation consequences

- Missing blocks or double-count them in checkpoints, the bridged representation on Ethereum may be incomplete or inconsistent.

Threats

- A new checkpoint might start at a block greater than end block + 1.
- A checkpoint might start at or before the previous checkpoint's end block.

Conclusion

The property holds.

The checkpoint module includes strict validation logic to ensure block continuity between consecutive checkpoints. In the `MsgCheckpoint` flow:

- The `MsgServer.Checkpoint` method checks that the `start_block` of the new checkpoint is **exactly one greater** than the `end_block` of the last committed checkpoint:

```
// check if new checkpoint's start block start from current tip
if lastCheckpoint.EndBlock+1 != msg.StartBlock {
  logger.Error("checkpoint not in continuity",
    "currentTip", lastCheckpoint.EndBlock,
    "startBlock", msg.StartBlock)

  return nil, types.ErrDiscontinuousCheckpoint
}
```

- This same logic is repeated in `PostHandleMsgCheckpoint`, reinforcing the continuity requirement before storing the checkpoint in the buffer.

Property CHP-9: Ensuring Single Checkpoint Presence in the Buffer

Category: Safety property

The checkpoint buffer must contain at most one checkpoint at any time.

Violation consequences

- Confusion around which checkpoint should be processed and bridged to the Ethereum root chain.

Threats

- If the buffer contains more than one checkpoint, it could lead to ambiguity about which checkpoint should be processed next.

Conclusion

The property holds.

This property is guaranteed by design (code ref):

```
bufferedCheckpoint collections.Item[types.Checkpoint]
```

Since `bufferedCheckpoint` is of type `collections.Item[types.Checkpoint]`, and `Item[V]` can **only store a single value**, it is guaranteed that there will be at most **one checkpoint** in the buffer at any given time.

However, it is important that checkpoint buffer is appropriately overwritten and flushed. Which is confirmed with properties analysis performed for:

- *Property CHP-10: No Duplicate Checkpoint Processing on Heimdall*
- *Property CHP-14: Checkpoint Buffer Must Be Flushed After ACK on Heimdall*

Property CHP-10: No Duplicate Checkpoint Processing on Heimdall

Category: Safety property

Checkpoints should not be processed on Heimdall if they already exist in the checkpoint buffer or do not sequentially continue from the previously processed checkpoint.

Threats

- The processed checkpoint is not added to the checkpoint buffer.
- If the system does not properly check the buffer before processing, the same checkpoint could be processed multiple times.
- It is possible to add checkpoints that are not sequentially continuing from the previous checkpoint.

Conclusion

The property holds.

The threat analysis confirmed:

- Once a `MsgCheckpoint` is successfully voted on and executed via the post-tx handler on Heimdall, the corresponding checkpoint is added to the checkpoint buffer.
- Prior to processing a new `MsgCheckpoint`, the system verifies the checkpoint buffer. If a checkpoint already exists and has not expired, any subsequent `MsgCheckpoint` will result in an error. This prevents multiple processing.
- New checkpoints can only be processed in two scenarios:
  1. The current checkpoint in the buffer has expired.
  2. The checkpoint has been flushed by the successful execution of `MsgCpACK` in its post-tx handler.
- In addition to buffer-based filtering, the system enforces **sequentiality**: a new checkpoint can only be processed if its starting block height is exactly one block higher than the ending block of the last stored checkpoint on Heimdall. This prevents re-processing of already handled or stale checkpoints.

These checks collectively guarantee that only unique and correctly ordered checkpoints are processed, satisfying the stated safety property.

Property CHP-11: Proposer Update After Checkpoint ACK

Category: Liveness property

Once a checkpoint is ACKed on the Ethereum root chain, the proposer must be updated on Heimdall.

Threats

- A malicious proposer could submit a checkpoint that never gets acknowledged (but also doesn't time out), leading to no proposer update and blocking progress.
- If an ACK does not trigger a proposer update, Heimdall might get stuck with an outdated proposer, preventing new checkpoints from being processed.

Conclusion

Once the Ethereum acknowledges the checkpoint, the `NewHeaderBlock` event is emitted. Bridge listener is listening for these events and triggers `sendCheckpointAckToHeimdall` (code ref) function on the bridge processor. This function handles checkpoint ACKs from the Ethereum L1 (root chain), with first validating that the ACK was not sent already: latest checkpoint `EndBlock` and event's `RootchainNewHeaderBlock End` data are compared (code ref). In cases when the newer checkpoint is ACKed, the `MsgCpAck` tx is broadcasted to the Heimdall chain (code ref).

Aside from the bridge processor, the `MsgCpAck` tx can be created from the arbitrary user, in cases when the ACKs are having issues being bridged from the Ethereum L1 root chain. Also, the missing ACKs are covered with the

NO-ACK mechanism, which also update the proposer (as analyzed with the *Property CHP-12: Proposer Update After Checkpoint NO-ACK*).

Additionally, before the checkpoint is sent to Ethereum, a majority of nodes must vote **Yes** on the `MsgCheckpoint` message. This ensures that a malicious proposer cannot enforce a faulty checkpoint, as every node independently verifies it against their local Bor chain node (ABCI ++ audited scope).

The proposer on Heimdall is updated in the block following a successful `MsgCpAck` vote by the majority of validator nodes when the message's post-tx handler logic (code ref) executes.

Property CHP-12: Proposer Update After Checkpoint NO-ACK

Findings: Potential Execution of MsgCpAck and MsgCpNoAck in the Same Block Height

Category: Liveness property

If a checkpoint is NO-ACKed the proposer must be updated.

Violation consequences

If the proposer is not updated after a checkpoint is rejected/not processed, the same proposer may continue submitting faulty checkpoints, preventing progress.

Threats

- Implementation issues in NO-ACKed processing on Heimdall v2.
- Collision in ACKs and NO-ACKs received for the same checkpoint impacting proposer update.

Conclusion

The property holds.

The threat analysis covered the following points:

- Potential implementation issues with the NO-ACK mechanism, including a review of the timers used.
- Confirmation that the proposer is correctly updated upon `MsgCpNoAck` execution.
- Examination of the possibility of triggering a `MsgCpNoAck` in the same block as receiving a `MsgCpAck`, and the resulting impact on processing.

Implementation of the bridge processor creating the `MsgCpAck` was analyzed, since the messages are expected to be broadcasted automatically (code ref1, ref2) as often as the `NoAckPoolInterval` config parameter is set for the bridge processor. (code ref1, ref2). The default value of the `NoAckPoolInterval` is set to 1010s, which is 10s longer that `CheckpointBufferTime`.

If ACKs are not received on Heimdall for a duration longer than `CheckpointBufferTime`, a `MsgNoCpAck` is broadcasted to Heimdall.

Once the `MsgCpNoAck` is received on the Heimdall chain, prior to being proposed in a block it will be verified for valid data and successful execution (code ref) and in that case enter the block being finalized. Since this is not a side-tx - it will be executed in the same block height when it was received.

- The latest checkpoint must be older than `CheckpointBufferTime` for the NO-ACK to be valid (ref).
- The proposer must send the `MsgCpNoAck` (ref).
  - Since no ACKs were processed, no proposers were updated; thus, latest checkpoint timestamp and the time elapsed since then must be considered when evaluating correct proposers of the `MsgCpNoAck` (code ref):

```
timeDiff := currentTime.Sub(lastCheckpointTime)

// count value is calculated based on the time passed since the last checkpoint
count := math.Floor(timeDiff.Seconds() / bufferTime.Seconds())
```

- The last processed NO-ACK time (`LastNoAck`) is checked to avoid excessive frequency (ref).

We further analyzed the potential collision between `MsgCpAck` and `MsgCpNoAck` during block N processing and potential security implication on proposer selection. The analysis is related with the [*Property CHP-19: Ensuring Authentic Checkpoint Data Across All Processing Phases*. This scenario can occur when a `MsgCpAck` from block N-1 is executed during the `PreBlocker`, while a newly submitted `MsgCpNoAck` in N block targets a really old checkpoint. Although unlikely, it exposes a specific edge case in the protocol design, as described in the *Potential Execution of MsgCpAck and MsgCpNoAck at the Same Block Height*.

While the proposer would still be updated correctly, the problem lies in the illogical execution of these two messages within the same block height.

Property CHP-13: ACKed Checkpoint Must Be Stored as the Latest Checkpoint on Heimdall

Findings: Insufficient Validation and Authorization in MsgCpAck Handling

Category: Safety property

Once a checkpoint is ACKed on the Ethereum root chain, the latest checkpoint must be stored in the checkpoint state.

Violation consequences

If the checkpoint acknowledged is not stored in the Heimdall state, it could lead to multiple processing of the same ACK, incorrectly broadcasting NO-ACKs and block further processing of the checkpoints proposed.

Threats

- If the checkpoint state is not updated upon ACK, the system may operate with an outdated latest checkpoint.
- Updating the latest checkpoint with invalid checkpoint.
- Updating the checkpoints with the latest acknowledged one but not incrementing the Ack counter.

Conclusion

The property does not hold.

After the `MsgCpAck` is voted on by the majority of the validator nodes in Heimdall v2, the post-tx handler will:

- Store the acknowledged checkpoint (code ref) in the map of all acknowledged checkpoints. If the Checkpoint Id already exists, an error is returned (code ref).
  - **Adjustments** to a checkpoint in the buffer are possible if an ACK is received with a different **end block**. However, appropriate validations are currently missing to ensure:
    - `EndBlock` is greater than `StartBlock` for Heimdall v2 `MsgCpAck`.
    - **Checkpoint ID** follows a sequentially increasing order.
- Increment the acknowledgment counter in state (code ref).
- Currently there is no enforcement for ack counter being equal to the checkpoint id being processed - which is necessary for correct functioning of the `GetLastCheckpoint` function (code ref)

The lack of validations allows malicious or unintended, manually crafted `MsgCpAck` transactions to be accepted, as explained in the analysis of *Property CHP-5: Preventing Arbitrary Checkpoint Message Interference* and the linked issue.

Property CHP-14: Checkpoint Buffer Must Be Flushed After ACK on Heimdall

Category: Liveness property

Once a checkpoint is ACKed on the Heimdall v2, the checkpoint buffer must be flushed.

Violation consequences

- Checkpoint placed in the buffer will be present until flushed as expired even though the checkpoint was processed, blocking the further proposed checkpoints processing.

Threats

- Buffer is not cleared after ACK.

Conclusion

The property holds.

Checkpoint buffer is flushed during the post-tx handler execution for the `MsgCpAck` voted on code ref).

The checkpoint stored in the buffer must have:

- the same `StartBlock` as in the ACK processed with the MsgCpAck.StartBlock
- the same `RootHash` - if the `StartBlock` and `EndBlock` are the same for the buffered checkpoint and `msgCpAck`.

It is considered valid, if the post-tx handler is processing the checkpoint that differs from the buffered checkpoint - and the **adjustment** is then performed. `MsgCpAck` message values for `EndBlock`, `RootHash` and `Proposer` are adjusted comparing to the stored buffered checkpoint (code ref).

When a checkpoint is acknowledged on Heimdall, it is stored in the state—either as an exact copy of the buffered checkpoint or in an adjusted form based on the received ACK. The ACK counter is then incremented, and the buffer is flushed (code ref).

Once the checkpoint buffer is flushed, it allows:

- the bridge processor to create and broadcast new Checkpoint to Heimdall v2 (code ref)
- when the `MsgCheckpoint` is broadcast from outside the bridge task (manually), the message server's checkpoint buffer checks will not prevent its processing.

Property CHP-15: Expired NO-ACKed Checkpoints Must Be Flushed Upon New Checkpoint Received

Category: Liveness property

Once the new `MsgCheckpoint` is received on Heimdall v2 the expired non acknowledged checkpoint should be flushed from the checkpoint buffer.

Violation consequences

Not removing expired checkpoints could block the entire checkpoint processing of the received proposed checkpoints on Heimdall v2.

Threats

- Issues in determining the expired checkpoint.

Conclusion

The property holds.

If the `MsgCheckpoint` is received - the message server logic will flush the existing checkpoint from the buffer in case of it being present in the buffer for at least `params.CheckpointBufferTime` (code ref).

Property CHP-16: Heimdall Vote Extensions Must Use Valid Ethereum-Formatted Signatures

Category: Safety property

Heimdall Vote extensions must contain signatures in the appropriate Ethereum format.

Threats

- Invalid signatures in non-Ethereum formats could lead to failed verification and invalid votes.
- Malicious actors could attempt to submit vote extensions with improperly formatted signatures to bypass validation checks.

Conclusion

The property holds.

Signatures included in Heimdall vote extensions are correctly formed over ABI-compatible byte sequences using the expected Ethereum-style structure. The `GetSideSignBytes()` method encodes all fields as fixed-length 32-byte chunks, ensuring compatibility with EVM-based signature verification.

One minor caveat is that the `BorChainId` field is not validated for correctness before inclusion in the signed data (finding *Missing Validation for BorChainId Field in MsgCheckpoint*). While this does not break the format itself, it may affect the semantic integrity of the signature if malformed or unintended values (e.g., default `0`) are used.

Property CHP-17: x/checkpoint Module Events Must Be Accurate and Complete

Category: Safety property

`x/checkpoint` module events emitted as results of tx, side tx and/or post tx handlers must contain correct data.

Violation consequences

- Incorrect event might trigger the wrong action or lead to no action at all (on the bridge).

Threat

- Event is not emitted
- Incorrect type of event is emitted.
- Incorrect event data is emitted.

Conclusion

The property holds.

Events are emitted with the correct type and data each time a message handler (`MsgServer` or post handler) executes successfully. No inconsistencies were observed during analysis.

Property CHP-18: New Voted on Checkpoint Must Trigger Ethereum Contract Calls

Category: Liveness property

Correctly emitted `EventTypeCheckpoint` events must eventually be processed on the bridge and trigger contract call sending the checkpoint over to the Ethereum root chain.

Violation consequences

- Checkpoints are not transferred to the Ethereum root chain.

Threats

- Implementation issues in processing the `EventTypeCheckpoint` and creating the Ethereum contract call.

Conclusion

The property **conditionally** holds.

The assumption is that the retry and delay implementation on the bridge processor once the contract call is made is correctly designed and implemented. The analysis of this artifact was not under the audit scope.

The property analysis covered, correctness review of:

- the `EventTypeCheckpoint` emitted,
- action triggered upon bridge listener catches the event - call root chain contract
- conditions upon which the contract call is not happening

Upon the voted-on `MsgCheckpoint` containing the proposed checkpoint, the post-tx handler executed in the following block during the PreBlocker emits the `EventTypeCheckpoint` (code ref). The emitted event contains identical data to what was placed in the checkpoint buffer (code ref), which includes the checkpoint that will be sent to the Ethereum root chain.

The bridge listener component's `ProcessBlockEvent` processes the `EventTypeCheckpoint` events emitted from Heimdall (code ref), from the last processed block height (code ref1, ref2). `sendCheckpointToRootchain` will be executed and the checkpoint confirmation event from Heimdall will be processed (code ref).

If the checkpoint should be processed on Ethereum (`shouldSendCheckpoint` code ref) - the `createAndSendCheckpointToRootchain` will trigger the root chain contract call (code ref) with the appropriate `sideTxData` (code ref1, ref2) extracted from the `txHash`.

Bridge processor will not trigger the contract call in case of:

- node not being the proposer and if
- checkpoint about to be sent to the Ethereum side is not the next one expected.

This is why the crucial check is querying the root chain contract for the last processed block - `currentChildBlock` from Bor (code ref). This check is essential to determine whether the new checkpoint is the immediate next one to be processed in sequence or if it has already been submitted (i.e., a duplicate).

Also, the Ethereum chain could potentially be slower in finalization of blocks and `currentChildBlock` might be updated after the `sendCheckpointToRootchain` task is already executed with a failure. In that case, the task will be executed for the specific number of **retries** (code ref) and there are also the **task delays** calculated and determined specifically to each of the tasks registered (code ref). Potentially if this retry mechanism is not correctly implemented, the event could be lost - the timings and number of retries must be carefully designed. This mechanism was outside of this scope, which is why the property is conditionally holds. We are assuming that the event will be given multiple chances for processing on the Ethereum side, once the appropriate block height is reached.

The malicious proposer could trigger the contract call, but would need to send the signatures of all the validators that voted yes for the exact `MsgCheckpoint` (code ref). The contract processing of the checkpoint is outside of this audit scope, but it is assumed that the signatures would be verified and the data signed.

Property CHP-19: Ensuring Authentic Checkpoint Data Across All Processing Phases

Findings: Inconsistent Checkpoint Handling When Multiple MsgCheckpoint Messages Exist in a Single Block

Category: Safety property

Checkpoint data in `MsgCheckpoint` must be validated during side transaction handlers and the post-transaction handler, before emitting `EventTypeCheckpoint` and bridging the Ethereum contract call.

Violation consequences

- Invalid data processed in one of the phases could lead to invalid checkpoint transferred to the Ethereum root chain.

Threats

- Side tx handlers voting YES on invalid data.
- Post tx handlers saving invalid checkpoint state.

Conclusion

The property does not hold.

A comparison of the validations executed across the different phases — from message server to side transaction handler to post-transaction handler — confirms that no invalid checkpoint can be stored in state under normal conditions. All critical fields are properly checked before the `EventTypeCheckpoint` is emitted and before any data reaches the bridge.

While there is a possibility that the proposer might change between the time of message submission and the execution of the post handler (i.e., in the next block), this is not expected to cause inconsistencies or issues in practice, as the originally submitted and validated checkpoint remains logically consistent.

However, a potential issue arises if multiple distinct but individually valid `MsgCheckpoint` messages appear within the same block, for example submitted manually via CLI by the current proposer. In such a case, only the first message is accepted and stored, while a different `MsgCheckpoint` may be used in vote extensions and signature aggregation, resulting in a possible mismatch between on-chain state and bridge expectations. This scenario is described in more detail in the finding *Inconsistent Checkpoint Handling When Multiple MsgCheckpoint Messages Exist in a Single Block*.

Property CHP-20: Ensuring bufferedCheckpoint Accurately Tracks Checkpoints Pending Ethereum Bridging

Category: Safety property

`x/checkpoint` `PostHandleMsgCheckpoint` handler must update the `bufferedCheckpoint` to contain the checkpoint about to be sent to the Ethereum chain.

Violation consequences

The lack of tracking for initiated checkpoint bridging could result in processing incorrect checkpoint ACKs on Heimdall or handling the same proposed checkpoint multiple times.

Threats

- `bufferedCheckpoint` is not updated or is updated with the incorrect checkpoint data.
- `bufferedCheckpoint` is flushed before the checkpoint is expired.
- Event `EventTypeCheckpoint` is emitted with incorrect data.
- Bridge processor is not calling the Ethereum smart contract or is submitting invalid

Conclusion

The property holds.

After the majority of validator nodes voted for the proposed checkpoint, the checkpoint is stored in the checkpoint buffer (code ref) during the post-tx handler execution for the `MsgCheckpoint` and contains the `timestamp` of the current block time when the state was updated: `timeStamp := uint64(ctx.BlockTime().Unix())` (code ref).

After the checkpoint addition to the buffer, it is of crucial importance that the checkpoint is:

- Not removed earlier than considered expired.
- The expiration criteria is the same on the bridge processor, prior to sending the checkpoint proposal to the Heimdall and once the `MsgCheckpoint` is received (message server validation logic).

As explained within the conclusion of *Property CHP-18: New Voted on Checkpoint Must Trigger Ethereum Contract Calls*:

> Upon updating the Heimdall v2 state and placing the checkpoint in the `bufferedCheckpoint` the event is emitted `EventTypeCheckpoint` (code ref). This heimdall v2 event is listened by the bridge listener (code ref) and the appropriate processor action will be executed (code ref): if the bridge processor is running on the proposer node and the checkpoint should be processed (code ref).

The checkpoint buffer will contain the checkpoint added when post-tx handler is executed until:

1. The new `MsgCheckpoint` is received, and the checkpoint already in the buffer is considered stale if it's at least `checkpointBufferTime` seconds older than the new one.

   The buffer is flushed when the following condition is met (code ref):

   - The buffer is empty (`checkpointBuffer.Timestamp == 0`), **or**
   - The new checkpoint's timestamp is greater than the existing one, **and** the difference between them is greater than or equal to `checkpointBufferTime` (which is derived from `params.CheckpointBufferTime.Seconds() = 1000s`).

2. After the ACKed checkpoint from Ethereum is processed on the Heimdall with the post-tx handler of the MsgCpACK. Then, after updating the checkpoints on heimdall (adding the latest ACKed Ethereum one) the buffer is flushed, making it ready to receive the new `MsgCheckpoint` pending for the Ethereum bridging (code ref).

Property CHP-21: Correctness of x/checkpoint Export and Import Genesis.json file

Findings: Checkpoint Module State is not Entirely Exported and Initialized, Incomplete Data Validation in Checkpoint ValidateGenesis Function

Category: Safety property

The state of the `x/checkpoint` module is fully exported in the genesis JSON file. Only a correctly formatted JSON file containing valid data can be imported to restore or initialize the module's state.

Violation consequences

- Exporting and then importing the exported state could lead to invalid data.

Threats

- Module's state can not be: exported; exported and then re-imported; exported with correct current state.

Conclusion

The property does not hold.

It has been concluded that the `x/checkpoint` module state is not fully exported/initialized and that there are validations missing, as listed in the issues reported.

Property CHP-22: Checkpoint Module Params Update Enforces Correctness

Findings: Unclear Expectations If AvgCheckpointLength Equals MaxCheckpointLength

Category: Safety property

The `x/checkpoint` module parameters must be updated by an authorized entity to valid, consistent values.

Violation consequences

- Invalid checkpoint parameters (zero values, inconsistent average vs. max length) could corrupt checkpoint generation logic, result in invalid or missing checkpoints, or halt critical cross-chain operations.

Threats

- `MaxCheckpointLength`, `AvgCheckpointLength`, or `ChildChainBlockInterval` improperly validated.
- Acceptance of conflicting parameter values (e.g., avg > max).
- Unauthorized or faulty updates to critical operational parameters.

Conclusion

The property holds.

It is concluded with the Polygon team that `MaxCheckpointLength` and `AvgCheckpointLength` can be equal, so the best course of action would be to change the bridge logic to allow `AvgCheckpointLength <= MaxCheckpointLength`. This comment was added to the issue reported.

All the parameters are validated for non-zero values (code ref). `MaxCheckpointLength` must not be less than `AvgCheckpointLength`; the edge case where `MaxCheckpointLength` equals `AvgCheckpointLength` is explained in the issue.

Property MS-1: x/milestone Module Queries Correctness

Findings: x/milestone Various Minor Code Improvements

Category: Safety property

The `x/milestone` module queries deliver accurate and reliable data while ensuring no sensitive security information is exposed.

Violation consequences

The system may return inaccurate or expose sensitive security information or misleading milestone data to external actors, leading to incorrect cross-chain behavior.

Threats

- Query protobuf definition implementation issues.
- Query issues due to mismatch between query proto definitions and implementation.

Conclusion

The property holds.

There are four queries defined for the `x/milestone` module:

- `GetMilestoneParams`: Correctly returns the appropriate parameters value, if present in the store. Otherwise the error is return.
- `GetMilestoneCount`: Correctly returns the appropriate `uint64` value, if present in the store. Otherwise the error is return.
- `GetLatestMilestone`: There is a minor improvement reported within the analysis of *Property MS-2: Ensuring Proper Error Handling in x/milestone Queries For Invalid Inputs*.
- `GetMilestoneByNumber` If the milestone with the Number does not exists it will not be return. There is a minor improvement reported within the analysis of *Property MS-2: Ensuring Proper Error Handling in x/milestone Queries For Invalid Inputs*.

One potential improvement is listed in the finding *x/milestone Various Minor Code Improvements*.

Property MS-2: Ensuring Proper Error Handling in x/milestone Queries For Invalid Inputs

Findings: Missing Input Data Validations in x/milestone Queries

Category: Safety property

`x/milestone` queries should return appropriate errors for invalid inputs (e.g., wrong address or missing parameters).

Violation consequences

- The system may produce misleading responses, obscure bugs, or cause clients to misinterpret the system's state.

Threats

- Querying for invalid format input does not return error but returns default value.
- Queries do not validate the necessary parameters before processing.

Conclusion

The property does not hold.

Minor improvements to the queries processing in the cases of missing milestones are reported in the informational issue.

Property MS-3: Correctness of x/milestone Export and Import Genesis.json file

Findings: Milestone Module State is not Entirely Exported and Initialized

Category: Safety property

The state of the `x/milestone` module is fully exported in the genesis JSON file. Only a correctly formatted JSON file containing valid data can be imported to restore or initialize the module's state.

Violation consequences

- Exporting and then importing the exported state could lead to invalid or missing data.

Threats

- Module's state can not be: exported; exported and then re-imported; exported with correct current state.

Conclusion

The property does not hold.

It has been concluded that the milestone module state is not fully exported/initialized and that there are validations missing, as listed in the issues reported.

There are no tx validations for the milestone module aside from `MaxMilestonePropositionLength` parameter value. The validations present when finalizing the milestone in `PreBlocker` are compared with the validations present in `ValidateGenesis`. Since the milestone state is not fully exported, there are also validations missing for the missing data, as well.

Property MS-4: Only Valid Milestones Can Be Proposed

Findings: Milestone Proposals Lack Proper Validation of Block Hashes, Malicious Proposer Can Inject Invalid Milestone

Category: Safety property

Only milestones that are valid - i.e.

- containing no more than `MaxMilestonePropositionLength` block hashes and
- starting from the last finalized milestone

can be proposed through the vote extension mechanism.

Violation Consequences

Invalid milestones could lead to the finalization of non-consecutive blocks, blocks with gaps, or blocks that were never produced on the Bor chain. This undermines the integrity of the chain's finalization process.

Threats

- Incorrect validation logic in milestone proposal flow allowing inconsistent milestone data.
- Mishandling of block hash lists exceeding `MaxMilestonePropositionLength`.
- Proposals starting from an incorrect or outdated finalized milestone point.

Conclusion

The property does not hold.

With the threat inspection we were considering the possibility of any malicious node or proposer impacting the finalization with:

- Injecting the invalid milestones - when proposing the block.
- Altering someone else's milestone injected with the VE.
- Proposing and influencing the finalization of invalid milestone.

It was confirmed that it is possible to propose a milestone with no block hashes - which is actually not an expected behaviour. This issue is explained in the finding *Proposer Can Submit Milestone Without Block Hashes*.

Property MS-5: Errors During Milestone Proposal Do Not Affect Consensus or Voting

Findings: Proposer Can Submit Milestone Without Block Hashes, x/milestone Various Minor Code Improvements

Category: Liveness property

The vote extension process continues even when invalid milestones are proposed or errors occur during milestone proposal. Such issues must not interfere with the core consensus voting process.

Violation consequences

- Consensus voting may be halted or prematurely aborted, reducing liveness and halting block production - even though milestone proposals are optional and not critical for consensus progress.

Threats

- Malformed data retrieved from external dependencies (Bor chain blocks) causing failure in proposal building, due to Issues in `GetBorChainBlocksInBatch` implementation.
- Internal panics or unhandled errors in the milestone proposal logic causing the entire vote extension handling to abort.
- Implementation issues in `ExtendVoteHandler` - errors or no milestones proposed are processed and impact aborting, not voting for a block. Not decoupling milestone proposals from the main vote pathway.
- Invalid milestone proposals are not properly filtered or skipped, causing rejection of the entire vote extension instead of proceeding without milestones.

Conclusion

The property conditionally holds.

The assumption is that more than 2/3 of Bor validators voting power will not report invalid block hashes. Currently, there are no checks in Hemidall v2 for confirming that block hashes return from the Bor are correct and existing, produced blocks' block hashes.

Each of the threats was analyzed:

- Malformed/Unexpected data retrieved from Bor chain: if the `GetBorChainBlocksInBatch` was not successfully executed (code ref), the error is returned and nil milestone is return from the `getBlockHashes` (code ref).
  - However, in cases where Bor does not return an error but provides no data, proposals with empty milestones could still be processed further. This issue has been reported, with a detailed explanation of the underlying scenario.
  - Additional minor improvements are reported in *x/milestone Various Minor Code Improvements.*
- Internal **panics** or **errors** in milestone proposition logic: If the milestones can not be proposed or the proposed milestone is invalid (code ref) - the VE is injected with `MilestoneProposition: nil`

```
vt := sidetxs.VoteExtension{
  Height:              req.Height,
  BlockHash:           req.Hash,
  SideTxResponses:     sideTxRes,
  MilestoneProposition: nil,
}
```

  - if the `MilestoneProposition` is `nil` - `VerifyVoteExtensionHandler` will let the validation of the VE continue (code ref).
- `ExtendVoteHandler` implementation: decoupling of VE injection and milestone proposition is present. If the valid milestone can not be added, it will be skipped.

- Invalid milestones ignored: invalid milestones are ignored and not proposed. However, the verification of the VE is rejecting the VE id invalid milestone is present - which is a correct behaviour. It was important to conclude there is no scenario where a node would inject an invalid milestone with a regular usage!

There was an additional improvement for the `ValidateMilestoneProposition` function proposed after the threat inspection, as described in the finding *Proposer Can Submit Milestone Without Block Hashes*.

Property MS-6: Invalid Milestone Proposals Are Excluded from Majority Determination

Findings: Malicious Proposer Can Inject Invalid Milestone

Category: Safety property

Only milestone proposals that are valid - meaning they follow structural and contextual requirements - are considered in the majority milestone determination. Proposals that are invalid (e.g., due to incorrect block ranges, missing blocks, or logical inconsistencies) must be ignored.

Violation consequences

- Inclusion of invalid proposals could result in finalizing milestones that include gaps, non-existent blocks, or unordered chains, undermining the correctness and safety of the finalized state.

Threats

- Missing validation logic in `GetMajorityMilestoneProposition`.
- Invalid proposals not being filtered prior to determining majority.
- Duplicate votes from the same validator being processed.

Conclusion

The property does not hold.

It i possible for a malicious proposer to inject invalid milestone in the proposed block, as described in the finding *Malicious Proposer Can Inject Invalid Milestone* and this milestone will not be excluded from the processing, potentially influencing the majority determination.

Property MS-7: Malformed or Corrupt Milestone Proposals Do Not Disrupt the Majority Determination

Category: Liveness property

Malformed proposals: such as those with incorrect encoding, unexpected structures, or missing required fields are gracefully rejected without impacting the rest of the milestone processing. The system must continue processing remaining valid proposals.

Violation consequences

- Malformed milestone proposals may cause crashes or panics, halting the vote extension logic and stalling consensus progress, even though such proposals are not required for liveness.

Threats

- Lack of decoding error handling or panic recovery.
- Absence of structure validation before processing proposals.
- Over-reliance on all proposals being well-formed.

Conclusion

The property holds.

The analysis evaluated checks to prevent the processing of malformed milestones incorporated in `GetMajorityMilestonePropositions` function.

The only scenario where processing milestones could lead to aborting block finalization is when `VoteExtensions` cannot be unmarshaled (code ref). However, this check is redundant, as it is already performed during the `ProcessProposalHandler`(code ref), making such a situation impossible in the `PreBlocker`.

The milestone field is not required - if the node could not propose the milestone, the field is left empty.

Property MS-8: Byzantine Milestone Submissions Are Tolerated Without Disrupting Majority Selection

Findings: Malicious Proposer Can Inject Invalid Milestone

Category: Safety property

The milestone selection process must remain robust in the presence of Byzantine validators submitting multiple, conflicting, or malformed milestone proposals. Such behavior must not disrupt the determination of the majority milestone or impact block finalization.

Violation consequences

- A single malicious validator could block majority selection or prevent finalization, disrupting consensus or causing inconsistent state advancement.

Threats

- Conflicting proposals from the same validator are not filtered or deduplicated.
- Malformed or mismatched proposals are not properly ignored or validated.
- Missing safeguards lead to panics, hangs, or incorrect majority logic.
- Altered milestone proposals.

Conclusion

The property does not hold.

The analysis confirmed:

- Presence of checks preventing multiple votes from the same validators - this is already checked during the `ProcessProposalHandler`when validating the Vote Extensions - `ValidateVoteExtensions` (code ref).
  - The check is present in the `PreBlocker`'s `GetMajorityMilestoneProposition` (code ref).
- `VoteExtensions` that can not be unmarshaled are the only reason when processing the milestones would impact on aborting the finalization of the block. However, this is a redundant check, already performed during the `ProcesProposalHandler`(code ref), making such a situation impossible in the `PreBlocker`.
- Presence of checks preventing `nil` milestones issues - those milestones are ignored (code ref).
- Only votes with `BlockIDFlagCommit` block id flag are processed with the `GetMajorityMilestoneProposition` (code ref).

During the threat analysis, we considered the possibility of a malicious node or proposer compromising the finalization process through the following actions:

- Injecting invalid milestones when proposing a block.
- Altering another proposer's milestone, injected with the validator evidence (VE).
- Proposing and influencing the finalization of an invalid milestone.

As outlined in the finding *Malicious Proposer Can Inject Invalid Milestone*, there exists a potential vector for influencing the determination of the majority milestone - at the very least, affecting the execution time.

A malicious proposer could inject a milestone that passes validation during `VerifyVoteExtension`, but then alter the milestone's block hashes. This could impact block finalization if the majority of validator voting power depends on the malicious proposer's milestone proposition, especially when their voting power is sufficient to tip the balance.

In conclusion, while such manipulation may not always succeed, it can be possible (similar to the classical up to 1/3 vote extensions proposer censorship).

It is crucial to implement monitoring mechanisms to detect malicious validators.

Property MS-9: Majority-Backed Valid Milestones Are Eventually Finalized and Stored

Category: Liveness property

Milestones that receive a majority of votes and are deemed valid (meaning they include no more than `MaxMilestonePropositionLength` block hashes and begin from the last finalized milestone) must be processed and persisted as the next finalized milestone.

Violation Consequences

- If majority-backed valid milestones are not processed and stored, milestone finalization will stall. This may block dependent processes (e.g., state sync, etc.) and degrade overall chain progress, even when consensus has been reached.

Threats

- Failure to persist the finalized milestone after majority voting.
- State inconsistency in milestone finalization logic.
- Logic bugs preventing detection of majority even when it exists.

Conclusion

The property holds.

After determining the majority milestone - if it exists and is valid - it will be finalized (stored) in the state (code ref).

The only way of persisting the milestone is to `AddMilestone` (code ref) and `SetMilestoneBlockNumber` (code ref) throw an error. However, they both perform basic read/write on the key-value storage, so unless storage is corrupted everything operates as expected and the milestone is finalized.

Logic in detection of majority will not abort the detection of the majority of the milestones, unless something is fundamentally wrong - which is only theoretically possible, as the same validations are performed upstream in `ValidateVoteExtensions`. Specifically, it aborts if::

- Vote extension `Unmarshal` fails (code ref) - already validated in `ProcessProposal` via `ValidateVoteExtensions` (code ref).
- Validator address conversion fails (code ref) also covered by upstream `ValidateVoteExtensions` (code ref).
- Fails because a validator does not belong in the `validatorSet` (code ref) We already check `oposition` returns without an `error` here.
- Validator not found in `validatorSet` (code ref) - already checked in `ValidateVoteExtensions` (code ref) and explained in *Property MS-11: Majority Determination Reflects Validator Set at Time of Milestone Proposal*.

- The `supportingValidatorList` is empty (code ref) in that case, `totalSupportingPower` is 0, and the function safely returns `nil` (code ref) without error.

Property MS-10: Function Returns nil When No Majority Milestone Exists

Category: Liveness property

When no single milestone reaches majority support, the function should gracefully return `nil` indicating absence of consensus of milestone - no milestone will be finalized!

Violation consequences

- Unclear behavior when no majority exists could cause inconsistent results or cause the `PreBlocker` and finalization of the block to fail.

Threats

- Returning the longest voted on milestone even if below majority.
- Returning random proposal when all support is sparse.
- Failing to distinguish between "no data" and "no majority".

Conclusion

The property holds.

A `nil` milestone is correctly returned in two types of situations:

1. When an **error** occurs during milestone determination - this signals a critical issue, and the block finalization process is aborted. However, such errors are expected to be unreachable in practice, as the same checks exist earlier in the `ProcessProposalHandler` (code ref) - the analysis performed in *Property MS-9: Majority-Backed Valid Milestones Are Eventually Finalized and Stored*.
2. When **no milestone can be formed** due to protocol-defined conditions, in which case a `nil` is returned **without an error**, as expected. These conditions are:
   - No block hash has >=2/3 of total validator voting power (i.e., no majority) (code ref).
   - No block starts at the expected milestone `startBlock`, which must be `lastEndBlock + 1` (code ref).
   - No continuous block sequence (starting from `startBlock`) is supported by the validator majority (code ref).
   - The longest continuous block sequence is not supported by 2/3+1 voting power (code ref).

In all these cases, the logic correctly returns a `nil` milestone and avoids finalization, aligning with the expected system behavior. The function's return values are handled conservatively: a `nil` milestone from `GetMajorityMilestoneProposition` is never processed further, as it denotes an invalid or absent milestone proposal (code ref):

```
if majorityMilestone != nil {
  if err := milestoneAbci.ValidateMilestoneProposition(ctx, &app.MilestoneKeeper,
  ↪  majorityMilestone); err != nil {
      logger.Error("Invalid milestone proposition", "error", err, "height", req.Height,
↪  "majorityMilestone", majorityMilestone)
      // We don't want to halt consensus because of invalid majority milestone proposition
    } else {
      isValidMilestone = true
    }
  }

  if isValidMilestone {
```

```
    // AddMilestone over cached context.
    ...
  }
}
```

Property MS-11: Majority Determination Reflects Validator Set at Time of Milestone Proposal

Category: Safety property

The majority milestone must be determined using the validator set and corresponding voting powers from the block height at which the milestone was proposed. Since post-handlers are executed in the `PreBlocker`, using the updated validator set from the current block could incorrectly influence the outcome.

Violation consequences

- Incorrect majority selection due to misaligned validator powers may result in invalid milestone finalization or missed consensus on valid milestones.

Threats

- Incorrect validator set or voting power used during milestone processing.

Conclusion

The property holds.

`validatorSet` originates from the previous block validator set (code ref1, ref2):

```
validatorSet, err := getPreviousBlockValidatorSet(ctx, app.StakeKeeper)
...
majorityMilestone, aggregatedProposers, proposer, err :=
↪  milestoneAbci.GetMajorityMilestoneProposition(validatorSet, extVoteInfo, logger,
↪  lastEndBlock)
```

Since:

- the previous block validator set is updated at the `ApplyAndReturnValidatorSetUpdate`(code ref) called from `EndBlocker` (code ref) and
- the `PreBlocker` executes before `EndBlocker` - the check on the `validatorSet` (code ref) performed in `ValidateVoteExtensions` applies here, as well

majority determination is performed with the validator set voting power that was present in the moment of injecting the milestones within the vote extensions.

Property MS-12: Deterministic Majority-Based Milestone Selection

Category: Safety / Determinism property

Given the same set of milestone proposals, the system deterministically returns the same majority milestone (if one exists), regardless of the iteration order or validator identifiers.

Violation consequences

- Non-deterministic milestone selection can lead to divergence in vote extension results among validators, causing mismatches in block proposals or vote extensions and ultimately risking consensus failure.

Threats

- Iteration over unsorted maps or sets.
- Inconsistent encoding, hashing, or equality checks between milestones.
- Selection logic unintentionally influenced by validator order or IDs.

Conclusion

The property holds.

With our analysis we concluded:

Iterations:

- Maps in Go iterate in **random order**, so this loop must always be followed by deterministic sorting **before** making decisions or producing output.
- Iteration with `range` over
  - `validatorVotes`: range over this map is performed, but also sorting of `supportingValidatorList`, which mitigates the problem in this context. sorted `supportingValidatorList` is used when `aggregatedProposerHash` is created (code ref).
  - `blockToHash` is read deterministically, but built from map iterations (code ref). This part seems okay since the inner loop enforces deterministic tie-breaking (code ref):

```
if blockHashVotes[blockNum][hashStr] > blockVotingPower[blockNum] ||
   (blockHashVotes[blockNum][hashStr] == blockVotingPower[blockNum] &&
    hashStr < common.BytesToHash(blockToHash[blockNum]).String()) {
```

  - `blockVotingPower`: is looped and used for initialization of `blockNumbers` (code ref), but `blockNumbers` is explicitly sorted before iteration (code ref).
  - `blockHashVotes`: used only for deterministic reading.
  - `valAddressToVotingPower`: used only for deterministic reading.
  - and `validatorAddresses`: used only for deterministic reading.

Enconding, Hashing, Equality:

- Use of `common.BytesToHash(blockHash).String()`
- **Equality checks** use `bytes.Equal` (code ref)
- aggregated proposer hash:

Logic influenced by the validator order:

- The function assumes `extVoteInfo` is provided in the same order to all nodes - this is an external assumption and must be guaranteed upstream (CometBFT). This is important in cases of duplicates votes. Duplicate votes are not possible since `ValidateVoteExtensions` executes validations during `ProcessProposalHandler` and `PrepareProposalHandler` (code ref).

`GetMajorityMilestoneProposition` function could potentially be refactored and improved. S**plitting into smaller helper functions** for readability and testability: Vote collection; Vote aggregation; Majority filtering; Range detection; Validator filtering; Hash computation as suggested in the finding ***`x/milestone` Various Minor Code Improvements***.

Property MS-13: Only the Milestone with Super-majority Support (2/3+ of Voting Power) Can Be Selected

Category: Safety property

A milestone is only selected if it is supported by:

- at least two-thirds of the total validator voting power, ensuring milestone is returned only if it has support from a strict majority of voting power, not just a plurality.

Violation consequences

- Selecting a milestone without super-majority support may misrepresent validator consensus, enabling adversarial or weak proposals to be finalized and potentially leading to state divergence.

Threats

- Incorrect total voting power or majority threshold calculations.
- Mishandling of abstentions, missing votes, or duplicated votes.
- Edge cases with even validator sets not handled correctly (e.g., exactly 50% support misclassified as majority).

Conclusion

The property holds.

Milestones are selected based on a super-majority threshold (>=2/3 of the total validator voting power), calculated from the full validator set.

In cases where validators report conflicting block hashes for the same block height and no single hash surpasses the threshold, a deterministic tiebreaker is applied: the lexicographically smallest string representation of the block hash is selected (code ref). This ensures only one block hash can be selected, even in the presence of a tie.

Missing votes from validators are not processed, but their voting power is still counted toward the threshold calculation, ensuring accurate enforcement of the 2/3 requirement (code ref1, ref2, ref3).

Property MS-14: Only the Longest Continuous Sequence Is Considered for Finalization

Category: Safety property

A milestone is only selected if it represents the **longest contiguous sequence of block hashes**, each individually supported by at least **two-thirds of the total validator voting power**.

Violation consequences

- Selecting a shorter or fragmented sequence despite the existence of a longer, valid one may lead to inconsistent views of finalized state across nodes, undermine milestone liveness guarantees, or break deterministic execution assumptions.

Threats

- Failure to detect continuity in `majorityBlocks`.
- Disordered or improperly sorted `majorityBlocks` leading to fragmented range selection.
- Incorrect `startBlock`.
- Multiple sequences competing for selection without deterministic prioritization.

Conclusion

The property holds.

We concluded that:

- `startBlock` is retrieved from the last finalized milestone in store (code ref) - if it exists, and the `endBlock` is **increased by 1,**expecting this value to be the next milestone's starting block.

  - if there are no milestones finalized, it is expected to start from 0 (code ref).
  - If there is no appropriate start block detected among proposed blocks (supported by the majority of the validators - there will be no milestone proposed for further selection (code ref1, ref2).

- `blockNumbers` in milestone sequence are sorted (code ref).
- `majorityBlocks` will be created in ordered manner with looping through the sorted `blockNumbers` (code ref).
  - each of the added block must have more than `majorityVP` (2/3+1 of the total voting power) - if this is not the case, it will not be added.
- Continuity is ensured by (code ref):

```
// Find the first continuous range starting from startBlock
endBlock := startBlock
for i := 0; i < len(majorityBlocks); i++ {
  if majorityBlocks[i] == startBlock {
    // Find continuous blocks after startBlock
    for j := i + 1; j < len(majorityBlocks); j++ {
      if majorityBlocks[j] == endBlock+1 {
        endBlock = majorityBlocks[j]
      } else {
        break
      }
    }
    break
  }
}

blockCount := endBlock - startBlock + 1
blockHashes := make([][]byte, 0, blockCount)
for i := startBlock; i <= endBlock; i++ {
  blockHashes = append(blockHashes, blockToHash[i])
}
```

  As soon as a discontinuity is detected, the sequence is considered terminated, and the milestone will include only the block hashes up to the last block that maintained continuity.

- After this the super-majority support is re-checked (code ref) as explained in more details in *Property MS-13: Only the Milestone with Super-majority Support (2/3+ of Voting Power) Can Be Selected*.

Property MS-15: Milestone Module Params Update Enforces Correctness

Category: Safety property

The `x/milestone` module parameters must be updated by an authorized entity to valid, properly validated values to ensure milestone generation remains safe.

Violation consequences

- Acceptance of invalid milestone proposition parameters could lead to broken milestone processing, generation of invalid milestones, or missed milestone finalization, compromising chain correctness.

Threats

- Missing validation on `MaxMilestonePropositionLength`.
- Unauthorized or improperly validated governance updates.

- Failure to reject invalid configurations in `UpdateParams`.

Conclusion

The property holds.

There is only one property `MaxMilestonePropositionLength`, updated by an authority address - governance module, and it is impossible to define zero length milestones (code ref).

Property STK-1: Restricting x/stake EndBlocker Panics to Critical Chain Integrity Failures

Findings: Panic Caused by Invalid Validator Public Key in EndBlocker

Category: Liveness property

The `x/stake` module's `EndBlocker` should only panic and halt the chain when continuing execution is not viable or could cause critical chain inconsistencies.

Violation consequences

- Halting the chain.

Threats

- A malicious user could craft transactions specifically designed to trigger panics.
- Implementation contains panics that are triggered in cases of non critical chain state.

Conclusion

The property holds.

The EndBlocker implementation in the `x/stake` module generally follows the *Property STK-1: Restricting x/stake EndBlocker Panics to Critical Chain Integrity Failures* expectations by only panicking in critical conditions (e.g., incomplete module setup).

However, one critical deviation was found — a **panic in the `CmtConsPublicKey()` function** when encountering an invalid validator public key.

This issue is described in detail in the finding *Panic Caused by Invalid Validator Public Key in `EndBlocker`*.

Property STK-2: Ensuring Input Validation for Correct Validator Updates

Findings: Lack of Authorization Checks for From Field in x/stake Messages, Heavy Reliance on L1 Event Logs Without Sufficient Local Validation in x/stake Messages

Category: Safety property

Existing validation of inputs on `x/stake` module must ensure correct tx processing and system behavior.

Threats

- Validations are missing for the API exposed checkpoint messages:
  - `MsgValidatorJoin`
  - `MsgStakeUpdate`
  - `MsgSignerUpdate`

- `MsgValidatorExit`

Conclusion

The property does not hold.

While the input validation for the `x/stake` module messages is generally well-implemented, several areas require attention. Potential issues include over-reliance on the correctness of L1 event log data, the improper use of a custom `Validate()` method instead of the standard `ValidateBasic()` method, and the lack of authorization checks for message senders. These concerns are further detailed in the following findings:

- *Heavy Reliance on L1 Event Logs Without Sufficient Local Validation in `x/stake` Messages.*
- *Implemented `Validate()` Methods Instead of `ValidateBasic()` for `x/stake` Messages.*
- *Lack of Authorization Checks for `From` Field in `x/stake` Messages.*

Additionally, although some validations are performed correctly during `MsgServer` execution, they are not consistently revalidated during post-handler execution. This could pose a risk if the same message appears across multiple transactions in the same block. However, this scenario is mitigated by the correct sequence computation and storage during the post-handler phase, ensuring that duplicate or replayed messages are rejected based on sequence collision.

Property STK-3: Stake Module Queries Correctness

Findings: Unsafe x/stake IsStakeTxOld Query, Improve Error Handling in x/stake Queries, Missing Input Data Validations in x/stake Queries

Category: Safety property

The `x/stake` module queries deliver accurate and reliable data while ensuring no sensitive security information is exposed.

Violation consequences

- The system may return inaccurate or expose sensitive security information or misleading milestone data to external actors, leading to incorrect cross-chain behavior.

Threats

- Query protobuf definition implementation issues.
- Query issues due to mismatch between query proto definitions and implementation.

Conclusion

The property does not hold.

Several improvements in error logging and validation were suggested in the reported issues.

The primary concern centers around the `IsStakeTxOld` query (code ref) query, which is currently marked as a `module-safe query`- yet this classification is not accurate. Additionally, another issue has been reported related to this conclusion.

Property STK-4: Ensuring MsgValidatorJoin Contains Accurate Ethereum Data

Category: Safety property

`MsgValidatorJoin` created on bridge processor must contain correct Ethereum join validator data in order for Heimdall validators to `VOTE_YES` during `ExtendVoteHandler`.

Violation consequences

- Validator might not be added on the Heimdall validator set.

Threats

- Message is containing incorrect data, impacting the voting for the side tx.
- Side tx handler logic is not correctly validating validator join event data.

Conclusion

The property holds.

The `MsgValidatorJoin` message is thoroughly validated across all phases of processing. The side transaction handler (`SideHandleMsgValidatorJoin`) performs strict cross-verification of all critical fields - such as `ValId`, `ActivationEpoch`, `Amount`, `Nonce`, and `SignerPubKey` - against the actual Ethereum event logs. In addition, replay attacks are mitigated using a sequence number derived from the transaction's `BlockNumber` and `LogIndex`, and proper validation is enforced to prevent malformed or duplicate validators.

These mechanisms ensure that only valid, correctly-sourced validator join requests originating from the Ethereum staking contract are accepted. The system relies on the correctness of Ethereum logs, which is consistent with the design assumptions of Heimdall v2.

Property STK-5: Preventing Duplicate Validator Joins in the Heimdall Validator Set

Category: Safety property

Validator already present in Heimdall validator set can not be joined again.

Violation consequences

Validator might be updated/reset on the Heimdall validator set.

Threats

- Invalid checks if the validator is present.
- Updating the signer to the same value is perceived as new validator joining the network.

Conclusion

The property holds.

The `ValidatorJoin` message is sufficiently protected against duplicate entries in the Heimdall validator set. It includes checks against both `ValId` and `Signer`, and replay protection through a sequence number. Additionally, side handler logic verifies the validity of event log data from Ethereum before allowing any new validator to join.

The Polygon team has confirmed that we can rely on the assumption that the L1 Event Log only contains information from finalized blocks. This ensures that messages from potentially invalid or non-finalized blocks are not processed, preventing inconsistencies between the L1 chain and the Heimdall v2 chain.

Property STK-6: Ensuring MsgStakeUpdate Accurately Reflects Ethereum Validator Stake for Heimdall v2 Voting

Category: Safety property

`MsgStakeUpdate` created by the bridge processor must contain correct Ethereum validator stake update data in order for Heimdall v2 chain nodes to `VOTE_YES` during `ExtendVoteHandler`.

Violation consequences

- Validator might be incorrectly updated on the Heimdall validator set.

Threats

- Message is containing incorrect data, impacting the voting for the side tx.
- Side tx handler logic is not correctly validating validator stake update event data.

Conclusion

The property holds.

The implementation of `MsgStakeUpdate` ensures correctness through comprehensive validation steps in the side handler (`SideHandleMsgStakeUpdate`). All relevant fields — including `ValId`, `NewAmount`, `Nonce`, and `BlockNumber` — are cross-checked against confirmed event log data retrieved from Ethereum. This guarantees that Heimdall validators only vote `YES` on accurate, finalized stake updates.

Property STK-7: Preventing MsgStakeUpdate from Setting Invalid Zero Stake Values

Category: Safety property

`MsgStakeUpdate` can update staked amount only to non zero values.

Violation consequences

- Invalid `x/stake` state validator set updates.

Threats

- `MsgStakeUpdate` can update staked amount to 0.
- `MsgStakeUpdate` can be created on the bridge processor for Ethereum event updating the staked amount to zero.

Conclusion

The property holds.

Although there is no explicit check preventing `NewAmount == 0` in the `MsgStakeUpdate` handlers, this scenario is effectively handled by the `GetPowerFromAmount()` method, which expects the input amount to be greater than one. If a zero value is passed, the method will return an error, causing the message to be rejected.

This ensures that stake updates resulting in zero voting power cannot be persisted, even if such a message were constructed or propagated by the bridge.

Property STK-8: Ensuring MsgValidatorExit Contains Correct Ethereum Data for Heimdall v2 Voting

Findings: Validator Object Retains Non-Zero Voting Power After Exit

Category: Safety property

`MsgValidatorExit` created by the bridge processor must contain correct Ethereum validator data in order for Heimdall v2 chain nodes to `VOTE_YES` during `ExtendVoteHandler`.

Violation consequences

- Invalid `x/stake` state validator set updates.

Threats

- Removing the invalid validator from the validator set.
- Not removing the validator, due to invalid data present in the `MsgValidatorExit`.

Conclusion

The property conditionally holds.

The implementation ensures that `MsgValidatorExit` messages are thoroughly validated against Ethereum event logs in `SideHandleMsgValidatorExit`, covering critical fields like `ValId`, `BlockNumber`, `DeactivationEpoch`, and `Nonce`.

Although local validation does not include these fields, the bridge logic correctly handles them through log comparison before applying any state changes. Therefore, the risk of removing or failing to remove a validator based on invalid input is mitigated by these safeguards.

However, it is important to note that the system implicitly trusts the correctness of the Ethereum event log, and `From` is not authenticated.

Property STK-9: Allowing Removed Validators to Rejoin the Heimdall Validator Set

Findings: Incorrect SignerPubKey Validation Prevents Exited Validators to Rejoin

Category: Liveness property

Validator removed from the Heimdall validator set can once again join the validator set.

Violation consequences

- Validator that was removed can't eventually rejoin the validator set, meaning that the system permanently blocks participation.

Threats

- `x/stake` state is not updated correctly.
- Existing msg, side tx and post tx validations prevent validator from re-joining the validator set.

Conclusion

The property does not hold.

The current `x/stake` implementation does not support rejoining of removed validators. Rejoining with the same `SignerPubKey` is rejected, as the signer is already present in store. Using a new `SignerPubKey` may cause a mismatch with the L1 event log, leading to vote rejection in the side handler.

Property STK-10: Ensuring MsgSignerUpdate Accurately Reflects Ethereum Validator Data for Heimdall v2 Voting

Findings: Missing Validation for Negative FeeToken Amounts During Transfers

Category: Safety property

`MsgSignerUpdate` created by the bridge processor must contain correct Ethereum validator data in order for Heimdall v2 chain nodes to `VOTE_YES` during `ExtendVoteHandler`.

Violation consequences

- Invalid `x/stake` state validator set updates.

Threats

- Message is containing incorrect Ethereum event data, leading to failure of the validation during the side tx handler.

Conclusion

The property conditionally holds.

The `SideHandleMsgSignerUpdate` function provides thorough validation of Ethereum event log data, ensuring that all fields - including `ValidatorId`, `BlockNumber`, `Nonce`, and `SignerPubKey` - are matched accurately before the update is approved.

Property STK-11: Replay Protection from duplicate Ethereum events in x/stake module

The replay protection mechanism for `x/stake` messages ensures that transactions already processed by post-transaction handlers cannot be re-executed.

Category: Safety property

Violation consequences

- Possibility of executing one `x/stake` message multiple times can impact the validators stake/power and introduce possibility of taking over the network.

Threats

- Sequence created for each processed `x/stake` message are not unique.
- Sequence is not stored in the state for the `x/stake` post tx handler.

Conclusion

The property holds.

The `x/stake` module correctly implements replay protection for all critical messages.

Each message creates a unique sequence from `(blockNumber, logIndex)`, checks it before execution, and stores it after a successful operation.

There are no gaps found that could allow duplicate Ethereum events to be processed twice.

Property STK-12: Correctness of x/stake Export and Import Genesis.json file

Findings: Stake Module State is not Entirely Exported and Initialized, Incomplete Data Validation in Stake ValidateGenesis Function

Category: Safety property

The state of the `x/stake` module is fully exported in the genesis JSON file. Only a correctly formatted JSON file containing valid data can be imported to restore or initialize the module's state.

Violation consequences

- Exporting and then importing the exported state could lead to invalid data

Threats

- Module's state can not be: exported; exported and then re-imported; exported with correct current state.

Conclusion

The property does not hold.

It has been concluded that the `x/stake` module state is not fully exported/initialized and that there are validations missing, as listed in the issues reported.

Property TU-1: Ensuring Input Validation for Correct Top-up Processing

Category: Safety property

Existing validation of inputs on `x/topup` module must ensure correct tx processing and system behavior.

Violation consequences

- Malformed `x/topup` transactions being processed could lead to inconsistent Heimdall v2 state, impacting the incorrect amount of the fee balance on dividend accounts of the Heimdall validators.

Threats

- Validations are missing for the API exposed topup messages.
- `MsgTopupTx` can contain negative values for Fee - amount of coins for the top-up and potentially influence validator and dividend account balances, in a unpredictable way.
- `MsgWithdrawTx` can contain negative values for Amount - amount of coins for the withdrawal and potentially influence validator and dividend account balances, in an unpredictable way.
- `MsgWithdrawTx` can not send 0 as `Amount` value.

Conclusion

The property holds.

Content of`MsgWithdrawFeeTx` and `MsgTopupTx` is validated at message server level. TopUps are part of the side transaction processing. The content of message `MsgTopupTx` is validated by the side transaction handler. Beside transaction hash, log index and block number being cross checked against the main chain this includes validation of the top up amount `msg.Fee` to be at least of a defined amount `DefaultFeeWantedPerTx`. The minimal amount check is also done before the triggering event `topup`is submitted.

There are no explicit checks of the amount of `MsgWithdrawFeeTx` message when processing the message. 0 as amount value is allowed on message level to handle the special case to result in full spendable amount of the account but rejected as final determined amount afterwards. Negative values are rejected as well on the level of coin validation done by the bank module when sending the amount to an account.

Property TU-2: x/topup Module Queries Correctness

Findings: TopUp - Unsafe Queries

Category: Safety property

The `x/topup` module queries deliver accurate and reliable data while ensuring no sensitive security information is exposed.

Violation consequences

- The system may return inaccurate or expose sensitive security information or misleading top-up data to external actors, leading to incorrect cross-chain behavior or decisions if used in bridging process of checkpoints (stake withdrawals).

Threats

- Query protobuf definition implementation issues.
- Query issues due to mismatch between query proto definitions and implementation.

Conclusion

The property does not hold.

All defined queries of `topup` module (`IsTopupTxOld`, `GetTopupTxSequence`, `GetDividendAccountByAddress`, `GetDividendAccountRootHash`, `VerifyAccountProofByAddress`, `GetAccountProofByAddress`) use `module_query_safe` option set to `true` requiring determinism on results and gas usage.

There is a potential issue for `IsTopupTxOld` , `GetTopupTxSequence` and `GetAccountProofByAddress`queries, which perform calls to a contract to get transaction information from main-net or staking information. Since `module_query_safe` is set to `true`, the query result and gas consumption must be deterministic to allow non-state-breaking calls between modules, keepers or CosmWasm contracts. This cannot be guaranteed due to the external dependency of the contract call done in the query, which can timeout. This can lead to state inconsistencies when other keepers or modules use these queries.

`GetDividendAccountByAddress` is using `util.FormatAddress()` which accepts any length of addresses and doesn't verify if the address is valid hex encoding. This can lead to potential problems.

`VerifyAccountProofByAddress` does not perform input validation on user address provided accepting any string length or content for user addresses to be queried. String comparison `EqualFold` in `GetAccountProof()` iterating over the full string length can be expensive.

Property TU-3: Ensuring Proper Error Handling in x/topup Queries For Invalid Inputs

Findings: Topup - Query Input Validation Missing

Category: Safety property

`x/topup` queries should return appropriate errors for invalid inputs (e.g. wrong address or missing parameters).

Violation consequences

- The system may produce misleading responses, obscure bugs, or cause clients to misinterpret the system's state.

Threats

- Querying for invalid format input does not return error but returns default value.
- Queries do not validate the necessary parameters before processing.

Conclusion

Property does not hold.

`GetTopupTxSequence` and `IsTopupTxOld` do not validate `TxHash` parameter. Accepting any string content and length does not comply to the expectations from underlying decoding function `DecodeString()` which is expecting only hexadecimal characters. Caller `common.FromHex()` does not guarantee this expectation. Errors returned from decoding function are ignored and content is converted to `Hash` and passed to the contract caller.

`VerifyAccountProofByAddress` query does not validate input `Address` nor `Proof`. Underlying `GetAccountProof()` does not do any user address check either before trying to get the merkle path. Note that `GetAccountProof()` is ignoring error returned from `tree.GetMerklePath()`.

`GetAccountProofByAddress` query does not validate input `Address` accepting any string content and passing it to `GetAccountProof()`.

Property TU-4: Correctness of x/topup Export and Import Genesis.json file

Category: Safety property

The state of the `x/topup` module is fully exported in the genesis JSON file. Only a correctly formatted JSON file containing valid data can be imported to restore or initialize the module's state.

Violation consequences

- Exporting and then importing the exported state could lead to invalid data

Threats

- Module's state can not be: exported; exported and then re-imported; exported with correct current state.

Conclusion

The property holds.

The `x/topup` keeper stores dividends for accounts and set of topup sequences. Both (`DividendAccount` , `TopupSequences`) are part in the genesis state definition `GenesisState` and ensured to be populated on `InitGenesis()`. Incomplete content are rejected on initialization. `ExportGenesis()` ensures all topup sequences and dividend accounts are part of the exported genesis state and fails otherwise.

Validation is done by the related message handler of the keeper when values are set by `SetTopupSequence()` , `SetDividendAccount()`.

Property TU-5: Ensuring Signature Validation for x/topup Messages

Category: Safety property

`x/topup` messages must be signed by the corresponding private key of the sender's address. It is not enforced that a current block proposer is a signer.

Violation consequences

- Unauthorized entities could submit forged top-up messages, leading to fraudulent or malicious state transitions.

Threats

- Signatures are not checked through defining signer field in protobuf message definitions.

Conclusion

The property holds.

For both messages,`MsgToupTx` and `MsgWithdrawFeeTx`, the `Proposer` field is defined to be the address which also has to sign the message when it is sent to Heimdall. Note that `Proposer` is not to be confused with block proposer. This is guaranteed by Cosmos SDK implementation for protobuf messages where a field is tagged as `signer` (setting the option `cosmos.msg.v1.signer` for the related field of the message).

Property TU-6: Bridge-Triggered Top-Up Fee Flow Integrity

Category: Safety property

Top-up fee messages broadcasted aside from the bridge can not interfere with the regular top-up fee flow triggered by the bridge processor.

Violation consequences

- Unauthorized entities could submit forged topup messages, leading to fraudulent or malicious state transitions or spamming.

Threats

- `MsgTopupTx` message can be sent by an arbitrary or malicious user in parallel with automatized bridge transferred messages and introduce issues.
- `MsgWithdrawalTx` message can be sent by an arbitrary or malicious user in parallel with automatized bridge transferred `MsgTopupTx` messages or other `MsgWithdrawalTx`and introduce issues.

Conclusion

The property holds.

Anyone (usually bridging process) can be a proposer of a topup, sign and send `MsgTopupTx` . Proposer of a topup has to sign the transaction (ref).

Spamming is discouraged by introducing `DefaultFeeWantedPerTx`.

Proposer of a `MsgWithdrawFeeTx` can be only a validator from which account assets are withdrawn (ref). Proposer of withdrawal has to sign transaction (ref) and by that making this process spam and fraud proof.

Because of L1 transaction information crosscheck of `MsgTopupTx` (*Property TU-9: Ensuring Authentic Topup Data Across All Processing Phases*), replay attack protection using sequences (*Property TU-10: Replay Protection from duplicate Ethereum events in `x/topup` module*) and restrictions of who can be `MsgWithdrawalTx` proposer there is no possible sequence of submitted `MsgTopupTx` and `MsgWithdrawalTx` messages that could produce unexpected (malicious) behaviour.

Property TU-7: Restricting Withdrawal to Account Owners

Category: Safety property

Only owners of the account are authorized to submit`MsgWithdrawFeeTx` to withdraw from their fee account/ balance.

Violation consequences

- Arbritrary user could impact validator's balance and reduce its fee balance.

Threats

- No appropriate signature field defined in the protobuf definition.

Conclusion

The property holds.

`MsgWithdrawFeeTx` defines `Proposer` for the signature field. Only the owner of the account with the address given in the`Proposer` field can submit the withdraw messages as it needs to be signed with the related key. The account with the address in `Proposer` is then used for the withdraw operations using the bank module.

Property TU-8: x/topup Module Events Must Be Accurate and Complete

Category: Safety property

`x/topup` module events emitted as results of tx, side tx and/or post tx handlers must contain correct data.

Violation consequences

- Incorrect event might trigger the wrong action or lead to no action at all (on the bridge).

Threats

- Event is not emitted.
- Incorrect type of event is emitted.
- Incorrect event data is emitted.

Conclusion

The property holds.

All module events for `MsgTopupTx` use `topup` as event type and `MsgWithdrawFeeTx` are using `fee-withdraw`. Events are emitted for each of these messages on related message handler with correct data.

Noticed during investigation: In function `BroadcastToHeimdall()` in `bridge/broadcaster/broadcaster.go`, `WithSignMode` is redundant.

Property TU-9: Ensuring Authentic Topup Data Across All Processing Phases

Category: Safety property

Correctness of the Topup data in `MsgTopupTx` must be validated during side transaction handlers and the post-transaction handler, before emitting `EventTypeTopup`. Data present in the message must be equal to the data present in the `txReceipt` retrieved from the Ethereum staking smart contract for the `txHash`.

Violation consequences

- Invalid data processed in one of the phases could lead to invalid topup processed on the Heimdall chain.

Threats

- Side tx handlers voting `YES` on invalid data.
- Post tx handlers processing invalid topup action and updating the state.

Conclusion

The property holds.

`SideHandleTopupTx` contains validations that make sure that information provided in `MsgTopupTx` matches with actual transaction information executed on L1 and if any of the parameters is differing from L1 transaction side transaction handler casts `NO` vote which (if final voting result is also `NO`) stops `PostHandleTopupTx` from further processing of topup.

Property TU-10: Replay Protection from Duplicate Ethereum Events in x/topup Module

Category: Safety property

The replay protection mechanism for `x/topup` messages ensures that transactions already processed by post-transaction handlers cannot be re-executed.

Violation consequences

- Possibility of executing one `x/topup` `MsgTopupTx` message multiple times can impact the validator's balances and dividend account amounts.

Threats

- Sequence created for each processed `x/topup` `MsgTopupTx` message is not unique.
- Sequence is not stored in the state for the `x/topup` post tx handler.

Conclusion

The property holds.

Calculation of sequence using block number, default log index unit (ref) and log index ensures that sequence is unique.

Multiple executions of the same transaction are prevented by keeping track of its sequence presence in keeper (ref1, ref2). At the end of `PostHandleTopupTx`, after successful execution, sequence is being stored in the state.

Property TU-11: Correct Inclusion of Dividend Balances in Checkpoint's AccountRootHash

Category: Safety property

`AccountRootHash` containing hash of all dividend accounts current balance is correctly stored in proposed checkpoint.

Violation consequences

- Incorrectly hashed dividend account balances may result in inaccurate top-up balances for validators on Ethereum. This discrepancy can lead to various issues, including potential liveness failures—especially if validators are left with zero top-up balances on Heimdall, preventing them from participating in consensus or other chain operations (due to misleading information on Ethereum)

Threats

- Not all dividend accounts are included in the `AccountRootHash`.
- The `AccountRootHash` is computed based on outdated or incorrect dividend account state.

Conclusion

The property holds.

When creating `MsgCheckpoint` to be broadcasted to Heimdall, bridge process fetches (ref1, ref2) dividend account root which is done by utilizing `x/topups` `GetDividendAccountRootHash` query function (ref). It fetches all dividend accounts and then uses them to calculate dividend account root hash (ref1, ref2). Note that used `NewTreeWithHashStrategy` function is externally defined in this repository. It is visible here that all dividend accounts are used when creating merkletree.

Inside `x/checkpoint` `msg_server` `Checkpoint` function when comparing actual dividend account root hash with one stated in MsgCheckpoint same steps are used as in `GetDividendAccountRootHash` to calculate dividend account root hash using current state.

Property TU-12: Correct AccountRootHash Eventually Bridged to L1

Category: Liveness property

Correct `AccountRootHash` is eventually bridged as part of the checkpoint to the L1 Ethereum root chain.

Violation consequences

- Chain liveness (issues with finalization).

Threats

- Different hashing functions are used on Heimdall and Ethereum when computing or verifying the `AccountRootHash`, causing checkpoint verification failures.

Conclusion

The property holds.

As mentioned in *Property TU-11: Correct Inclusion of Dividend Balances in Checkpoint's AccountRootHash*, both the bridging process and the `x/checkpoint` `msg_server` use identical steps and functions when calculating dividend account root hash (ref1, ref2).

Hashing and merkle tree creation is done using this external function that passes keccak256 hashing strategy. `DividendAccount` also implements its `CalculateHash` using keccack256.

These functions (ref1, ref2, ref3) are used to send checkpoint to root chain, packing dividend account root hash extracted from `MsgCheckpoint` (already been validated) among the other information (ref1, ref2) that is being sent.

Checkpoint is submitted (ref1, ref2, ref3) using this function on L1. On L1, inside `submitCheckpoint` `accountHash` is passed to `stakeManager.checkSignatures` function (ref) where it is used to update `accountStateRoot` (ref1, ref2). This concludes that no additional computing or verification of dividend account root hash is done on L1 which means that there are not any conflicting hashing functions on L1 and L2 which would cause checkpoint verification failures.

Property TU-13: x/topup Module Requires Mint/Burn Privileges

Category: Liveness property

`x/topup` module account must have minting and burning privileges.

Violation consequences

- If the `x/topup` module account lacks minting or burning privileges, it may be unable to properly process and reflect top-up inflows (from Ethereum) or withdrawals (to Ethereum).

Threats

- Incorrect initialization of the `x/topup` module account.
- The module is deployed or upgraded without mint/burn rights.
- Mint/burn privileges are accidentally or maliciously revoked via parameter/gov changes.
- Blocked `x/topup` module account.

Conclusion

The property holds.

Heimdall application set module account permissions of `x/topup` module to grant `Burn` and `Mint` actions and initializes account keeper with these module permissions. Bank module is setup with no sending or minting restrictions blocking transfer of funds or minting coins in `x/topup` module.

Mint and burn privileges of module account in `x/auth` module can't be altered by governance. Transfer of coins (send/burn) cannot be prevented by bank's `SendEnable` parameter as `x/topup` module is operating on bank's keeper instance where the related check is done on message server level.

However `x/topup`'s own msg handler `HandleTopupTx()` for topup transactions fails if `SendEnable` is not `true` for the default denom `pol` (see `x/topup/keeper/msg_server.go` L45) and would prevent top-up inflows.

Property TU-14: No Restrictions for MsgTopupTx User Account Field

Category: Liveness property

There must be no restrictions in user being equal to the proposer validator address.

Violation Consequences

- If there are specific user accounts permitted to perform MsgTopupTx - we could end up in a situation where a validator does not have a sufficient amount of balance to cover the tx fees.

Threats

- Limitations in user addresses (validator could be a user initiating the `MsgTopupTx`).
- User is among blocked addresses (cannot receive the remaining funds and blocks the entire top-up mechanism).

Conclusion

The property holds.

There is no restriction in the processing of `MsgTopupTx` messages put in place that requires the proposer being equal to user.

Property TU-15: Top-Up Amount Must Meet Minimum Fee Requirement

Category: Liveness property

`MsgTopupTx` top-up amount must be equal to at least `DefaultFeeWantedPerTx` amount.

Violation consequences

- If the top-up amount is below the `DefaultFeeWantedPerTx`, the validator may not have sufficient funds on Heimdall to cover transaction fees.

Threats

- A bug or missing validation allows sending `MsgTopupTx` with value below the configured threshold.
- Misconfigured `DefaultFeeWantedPerTx` set too low or too high, disrupting normal validator funding behavior).
- Spamming the network with small top-up messages below fee requirements to waste resources or exploit edge cases.

Conclusion

The property holds.

Event handler and side-message handler for `MsgTopupTx` messages check that the topup amount of the received message is not less than the amount defined in `x/auth` modules `DefaultFeeWantedPerTx` variable. The value of default fee wanted did not change in value from Heimdall v1 apart the denom change from `matic` to `pol`.

Property TU-16: Default Fee Transferred to Proposer; Remainder to User

Category: Safety property

From the top-up amount the `DefaultFeeWantedPerTx`is removed and transferred to the `Proposer` of the message and the remainder is transferred to the `User` of the message.

Violation consequences

- The user assumes that only the minimum top-up amount - `DefaultFeeWantedPerTx` - will be transferred to the proposer. If this assumption is violated, the user may unintentionally transfer more funds than expected, leading to potential loss.

Threats

- The transfer logic misroutes the full amount to either the proposer or the user.
- Malicious `MsgTopupTx` bypasses enforcement of the default fee deduction.

Conclusion

The property holds.

The handler of the post side transaction for top-up messages transfers the `DefaultFeeWantedPerTx` amount from the user account to the proposer account after the full top-up amount was added to the user's account. This is done only for side transactions with result `Vote_VOTE_YES` and returns with an error otherwise.

Property TU-17: Withdrawal Cannot Exceed Validator Balance

Category: Safety property

It is impossible to withdraw more than the proposer validator holds on the account.

Violation consequences

- Allowing withdrawals greater than the validator's current balance on Heimdall could lead to inconsistencies between Heimdall and Ethereum states and potential liveness issues.

Threats

- Missing validation allows over-withdrawal from a validator's account.
- Bugs in state tracking lead to incorrect balance assumptions.

Conclusion

The property holds.

`WithdrawFeeTx()` uses direct API call `SendCoinsFromAccountToModule()` of Cosmos SDK's bank module to transfer the withdraw amount from the proposer's account to the `topup` module's account. `SendCoinsFromAccountToModule` validates that the `Denom` is valid, the amount is not negative and the resulting balance is not negative.

State of all account balances are tracked in Cosmos SDK's bank module.

Property TU-18: Withdrawal Amount Must Be Non-Negative

Findings: Early Validation Missing for Negative Withdrawal Amounts in MsgWithdrawFeeTx

Category: Safety property

Amount burned and stored under dividend account fee for the proposer validator must not be negative.

Threats

- Withdrawal message bypasses checks (or there are no checks) allowing negative amounts.
- Improper handling of subtraction leads to large unintended values.

Conclusion

The property holds.

While it is possible to send `MsgWithdrawFeeTx` with negative number as `Amount` it will fail validation when trying to `SendCoinsFromAccountToModule` (ref). It will fail as soon it tries to call `SendCoins` on `subUnlockedCoins` (ref1, ref2, ref3, ref4). More details in reported informational issue.

Overflows are also prevented using safe additions and subtractions (e.g. ref).

Property TU-19: Validator Withdrawn/Reduced Balance amount Must Equal Dividend Fee Increase

Category: Safety property

The decrease of the withdrawn amount from the validator's account must be the same as the increase of the dividend account fee stored.

Violation consequences

- Validators top-up amounts are misaligned across Ethereum and Heimdall state

Threats

- Incorrect implementation of the `MsgWithdrawFeeTx`.

Conclusion

The property holds.

Same `coins` variable that is initialized using `amount` from `MsgWithdrawFeeTx` (ref) is used as parameter for sending coins to module, burning them and then adding fee (uses same amount that was used to initialize `coins`) to dividend account. There are no side effects changing `coins` amount (or `amount` variable) during calls of these functions.

Property TU-20: Full Withdrawal on Zero Withdrawal Amount in MsgWithdrawFeeTx

Category: Safety property

If the withdrawal amount is not specified (i.e., zero), the entire balance of the validator's account is added and stored in the validator's dividend account.

Violation consequences

- Unexpected results during withdrawals could result in increased transaction costs for the proposer, who may initiate multiple withdrawal transactions.

Threats

- Incorrect implementation of the `MsgWithdrawFeeTx` (incorrect processing of amount equal to 0).

Conclusion

The property holds.

If `amount` stated in the `MsgWithdrawFeeTx` is equal to zero, `amount` to be withdrawn is set to maximum spendable amount by account (ref). If amount turns out to be zero, insufficient funds error is being returned (ref), otherwise `amount` is sent from proposers account to module account (topup), burned and at the end same `amount` is added to validator's dividend account.

Property TU-21: Fee Amount in Dividend Account - Strictly Increasing with Withdrawals Performed

Category: Safety property

Dividend account entries in state are expected to be processed ONLY when a withdrawal is triggered on Heimdall. All the additional accounting logic is located in Ethereum staking smart contract and assumed to be working correctly (not an artifact within the current audit).

Violation consequences

- Assumptions on the Ethereum side would be broken.

Threats

- implementation issues: actions upon `MsgCpAck` processing, validator exiting/joining the validator set (`MsgValidatorExit`/`MsgValidatorJoin`).

Conclusion

The property holds.

It is assumed that the L1 contracts will correctly handle updates to the `dividendAccounts` balances. As discussed with the Polygon team, this logic resides in the staking smart contract, which is outside the scope of this audit.

`SetDividendAccount` is the only function directly writing to `dividendAccounts` map and it is used during genesis (ref) and in `AddFeeToDividendAccount` (ref).

`AddFeeToDividendAccount` is only called from `WithdrawFeeTx` (ref).

All other references to `dividendAccounts` are for reading, querying, or test setup/validation.

Property HEIM-1: Correct wiring of x/gov module in Heimdall v2 app

Category: Liveness property

The wiring of the `x/gov` module in the Heimdall v2 app must follow Cosmos SDK standards to ensure compatibility with both legacy and new governance proposals. This includes properly registering the message handlers and ensuring that the `MsgServiceRouter` is correctly configured to route proposal messages. For legacy proposals, specific transaction commands must be registered to ensure proper processing. Furthermore, the `SetLegacyRouter(govRouter)` must be set to support routing for legacy proposals.

Violation consequences

- Incorrect wiring of the `x/gov` module could lead to issues with process governance proposals. Without proper governance functionality, key network upgrades, parameter changes, and other governance proposals would be impossible to process.

Threats

- Legacy proposals are not supported:
  - The relevant transaction commands are not registered using `AddTxCommands(cmd)` to route the `submit-legacy-proposal` message to the appropriate handler.
  - The `SetLegacyRouter(govRouter)` is not correctly configured to support the routing for legacy proposals.
- Wiring Is Not Performed Correctly:
  - The message handler is not registered in the `MsgServiceRouter` to ensure the correct routing and handling of governance proposal messages.

Conclusion

The property holds.

The following Heimdall v2 modules have parameters that must be updated only by the governance module (gov proposals): `x/bor`, `x/checkpoint` and `x/milestone` and their authority is correctly initialized to `authtypes.NewModuleAddress(govtypes.ModuleName).String()` (code ref).

The analysis concluded the governance module is integrated correctly into the Cosmos SDK app with the following considerations:

- Proposal routing correctly routes proposal messages to the relevant handlers (code ref) (`govv1beta1.ProposalHandler`, `params.NewParamChangeProposalHandler`).
- Backward Compatibility: The use of the legacy router (`SetLegacyRouter`) ensures compatibility with older governance versions (code ref). The following commands are registered (code ref), in order to support legacy proposal submission.
- Parameter and Consensus Management is correctly handled. The use of `ParamsKeeper` and `ConsensusParamsKeeper` ensures that governance parameters and consensus parameters are properly managed (code ref).
- Governance Hooks mechanism for custom hooks (`govtypes.NewMultiGovHooks`) is set up though it appears the actual hook behavior is not defined for Heimdall v2 app (code ref).

Threat GT-1: Development & testing practices and quality of code

Findings: x/checkpoint Various Minor Code Improvements, Improve Error Handling in x/checkpoint Queries, x/milestone Various Minor Code Improvements, Improve Error Handling in x/stake Queries, x/stake Various Minor Code Improvements

Threats

- Error handling:
  - Error codes are correctly defined in protocol.
  - Returning a meaningful error message when a tx/query cannot be executed.
- Correct logging
- No code duplication
- Optimization improvements
- Query performances and scalability:
  - Implement pagination for queries that return large data set results (Include `PageRequest` and `PageResponse` in query messages; apply pagination in gRPC query handlers; CLI should include pagination flags).
  - Efficient query execution to avoid unnecessary resource consumption.
  - Avoiding unbounded loops or excessive data loading.
- CLI exposure for queries and transactions:
  - CLI exposed queries implementing pagination should include pagination flags.
- Test coverage: Assessing whether unit and integration tests cover edge cases, error handling, and expected query results.

Threat GT-2: Arithmetic operations

Threats

Consider issues such as overflows, underflows, and overlaps in arithmetic operations, particularly in Go (Golang). Ensure that appropriate safeguards and validations are in place to prevent these errors during execution.

Threat GT-3: DoS attacks due to unbounded iteration

Threats

- Unbounded loops: Avoid iterating over large or unlimited maps, lists, or key ranges in a single transaction.
- Pagination: Always implement and enforce pagination for queries and state iterations.
- Nested iterations: Watch for nested loops over state that could lead to quadratic time.
- Accumulator growth: Prevent unbounded accumulation of results or logs within a single transaction.

Threat GT-4: Non-determinism sources

Threats

- Time-based logic: Avoid use of `time.Now()` or block time for logic affecting state transitions.
- Map iteration: Ensure deterministic iteration over maps (e.g., sort keys before iteration).
- Randomness: Do not use `math/rand` or other non-deterministic sources in state-altering code.
- I/O or external calls: Avoid logic relying on external APIs or filesystem that can produce non-deterministic results.
- Inconsistent serialization: Ensure stable marshaling/unmarshaling (e.g., Protobuf vs Amino differences).

Threat GT-5: Protobuf definition implementation issues

Threats

- Field handling: Ensure correct use of fields (renaming, deprecation, type changes).
- Dynamic fields: Verify dynamic fields in URLs are enclosed in curly braces.
- Query path conflicts: Ensure unique prefixes to avoid conflicts between query routes.
- Nullable fields: Use gogoproto.nullable = false for repeated fields to prevent runtime errors.
- Deprecated fields: Ensure deprecated fields are removed or marked appropriately to avoid unintended usage.
- Backward compatibility: Ensure changes maintain compatibility or versioning is clear. omit

---

Findings

Finding	Type	Severity	Status
Governance Refund of Deposits	Implementation	Informational	Acknowledged
Governance Tally Results	Implementation	Informational	Acknowledged
Governance Expedited Proposal Deposits	Protocol	Informational	Acknowledged
Ante Decorator Deducting Signature Verification Fees	Implementation	Low	Acknowledged
Improper Validation in x/auth Module Allows for Negative Transaction Fees	Implementation	Medium	Acknowledged
Block Proposer Tracking Failure in Heimdall v2 Causes Reward Distribution Malfunction	Implementation	High	Acknowledged
TxSigLimit Validation Does Not Enforce Heimdall's Lack of Multisig Support	Implementation	Informational	Acknowledged
Miscellaneous Recommendations	Implementation	Informational	Acknowledged
Redundant Height Check in PreBlocker Vote Extension Activation Logic	Implementation	Informational	Acknowledged
A Malicious Validator Introduces an Extremely Long Vote Extension Preventing Transactions from Being Added in a Block	Implementation	High	Acknowledged
A Malicious Validator Introduces a Bogus Vote Extension that Halts the Chain	Implementation	High	Acknowledged
x/checkpoint Various Minor Code Improvements	Implementation	Informational	Acknowledged
Missing Validation for BorChainId Field in MsgCheckpoint	Implementation	Low	Acknowledged
GetCurrentProposer and GetProposers are Implemented as x/checkpoint Queries	Implementation	Informational	Acknowledged

Finding	Type	Severity	Status
Missing Validations for x/checkpoint Query Inputs Before Processing	Implementation	Medium	Acknowledged
Improve Error Handling in x/checkpoint Queries	Implementation	Informational	Acknowledged
Inconsistent Checkpoint Handling When Multiple MsgCheckpoint Messages Exist in a Single Block	Implementation	High	Acknowledged
Missing Checkpoint Length Validations in MsgCheckpoint	Implementation	Low	Acknowledged
Insufficient Validation and Authorization in MsgCpAck Handling	Implementation	High	Acknowledged
Unused and Unvalidated Fields in MsgCpAck (TxHash and LogIndex)	Implementation	Informational	Acknowledged
Potential Execution of MsgCpAck and MsgCpNoAck in the Same Block Height	Protocol	Informational	Acknowledged
Unclear Expectations If AvgCheckpointLength Equals MaxCheckpointLength	Protocol	Low	Acknowledged
Checkpoint Module State is not Entirely Exported and Initialized	Implementation	Low	Acknowledged
Incomplete Data Validation in Checkpoint ValidateGenesis Function	Implementation	Low	Acknowledged
Milestone Module State is not Entirely Exported and Initialized	Implementation	Low	Acknowledged
Milestone Proposals Lack Proper Validation of Block Hashes	Implementation	Informational	Acknowledged
Malicious Proposer Can Inject Invalid Milestone	Implementation	High	Acknowledged
x/milestone Various Minor Code Improvements	Implementation	Informational	Acknowledged
Proposer Can Submit Milestone Without Block Hashes	Implementation	Informational	Acknowledged
Missing Input Data Validations in x/milestone Queries	Implementation	Informational	Acknowledged
Panic Caused by Invalid Validator Public Key in EndBlocker	Implementation	Informational	Acknowledged
TopUp - Unsafe Queries	Implementation	Informational	Acknowledged

Finding	Type	Severity	Status
Missing Validation for Negative FeeToken Amounts During Transfers	Implementation	Informational	Acknowledged
Topup - Query Input Validation Missing	Implementation	Informational	Acknowledged
Validator Object Retains Non-Zero Voting Power After Exit	Implementation	Low	Acknowledged
Unsafe x/stake IsStakeTxOld Query	Implementation	Informational	Acknowledged
Missing Input Data Validations in x/stake Queries	Implementation	Informational	Acknowledged
Improve Error Handling in x/stake Queries	Implementation	Informational	Acknowledged
Implemented Validate() Methods Instead of ValidateBasic() for x/stake Messages	Implementation	Low	Acknowledged
x/stake Various Minor Code Improvements	Implementation	Informational	Acknowledged
Lack of Authorization Checks for From Field in x/stake Messages	Protocol	Medium	Acknowledged
Stake Module State is not Entirely Exported and Initialized	Implementation	Low	Acknowledged
Heavy Reliance on L1 Event Logs Without Sufficient Local Validation in x/stake Messages	Implementation	Informational	Acknowledged
Incomplete Data Validation in Stake ValidateGenesis Function	Implementation	Low	Acknowledged
Incorrect SignerPubKey Validation Prevents Exited Validators to Rejoin	Implementation	Low	Acknowledged
Early Validation Missing for Negative Withdrawal Amounts in MsgWithdrawFeeTx	Implementation	Informational	Acknowledged

Governance Refund of Deposits

ID	IF-FINDING-001
Severity	Informational
Impact	2 - Medium
Exploitability	0 - None
Type	Implementation
Status	Acknowledged

Involved artifacts

- Polygon fork of Cosmos SDK v0.1.16-beta-polygon: x/gov/keeper/deposit.go

Description

`DistributeAndDeleteDeposits()` should distribute deposits across active validators equally.

If the list of validators received from staking module contains a validator where its account address can't be determined the deposits would only be refunded to a subset of valid validators.

Problem Scenarios

The list of validators is created by iterating over validators using the external iterator function `IterateCurrentValidatorsAndApplyFn()` of the staking module. The iterator calls the passed callback function on each element and stops iteration in case of an error from getting the validator set or `true` returned by the callback function indicating to stop iteration.

Current implementation of this callback function returns `true` if the account address of an validator can't be determined `AccAddressFromHex()`. This would result in stopping the iteration at that point and only those validators up to that point would be part of the list of those being refunded.

Recommendation

We recommend to encounter all valid validators and ignore invalid ones when determining the list of validators the deposits should be distributed to. The required change to do so is to return `false` in the callback function of the iterator, requesting the iterator not to stop iteration and to continue.

Governance Tally Results

ID	IF-FINDING-002
Severity	Informational
Impact	3 - High
Exploitability	0 - None
Type	Implementation
Status	Acknowledged

Involved artifacts

- Polygon fork of Cosmos SDK v0.1.16-beta-polygon: x/gov/keeper/tally.go

Description

The tally result for proposals in governance depend on the total voting power which is used to decide if quorum is reached. Current implementation determines total voting power by creating the list of current validators when tallying votes for a proposal. This list of validators might be incomplete and contain only a subset of valid validators in case of an invalid validator. This would lead to an invalid calculation of total voting power and allow this subset of validators to reach quorum with their votes.

Problem Scenarios

The Tally() function in governance module determines the tally result for a proposal. Quorum needs to be reached to determine decide if a proposal fails or passes. The quorum depends on the total voting power which is calculated form the list of current validators. This list of validators is created when iterating over a set of validators using the external iterator function `IterateCurrentValidatorsAndApplyFn()` of the staking module. The iterator calls the passed callback function on each element (validator) and stops iteration in case of an error from getting the validator set or if the callback function requests to stop iteration by returning `true`.

Current implementation of this callback function returns `true` if the account address of an validator can not be determined `ValidatorAddressCodec()`. This would result in stopping the iteration at that point and only validators processed so far would be considered as the current active validator set. This list is used to determine the total voting power and even a single validator can reach quorum and decide the tally result of the proposal even if the majority of validators would have voted differently.

Exploitability strongly depends on the external module implementing the iterator. In case of heimdall application this can be considered `Low` or even `None` as it would imply a validator with an invalid validator address in the active validator set.

Recommendation

Invalid validators should be ignored in the tally function. The iterator callback function should not request to stop iteration and return `false` in case of errors getting the validator address (see here).

Governance Expedited Proposal Deposits

ID	IF-FINDING-003
Severity	Informational
Impact	1 - Low
Exploitability	
Type	Protocol
Status	Acknowledged

Involved artifacts

- `x/gov/abci.go` `EndBlocker()`

Description

Expedited proposals loose all deposits when converted to standard proposal. This is an issue for depositors as their proposals are not refunded back to them (as it is handled as a 'rejected proposal') but distributed equally across validators.

Default deposit management for expedited proposals should not touch deposits when it will be converted to a standard proposal.

Problem Scenarios

When an expedited proposal does not get enough votes during the expedited voting period it will be converted to a standard proposal and voting period will be extended. In default SDK implementation this will not update the deposits done so far where in Polygon's fork of the SDK the deposits are refunded equally across all validators which might result in deposit loss for some of them.

Recommendation

We recommend to adapt to Cosmos SDKs default behaviour for expedited proposals (see EndBlocker() Line 139-168).

Ante Decorator Deducting Signature Verification Fees

ID	IF-FINDING-004
Severity	Low
Impact	1 - Low
Exploitability	1 - Low
Type	Implementation
Status	Acknowledged

Involved artifacts

- Polygon fork of Cosmos SDK v0.1.16-beta-polygon:
  - x/auth/ante/sigverify.go
  - x/auth/types/auth.pb.go

Description

Because Heimdall deducts fixed fee per transaction that is calculated using modules parameters (ref1, ref2, ref3) using `DeductFeeDecorator`, gas consumption in `ConsumeTxSizeGasDecorator` is disabled but despite that `SigGasConsumeDecorator` consumes gas required for signature verification.

Problem Scenarios

Depending on signature verification algorithm that is used for transaction signing, different transaction fees will be deducted instead of same fixed fee for every transaction.

Recommendation

Disable `SigGasConsumeDecorator` as it is done for `ConsumeTxSizeGasDecorator` or set all parameters related to signature verification gas consumption (ref) to zero and make sure they stay zero by enforcing validity checks during genesis and parameter updates.

Improper Validation in x/auth Module Allows for Negative Transaction Fees

ID	IF-FINDING-005
Severity	Medium
Impact	3 - High
Exploitability	1 - Low
Type	Implementation
Status	Acknowledged

Involved artifacts

- Polygon fork of Cosmos SDK v0.1.16-beta-polygon:
  - x/auth/types/params.go

Description

Validation of `TxFees` `x/auth` parameter allows `TxFees` to be set to a negative number during genesis or parameters update.

Problem Scenarios

During genesis or update of parameters through governance `TxFees` could be set to the negative value.

If `TxFees` is set to a negative value `DeductFeeDecorator` will produce an error during fee validity check before deducting fees and make transactions impossible to finish. Once in this state, the chain would enter a deadlock: since all transactions would fail, it would be impossible to fix the parameter value using `UpdateParams` function.

Recommendation

Add explicit non-negative validation to the `validateTxFees` function.

Block Proposer Tracking Failure in Heimdall v2 Causes Reward Distribution Malfunction

ID	IF-FINDING-006
Severity	High
Impact	2 - Medium
Exploitability	3 - High
Type	Implementation
Status	Acknowledged

Involved artifacts

- Polygon fork of Cosmos SDK v0.1.16-beta-polygon:
  - x/auth/keeper/keeper.go
- Heimdall-v2:
  - app/app.go
  - app/abci.go

Description

The migration from Heimdall v1 to v2 introduced a flaw in the block proposer tracking mechanism. In Heimdall v2, the `BeginBlocker` function has been modified to no longer receive the `abci.RequestBeginBlock` parameter which previously contained the proposer's address. Instead, it attempts to retrieve the current block proposer from the app's KVStore using the newly introduced `GetBlockProposer` function.

However, there is no implementation that writes the proposer's address to the KVStore before `BeginBlocker` is called. This oversight causes `GetBlockProposer` to consistently return false for its second return parameter, indicating that the proposer's address is not found. As a result, the block proposer tracking logic in `BeginBlocker` is always skipped, which subsequently causes the reward distribution logic in `EndBlocker` to be omitted.

Additionally, the current implementation inefficiently converts the proposer's address from `sdk.AccAddress` to `string` and then back to `sdk.AccAddress` unnecessarily.

Problem Scenarios

Validators do not receive their expected rewards for block proposing, as the reward distribution logic in `EndBlocker` is never executed due to the missing proposer information.

Recommendation

Move the block proposer setting logic from `BeginBlocker` to `PreBlocker`, where the proposer's address is directly available from parameters.

Remove redundant address conversions in the current implementation. If address validation is required, implement it once rather than converting back and forth.

TxSigLimit Validation Does Not Enforce Heimdall's Lack of Multisig Support

ID	IF-FINDING-007
Severity	Informational
Impact	1 - Low
Exploitability	0 - None
Type	Implementation
Status	Acknowledged

Involved artifacts

- Polygon fork of Cosmos SDK v0.1.16-beta-polygon:
  - x/auth/types/params.go

Description

Heimdall does not support multisig, but the validation for the `TxSigLimit` parameter in `validateTxSigLimit` function does not enforce that this value must be set to 1.

Problem Scenarios

While the probability of this happening is low, if `TxSigLimit` is set to a value different than 1 through either genesis initialization or a governance parameter update, transactions using multisig would not be rejected if `TxSigLimit` is not set to 1 (ref) and enable execution of untested code.

Although there is an explicit check inside `SigGasConsumeDecorator` to enforce a single signature, it is strongly suggested to add a check in the parameter validation for `TxSigLimit` to ensure it is 1 and prevent multisig as `SigGasConsumeDecorator` might be removed from the list of ante handlers.

Recommendation

Enhance the `validateTxSigLimit` function to explicitly enforce that the value must be exactly 1.

Miscellaneous Recommendations

ID	IF-FINDING-008
Severity	Informational
Impact	0 - None
Exploitability	0 - None
Type	Implementation
Status	Acknowledged

Involved artifacts

- app/abci.go
- app/vote_ext_utils.go
- `side_msg_server.go` file for all modules
- `msg_server.go` file for all modules

Recommendation

- There are cases where a side handler returns `Vote_VOTE_NO` when the RPC call fails (e.g., `MsgCpAck`) while in other cases a side handler returns `Vote_UNSPECIFIED` (e.g., `MsgProposeSpan`). It is not clear why this discrepancy exists so it would be nice to clarify.
- Although logging is performed in post handlers themselves, some logging might be useful here in case a post handler fails to execute. Additional, logging (or even failing since this would imply something is off with the setup) might be useful in case more than one messages with post handlers reside in a transaction here.
- In multiple places, the following comment `check if incoming tx is older` appears (e.g., `MsgStakeUpdate`). However, the follow-up check does not check whether the transaction is older or not but whether this event has already been processed (i.e., prevent replay attacks). The comment can be rephrased to capture this.
- Slashing or jailing can potentially be introduced for CometBFT block double signing, but as well for misbehavior on vote extensions (e.g., a validator votes twice for a transaction as part of its a vote extension) to deter Byzantine behavior.
- The `getPreviousBlockValidatorSet` is invoked three times, in `ValidateVoteExtensions`, in `checkNonRpVoteExtensionSignatures`, and `getMajorityNonRpVoteExtension`. If the returned validator set is guaranteed to remain consistent across these calls, the result could be cached and reused for efficiency. Otherwise, diverging results may indicate an inconsistency in how the validator set is derived.

Redundant Height Check in PreBlocker Vote Extension Activation Logic

ID	IF-FINDING-009
Severity	Informational
Impact	0 - None
Exploitability	0 - None
Type	Implementation
Status	Acknowledged

Involved artifacts

- app/abci.go
- app/vote_ext_utils.go

Description

In the `PreBlocker` method of `HeimdallApp`, two separate height-based checks are performed to determine whether vote extensions should be processed:

```
if err := checkIfVoteExtensionsDisabled(ctx, req.Height+1); err != nil {
  return nil, err
}

// ...

if req.Height <= retrieveVoteExtensionsEnableHeight(ctx) {
  if len(extVoteInfo) != 0 {
    logger.Error("Unexpected behavior, non-empty VEs found in the initial height's
↪  pre-blocker", "height", req.Height)
    return nil, errors.New("non-empty VEs found in the initial height's pre-blocker")
  }
  return app.ModuleManager.PreBlock(ctx)
}
```

The first check ensures vote extensions are enabled from the correct height onward (`voteExtensionsEnableHeight + 1`). This is consistent with the upgrade plan (e.g., Heimdall v1 ends at height 100, and v2 with VEs begins at 101).

However, the second check performs a similar condition, but includes equality (`<=`). This results in overlapping logic that is effectively redundant when `PreBlocker` is only expected to run on blocks *after* vote extensions are active.

Problem Scenarios

This duplication adds unnecessary complexity and may cause confusion during future maintenance. It does not introduce a correctness bug but may falsely imply that `PreBlocker` needs to handle height = `voteExtensionsEnableHeight`, which isn't necessary based on the current protocol assumptions.

Recommendation

Refactor `PreBlocker` to rely solely on the initial height check using `checkIfVoteExtensionsDisabled(ctx, req.Height+1)`. Once this check passes, it can be assumed that vote extensions must be present and valid, and the redundant `if req.Height <= voteExtensionsEnableHeight` block can be removed.

This simplifies the logic without changing behavior.

A Malicious Validator Introduces an Extremely Long Vote Extension Preventing Transactions from Being Added in a Block

ID	IF-FINDING-010
Severity	High
Impact	3 - High
Exploitability	2 - Medium
Type	Implementation
Status	Acknowledged

Involved artifacts

- app/abci.go
- app/vote_ext_utils.go

Description

A malicious validator can introduce an extremely large vote extension `lv` in every (i.e., for every round and every height) of its `pre-commit` messages by introducing an extremely large number of `sideTxResponses`.

Problem Scenarios

As a result, an upcoming validator proposing a block in `PrepareProposal` includes `lv` in `txs[0]` and if `lv` is too big (i.e., `size(lv) > req.MaxTxBytes`), then the condition `totalTxBytes+len(proposedTx) > int(req.MaxTxBytes)` evaluates to `true` and hence no transaction is added to the proposed block. Note that `validateSideTxResponses` does not contain a check on the number of `sideTxResponses`, so a malicious validator can include `sideTxResponses` for transactions that do not exist.

Recommendation

Introduce an additional check in `validateSideTxResponses` that bounds the number of `sideTxResponses` to guarantee that we have space left (besides the vote extensions) in a block to add transactions.

A Malicious Validator Introduces a Bogus Vote Extension that Halts the Chain

ID	IF-FINDING-011
Severity	High
Impact	3 - High
Exploitability	2 - Medium
Type	Implementation
Status	Acknowledged

Involved artifacts

- app/abci.go
- app/vote_ext_utils.go

Description

A malicious validator can introduce a bogus vote extension in a block that passes `ValidateVoteExtensions` but later on, this vote extension makes `PreBlocker` fail hence halting the chain.

This can occur because `PreBlocker` calls `tallyVotes` and returns an `error` if `tallyVotes` returns an `error`. However, `tallyVotes` calls `aggregateVotes` and in `aggregateVotes`, we perform a block-hash consistency check that checks that all vote extensions have the exact same `BlockHash`. However, such a block-hash check does not appear in `ValidateVoteExtensions`.

Problem Scenarios

Because of the above, a malicious validator `V` can propose a block `B` that in `B.txs[0]` contains `V`'s vote extension that does not have a correct `BlockHash`. Because `ValidateVoteExtensions` does not perform a block-hash check, block `B` is successfully processed (in `NewProcessProposalHandler`) by other validators and `B` is eventually decided upon. Afterwards, when we execute `PreBlocker` for `B`, we call `tallyVotes` that calls `aggregateVotes` that returns an `error` because the block-hash consistency check fails. Hence `tallyVotes` ends-up returning an `error` and `PreBlocker` also returns an `error`. As a result, Cosmos SDK `FinalizeBlock` returns an error (due to `preBlock` returning an `error`) which can lead to a chain halt. Note that `V`'s vote extension with the bogus block hash is not verified during `VerifyVoteExtensionHandler` because the malicious validator directly inject this bogus vote extension as part of the block that `V` proposes and not as part of its `pre-commit` message.

Recommendation

Introduce the block-hash check in `ValidateVoteExtensions` as well (not only have it in `VerifyVoteExtensionHandler`) and reject blocks (during `ProcessProposal`) that contain invalid vote extensions.

x/checkpoint Various Minor Code Improvements

ID	IF-FINDING-012
Severity	Informational
Impact	0 - None
Exploitability	0 - None
Type	Implementation
Status	Acknowledged

Involved artifacts

- x/checkpoint/keeper/
- x/checkpoint/types/

Description

Here is the list of aesthetic and minor code improvements and issues around logging found during the code inspection of the `x/checkpoint` Heimdall v2 module. They do not pose a security threat nor do they introduce an issue, but the following suggestions are shared to improve the code readability, keep consistency, optimize, and improve logging.

- Logging `PostHandleMsgCheckpointAck` error log created with wrong data: instead of end block - start block is logged (code ref).
- Bridge implementation related:
  - bridge/processor/checkpoint.go
  - redundant call of `shouldSendCheckpoint` function (code ref), since the same function was executed (code ref) prior to the `CreateAndSendCheckpointToRootchain` function.
- Call `HasCheckpointBuffer` (code ref) from `GetCheckpointFromBuffer` (code ref) instead of this code section (code ref)

```
exists, err := k.bufferedCheckpoint.Has(ctx)
  if err != nil {
    k.Logger(ctx).Error("error while checking for existence of the buffered checkpoint in
↪  store", "err", err)
    return checkpoint, err
  }
```

- msg_server.go#L68 and side_msg_server.go#L199: The condition `lastCheckpoint.EndBlock > msg.StartBlock` is redundant, as it is fully covered by the more specific check `lastCheckpoint.EndBlock + 1 != msg.StartBlock`. For example, if `lastCheckpoint.EndBlock = 10`, the first condition allows `StartBlock = 11, 12, 13,...`, while the second restricts it strictly to `11`.
- msg_server.go#L177 and side_msg_server.go#L319: The check `msg.StartBlock == checkpointObj.StartBlock` is unnecessary here, since the same condition is already verified earlier in the same flow.
- msg_server.go#L85 and side_msg_server.go#L216: Consider replacing `err.Error() == types.ErrNoCheckpointFound.Error()` with a safer and more idiomatic comparison using `errors.Is(err, types.ErrNoCheckpointFound)`.
- side_msg_server.go#L227: A potential error returned by `GetCheckpointFromBuffer` is not handled. If `err != nil` but `doExist == true`, the flow may continue incorrectly. This can lead to unexpected logic execution on an invalid result.
- msg.go#L64: The check `msg.EndBlock == 0` is redundant. It is already covered by `startBlock >= endBlock` and by the fact that both fields are of type `uint64`.

- msg.go#L196: The check for `len(msg.RootHash) == common.HashLength` is missing. This may allow incorrectly sized root hashes to pass validation.

Problem Scenarios

Findings listed above could not introduce any issues, they are suggestions for code improvements.

Recommendation

As explained in the *Description* section.

Missing Validation for BorChainId Field in MsgCheckpoint

ID	IF-FINDING-013
Severity	Low
Impact	1 - Low
Exploitability	1 - Low
Type	Implementation
Status	Acknowledged

Involved artifacts

- /x/checkpoint

Description

The `BorChainId` field in the `MsgCheckpoint` message is not validated during message processing. This field is used in both `GetSideSignBytes()` (for signature generation) and `UnpackCheckpointSideSignBytes()` (for decoding), where it is parsed using `strconv.ParseUint`.

In the `GetSideSignBytes()` function, `BorChainId` is parsed using `strconv.ParseUint`(msg.go#L74). When this field contains a malformed or non-numeric string (e.g., `"a-b-c"`), the parsing **fails silently** — returning a default value of `0` without any error. As a result, the system processes the message as if `BorChainId = "0"`, which can cause inconsistent or misleading behavior in downstream components.

Problem Scenarios

The message will still be processed, and an incorrect value may propagate to other components of the system, potentially leading to inconsistent behavior.

Recommendation

Consider whether `BorChainId` needs to be an explicit field in `MsgCheckpoint`, or if it can instead be retrieved from chain parameters (as is done in other components).

If the field is required, proper validation should be added to ensure that it contains a valid and appropriate value, such as a well-formed numeric string representing a supported Bor chain ID.

GetCurrentProposer and GetProposers are Implemented as x/checkpoint Queries

ID	IF-FINDING-014
Severity	Informational
Impact	0 - None
Exploitability	0 - None
Type	Implementation
Status	Acknowledged

Involved artifacts

- x/checkpoint/keeper/grpc_query.go
- checkpoint/query.proto

Description

`GetCurrentProposer` (code ref) and `GetProposers` (code ref) are implemented as `x/checkpoint` module state queries (proto ref), while both `queryServer` implementations of query functions are retrieving the necessary data from the `x/stake` module.

Problem Scenarios

The queries should be implemented as part of the x/stake module query endpoint.

The same query `GetProposers` is implemented on x/stake module as `GetProposersByTimes` (code ref).

Recommendation

Remove `x/checkpoint` queries and add new one `GetCurrentProposer` on `x/stake` module - if this doesn't introduce impact on clients and UI.

Missing Validations for x/checkpoint Query Inputs Before Processing

ID	IF-FINDING-015
Severity	Medium
Impact	1 - Low
Exploitability	3 - High
Type	Implementation
Status	Acknowledged

Involved artifacts

- x/checkpoint/keeper/grpc_query.go

Description

There are two queries that receive input values:

- `GetProposers`, receives `QueryProposerRequest` - `uint64 times`
- `GetNextCheckpoint`, receives `QueryNextCheckpointRequest` - `string bor_chain_id`

It was concluded that validation of these fields are missing as described in the problem scenarios.

Problem Scenarios

`QueryProposerRequest` does not validate the times value:

- if times = 0 - query should not be executed since the empty validator list will be return.
- there is no `math.MaxInt64` validation is not validated as in x/stake implementation (code ref)

```
req.Times >= math.MaxInt64
```

but this is not impacting since the times value is clamped with `validatorSet.Validators` length (code ref):

```
times := int(req.Times)
if times > len(validatorSet.Validators) {
  times = len(validatorSet.Validators)
}
```

These missing validations are not impacting the functionality, but could end the query processing earlier.

`QueryNextCheckpointRequest` contains `bor_chain_id` field of type `string`, that a user could set to an arbitrary value.

It is unclear why is this value sent externally, since it can be retrieved from the `x/chainmanager` chain params (code ref1, ref2).

In the current implementation, the checkpoint returned from the query will contain an externally sent, unvalidated Bor chain ID that could potentially have an invalid value. The impact of this issue depends on how this information is used downstream (code ref):

```
func (q queryServer) GetNextCheckpoint(ctx context.Context, req
↪  *types.QueryNextCheckpointRequest) (*types.QueryNextCheckpointResponse, error) {
  if req == nil {
    return nil, status.Error(codes.InvalidArgument, "empty request")
  }
  ...
  checkpointMsg := types.MsgCheckpoint{
```

```
    Proposer:         proposer.Signer,
    StartBlock:       start,
    EndBlock:         endBlockNumber,
    RootHash:         rootHash,
    AccountRootHash:  accRootHash,
    BorChainId:       req.BorChainId,
  }

  return &types.QueryNextCheckpointResponse{Checkpoint: checkpointMsg}, nil
}
```

Recommendation

Add the input validations described above for both the `GetProposers` and `GetNextCheckpoint` queries.

For the `GetNextCheckpoint` remove `bor_chain_id` as an input parameter and use on-chain value stored in the `x/chainmanager` module parameters.

Improve Error Handling in x/checkpoint Queries

ID	IF-FINDING-016
Severity	Informational
Impact	0 - None
Exploitability	0 - None
Type	Implementation
Status	Acknowledged

Involved artifacts

- x/checkpoint/keeper/grpc_query.go

Description

`GetCheckpointList` query implementation

- Error type `codes.InvalidArgument` for pagination failure is misleading (code ref). This assumes *every* `err` from `query.CollectionPaginate` is due to invalid arguments. Consider propagating the original error or inspecting it for better classification (e.g., internal vs. bad input). `code.Internal` is usually used in the remaining query functions with an additional logging of details.

`GetCheckpointOverview` error handling consistency

- The error handling in `GetCheckpointOverview` function (code ref) returns a status error with `codes.Internal` for all errors, which is fine, but it could be more specific. The improvement would be to create custom error messages to make it clear which specific operation failed, such as:

```
return nil, status.Errorf(codes.Internal, "failed to get validator set: %v", err)
```

Problem Scenarios

Findings listed above could not introduce any issues, they are suggestions for error handling improvements.

Recommendation

As explained in the *Description* section. Review other queries for inconsistencies in error handling.

Inconsistent Checkpoint Handling When Multiple MsgCheckpoint Messages Exist in a Single Block

ID	IF-FINDING-017
Severity	High
Impact	3 - High
Exploitability	2 - Medium
Type	Implementation
Status	Acknowledged

Involved artifacts

- /x/checkpoint/keeper
- /app/abci.go
- /bridge/processor/checkpoint.go

Description

When multiple transactions containing different `MsgCheckpoint` messages (e.g., with differing `EndBlock` or `RootHash`) are included in the same block, only the **first** valid message is processed and stored in the Heimdall state via the `PostHandleMsgCheckpoint` method (side_msg_server.go#L248-L260). All subsequent messages are rejected due to the presence of an already populated `bufferedCheckpoint` (side_msg_server.go#L243).

However, the `ExtendVoteHandler` (abci.go#L223) still iterates over all messages in the block, with the `nonRpVoteExt` value being overwritten each time a new `MsgCheckpoint` is encountered - ultimately retaining only the **last one**. This value is then used to generate checkpoint signatures (abci.go#463–480) and is returned to the bridge processor via query (bridge/processor/checkpoint.go#541–545).

This creates a mismatch between:

- The checkpoint **signed and returned to the bridge**, and
- The checkpoint **stored in Heimdall state and emitted via event**.

This inconsistency could cause issues in downstream components like the bridge, which expects the signed checkpoint to reflect the authoritative state.

Problem Scenarios

- A **malicious proposer** could include multiple distinct but valid `MsgCheckpoint` messages in a single block (e.g., ranges 5-10 and 5-15, each correctly signed and formatted). The system would accept only the first message but sign and expose the last one to the bridge.
- A **legitimate proposer** might inadvertently submit two checkpoints in the same block (e.g., one from the bridge, another from CLI), leading to the same inconsistency.
- External systems monitoring `EventTypeCheckpoint` could act on data that differs from what is actually being bridged to Ethereum.

Recommendation

Implement a safeguard to ensure that only one `MsgCheckpoint` exists per block, and align the signature generation logic (`nonRpVoteExt`) with the checkpoint stored in state.

Missing Checkpoint Length Validations in MsgCheckpoint

ID	IF-FINDING-018
Severity	Low
Impact	1 - Low
Exploitability	2 - Medium
Type	Implementation
Status	Acknowledged

Involved artifacts

- bridge/processor/checkpoint.go
- x/checkpoint/keeper/msg_server.go
- x/checkpoint/keeper/side_msg_server.go

Description

The function `nextExpectedCheckpoint` determines the next probable checkpoint that needs to be sent based on the latest child block from Bor and contract checkpoint state (code ref).

The logic enforces **bounded checkpoint lengths** using two key parameters:

- `AvgCheckpointLength`: the typical length (number of blocks) expected for a checkpoint.
- `MaxCheckpointLength`: the upper limit to prevent overly large checkpoints.

There is a special case processed within the `nextExpectedCheckpoint` function (code ref) when the producers are potentially down and the block production slows down. If not enough blocks are available (`diff < AvgCheckpointLength`), or the calculated range is empty, the system checks how much time has passed since the last checkpoint (code ref):

```
if currentTime - lastCheckpointTime > MaxCheckpointLength * 2
```

This doesn't make previously rejected or invalid checkpoints valid per se- it just allows the system to generate a new checkpoint under a time-based fallback condition when block production is lagging.

Problem Scenarios

A malicious proposer may bypass the automated bridge processing logic by crafting a `MsgCheckpoint` directly or by manipulating the expectation checks enforced by the bridge processor. Therefore, the same set of validation rules that ensure checkpoint correctness must also be implemented in the `MsgCheckpoint` message server and side-tx handlers to guarantee consistency and prevent invalid checkpoints from being accepted.

Although only proposers can produce invalid checkpoints, subsequent honest proposers will resume submitting checkpoints with expected lengths, thereby limiting the impact. However, the risk increases if multiple malicious proposers are consecutively selected, potentially disrupting checkpoint cadence over an extended period.

Recommendation

Introduce the same validation checks from the `nextExpectedCheckpoint` function for `MsgCheckpoint`:

- Message server handler - to prevent invalid messages from being included in proposed blocks, unintentionally
- Side-tx handlers- to prevent maliciously proposed checkpoints with invalid lengths from receiving YES votes during the PreCommit phase.

Insufficient Validation and Authorization in MsgCpAck Handling

ID	IF-FINDING-019
Severity	High
Impact	3 - High
Exploitability	2 - Medium
Type	Implementation
Status	Acknowledged

Involved artifacts

- /x/checkpoint

Description

The `MsgCpAck` message is signed using the `From` field, which is only validated for proper formatting - not for authorization. This means that **any Heimdall v2 user** can submit a valid `MsgCpAck`, regardless of whether they were the original proposer of the checkpoint.

Unlike `MsgCheckpoint`, where local validation enforces the correctness of block ranges and proposer identity, `MsgCpAck` performs minimal local validation. Specifically:

- Only the equality of `start_block` with the buffered checkpoint is checked in `MsgServer` and `PostHandle`.
- The `SideHandle` does validate `start_block`, `end_block`, `Proposer`, and `Number` - but this is done **exclusively by comparing to L1 root chain contract state**, introducing a trust dependency.

If the L1 contract is misconfigured or externally manipulated (including by a malicious proposer), **invalid data in `MsgCpAck` can still be accepted and stored**, such as:

- `end_block == 0`,
- `end_block <= start_block`,
- incorrect proposer identity,
- or a non-sequential checkpoint number.

Problem Scenarios

- **Root chain contract returns invalid checkpoint metadata:**

  Due to a bug or misconfiguration in the root chain contract, it returns an invalid checkpoint with `end_block = 0` or `end_block <= start_block`. Because this data is blindly trusted in `SideHandleMsgCheckpointAck`, the system accepts the message and stores malformed checkpoint state.

- **Checkpoint number is non-sequential, blocks further progress:**

  A malicious user submits a `MsgCpAck` with a `Number` that skips ahead (e.g., the current checkpoint ID is 42, but the message uses 45). The message passes contract-based validation, but causes a mismatch in the internal counter, making it impossible to submit checkpoint 43-44 and **blocking future checkpoint submissions**.

Recommendation

- Introduce stricter validation for `MsgCpAck` at the `ValidateBasic`, message server and post-handler level:
  - Validate that `end_block > start_block` and `end_block != 0`,
  - Optionally require that the signer (`From`) matches the `Proposer` field or enforce authorization logic.
  - Ensure that the `Number` field is sequential and consistent with existing checkpoint IDs to avoid accidental or intentional skipping of checkpoint indices.
- Reconsider the full reliance on L1 contract data for critical checkpoint integrity checks. Add local safeguards to reduce the blast radius of misconfigured or compromised contracts.

Unused and Unvalidated Fields in MsgCpAck (TxHash and LogIndex)

ID	IF-FINDING-020
Severity	Informational
Impact	0 - None
Exploitability	0 - None
Type	Implementation
Status	Acknowledged

Involved artifacts

- /x/checkpoint

Description

The `MsgCpAck` message struct contains two fields, `TxHash` and `LogIndex`, that are not used anywhere in the processing flow. These fields are neither validated nor referenced in any logic within the `x/checkpoint` module.

Problem Scenarios

While these unused fields are currently harmless, keeping them in production messages poses several risks:

- Creates confusion for developers,
- Opens possibilities for misuse in future implementations.

Recommendation

- If `TxHash` and `LogIndex` are not intended to be used, **remove them** from the `MsgCpAck` definition to reduce complexity and minimize attack surface.
- If these fields are planned for future use, implement **explicit validation** for their values.

Potential Execution of MsgCpAck and MsgCpNoAck in the Same Block Height

ID	IF-FINDING-021
Severity	Informational
Impact	0 - None
Exploitability	0 - None
Type	Protocol
Status	Acknowledged

Involved artifacts

-

Description

Currently, it is possible to propose a block containing multiple `x/checkpoint` messages - all variations are possible:

- multiple `MsgCheckpoint` messages (explained in more detail with the issue: *Inconsistent Checkpoint Handling When Multiple MsgCheckpoint Messages Exist in a Single Block*)
- multiple `MsgCpAck`, `MsgCpNoAcks`
- different combinations of all three types of messages.

Also, it is possible that **voted on checkpoint messages from the N-1** block **collide** with the **newly arrived x/checkpoint** messages. This is not posing a problem and is considered a valid flow - since side-tx must be voted on and are executed in the following height, unless a `MsgCpAck` post-tx handler is executed in the same `N` block as the newly arrived `MsgCpNoAck`. This means that, even though the arrived ACK, successfully voted on by the majority of the nodes will be executed in the `PreBlocker` phase of block `N` finalization - the NO-ACK can still be executed, under special conditions explained below.

Problem Scenarios

The combination of executing `MsgCpAck` and `MsgCpNoAck` is possible, in cases of processing a very "old" checkpoint present in the checkpoint buffer:

- `MsgCpAck` was received for a really old checkpoint (older than `n *CheckpointBufferTime`, where `n > 1`) in `N block`
- `MsgCpNoAck` is broadcasted and becomes part of the `N+1 block`, since conditions are fulfilled
- `post-tx` is executed for `MsgCpAck` in `PreBlocker of N+1` -> updates the `latestCheckpoint`, but the timestamp is still old enough
  - new proposer is selected
- It seems possible that during the execution phase for block `N+1` txs - `MsgCpNoAck` could be executed:
  - if timestamp of the latest checkpoint is still old enough, even though updated in `PreBlocker` of `N+1` and
  - sender is still a valid proposer (considering the time elapsed since)
  - proposer will once again be updated, `lastNoAck` will be marked in state (even though the ACK was processed in the same block, and checkpoint buffer flushed,..)

The result is that both post-tx `MsgCpAck` and `MsgCpNoAck` could potentially be **executed at the same block height** - which is unexpected and incorrect from a protocol design point of view - but no other impact or security implications were detected, other then changing proposers with both handler executions.

Recommendation

It is recommended to introduce limitations on the number of `x/checkpoint` messages included in proposed blocks (`PrepareProposalHandler` and `ProcessProposalHandler`). This can be achieved either by filtering out later

transactions containing `x/checkpoint` messages or by rejecting the proposed block entirely.

Message validation can occur both during the message handler execution phase and in the post-handler execution phase. Several specific cases must be handled carefully:

- Valid combinations of `x/checkpoint` messages must be allowed, such as one `MsgCheckpoint` with one or more `MsgCpAck` messages.
- Multiple `MsgCheckpoint` transactions must all be signed by the proposer. In case of multiple valid `MsgCheckpoint txs` - the proposed block could be considered invalid. The valid/non-faulty proposer would eventually be selected and propose the valid block, containing one `MsgCheckpoint` tx.
- Multiple `MsgCpNoAck` messages must also be signed by the proposer. The current logic is already limiting frequency of these `txs` with the `lastNoAck` timestamp mechanism. In case of multiple valid MsgCpNoAck txs - the proposed block could be considered invalid. The valid/non-faulty proposer would eventually be selected and propose the valid block, containing one MsgCpNoAck tx.
- Multiple `MsgCpAck` messages, which do not require proposer signatures, may be submitted by any user; however, post-handler logic ensures only valid checkpoints are persisted.
- To properly handle cases involving a `MsgCpAck` at height N-1 and a `MsgCpNoAck` at height N, additional validation based on the newly introduced `lastAck` timestamp may be necessary.

Any solution must be carefully designed to avoid introducing liveness issues.

Unclear Expectations If AvgCheckpointLength Equals MaxCheckpointLength

ID	IF-FINDING-022
Severity	Low
Impact	1 - Low
Exploitability	1 - Low
Type	Protocol
Status	Acknowledged

Involved artifacts

- x/checkpoint/types/params.go
- bridge/processor/checkpoint.go

Description

It is unclear whether `MaxCheckpointLength` is intended to be equal to `AvgCheckpointLength`. The current validations in the `Validate` function (code ref) allow this edge case.

```
func (p Params) Validate() error {
  if p.MaxCheckpointLength == 0 {
    return fmt.Errorf("max checkpoint length should be non-zero")
  }

  if p.AvgCheckpointLength == 0 {
    return fmt.Errorf("value of avg checkpoint length should be non-zero")
  }

  if p.MaxCheckpointLength < p.AvgCheckpointLength {
    return fmt.Errorf("avg checkpoint length should not be greater than max checkpoint length")
  }

  if p.ChildChainBlockInterval == 0 {
    return fmt.Errorf("child chain block interval should be greater than zero")
  }

  return nil
}
```

Problem Scenarios

The bridge processing logic in `nextExpectedCheckpoint` (code ref) implicitly assumes `AvgCheckpointLength` is **smaller** than `MaxCheckpointLength`. After rounding `diff` down to the nearest multiple of `AvgCheckpointLength`, the logic subtracts 1 more to ensure that: **Final checkpoint size** is *less than* `MaxCheckpointLength` and **never** exactly hits `MaxCheckpointLength`

```
// cap with max checkpoint length
if expectedDiff > checkpointParams.MaxCheckpointLength-1 {
  expectedDiff = checkpointParams.MaxCheckpointLength - 1
}
```

When `AvgCheckpointLength == MaxCheckpointLength`, this assumption breaks.

Recommendation

The expected behavior needs to be explicitly defined:

- Should `AvgCheckpointLength` and `MaxCheckpointLength` ever be allowed to be equal?
- If they are equal, is it acceptable for a checkpoint to reach the full `MaxCheckpointLength`?

Depending on the intended design, the issue can be addressed in two ways:

- **At parameter validation**: enforce that `AvgCheckpointLength < MaxCheckpointLength`, preventing invalid configurations.
- **Or at processing logic**: adjust the checkpoint calculation to correctly handle the case when both parameters are equal, allowing a checkpoint to reach the maximum size if intended.

It is concluded with the Polygon team that `MaxCheckpointLength` and `AvgCheckpointLength` can be equal, so the best course of action would be to change the bridge logic to allow `AvgCheckpointLength <= MaxCheckpointLength`.

Because of this, the property still holds, and the validations are in place.

Checkpoint Module State is not Entirely Exported and Initialized

ID	IF-FINDING-023
Severity	Low
Impact	2 - Medium
Exploitability	1 - Low
Type	Implementation
Status	Acknowledged

Involved artifacts

- x/checkpoint/keeper/genesis.go
- proto/heimdallv2/checkpoint/genesis.proto
- x/checkpoint/types/keys.go

Description

Checkpoint genesis state is defined as (code ref):

```
message GenesisState {
  Params params = 1 [ (gogoproto.nullable) = false, (amino.dont_omitempty) = true ];
  Checkpoint buffered_checkpoint = 2 [ (gogoproto.nullable) = true, (amino.dont_omitempty) =
↪  true ];
  uint64 last_no_ack = 3 [ (amino.dont_omitempty) = true ];
  uint64 ack_count = 4 [ (amino.dont_omitempty) = true ];
  repeated Checkpoint checkpoints = 5 [ (gogoproto.nullable) = false, (amino.dont_omitempty) =
↪  true ];
}
```

While the state is consisted of the following data entries (code ref):

```
var (
  // ParamsPrefixKey represents the prefix for param
  ParamsPrefixKey = collections.NewPrefix([]byte{0x80})

  // CheckpointMapPrefixKey represents the key for each key for the checkpoint map
  CheckpointMapPrefixKey = collections.NewPrefix([]byte{0x81})
  // BufferedCheckpointPrefixKey represents the prefix for buffered checkpoint
  BufferedCheckpointPrefixKey = collections.NewPrefix([]byte{0x82})

  // AckCountPrefixKey represents the prefix for ack count
  AckCountPrefixKey = collections.NewPrefix([]byte{0x83})

  // LastNoAckPrefixKey represents the prefix for last no ack
  LastNoAckPrefixKey = collections.NewPrefix([]byte{0x84})

  // CheckpointSignaturesPrefixKey represents the prefix for checkpoint signatures
  CheckpointSignaturesPrefixKey = collections.NewPrefix([]byte{0x85})

  // CheckpointSignaturesTxHashPrefixKey represents the prefix for checkpoint signatures tx
  ↪  hash
  CheckpointSignaturesTxHashPrefixKey = collections.NewPrefix([]byte{0x86})
)
```

Problem Scenarios

The entire `x/checkpoint` module state should be exported and the implementation must support the full state import, together with the validations. The following state entries are missing from the `ExportGenesis` (code ref) - and `InitGenesis` functions (code ref):

- `CheckpointSignaturesPrefixKey`
- `CheckpointSignaturesTxHashPrefixKey`

The inability to export and initialize the current state of the checkpoint module as necessary poses a significant risk of potential data loss. Moreover, this limitation makes it impossible to create backups of the state, further complicating data management and security measures.

Also the data missing is used when bridging the data between the Heimdall and Ethereum layers. There are also checkpoint queries implemented for retrieving the data that might be potentially lost during the chain migration (a.k.a hard fork).

Recommendation

Implement proper genesis state with possibility of exporting and initializing the entire `x/checkpoint` state.

Incomplete Data Validation in Checkpoint ValidateGenesis Function

ID	IF-FINDING-024
Severity	Low
Impact	2 - Medium
Exploitability	1 - Low
Type	Implementation
Status	Acknowledged

Involved artifacts

- x/checkpoint/types/msg.go
- x/checkpoint/keeper/genesis.go
- x/checkpoint/module.go
- x/checkpoint/types/genesis.go

Description

`InitGenesis` (code ref) can be invoked during chain initialization or during a chain migration (i.e., hard fork) using the genesis file provided by the operator. The resulting state must be validated by calling `ValidateGenesis` (code ref).

If validations are missing or not aligned between genesis validation and `ValidateBasic` functions, then during a chain upgrade or migration, calling `InitGenesis` can allow operator mistakes (or malicious modifications) in the genesis JSON to cause a chain halt or initialize an invalid (corrupted) state in the store.

Problem Scenarios

Currently, the genesis state validation for the `x/checkpoint` module does not cover critical checkpoint state data:

- Checkpoints should be validated prior to adding them to the state - all the validations validating the validity of the checkpoint must be present as in `ValidateBasic` for the `MsgCheckpoint` (code ref).
- Also, the data currently excluded from `ExportGenesis` and `InitGenesis` functions could be validated to confirm correctness of the genesis file:
  - `CheckpointSignature` and
  - `txHash` expected string length and format.

Recommendation

Add all missing validations to the `InitGenesis` flow and ensure they are consistent with the validations performed during transaction processing.

Milestone Module State is not Entirely Exported and Initialized

ID	IF-FINDING-025
Severity	Low
Impact	2 - Medium
Exploitability	1 - Low
Type	Implementation
Status	Acknowledged

Involved artifacts

- proto/heimdallv2/milestone/genesis.proto
- x/milestone/types/keys.go
- x/milestone/keeper/genesis.go

Description

Milestone genesis state is defined as (code ref):

```
message GenesisState {
  Params params = 1
      [ (gogoproto.nullable) = false, (amino.dont_omitempty) = true ];
}
```

While, the state is consisted of the following data entries (code ref):

```
var (
  ParamsPrefixKey       = collections.NewPrefix([]byte{0x80})
  MilestoneMapPrefixKey = collections.NewPrefix([]byte{0x81})
  CountPrefixKey        = collections.NewPrefix([]byte{0x83})
  BlockNumberPrefixKey  = collections.NewPrefix([]byte{0x84})
)
```

Problem Scenarios

The entire `x/milestone` module state should be exported and the implementation must support the full state import, together with the validations. The following state entries are missing from the `ExportGenesis` (code ref) - and `InitGenesis` functions (code ref):

The inability to export and initialize the current state of the milestone module as necessary poses a significant risk of potential data loss. Moreover, this limitation makes it impossible to create backups of the state, further complicating data management and security measures.

Also the data missing is used when optimizing the finalization of the Bor chain block production in Heimdall v2. There are also milestone queries implemented for retrieving the data that might be potentially lost during the chain migration (a.k.a hard fork).

Recommendation

Implement proper genesis state with possibility of exporting and initializing the entire milestone state. The validations present when finalizing the milestone with `ValidateMilestoneProposition` (code ref) in `PreBlocker` should be present in `ValidateGenesis`, as well. Also, validation to confirm that milestone count is equal to the latest finalized milestone as well as valid values for the `BlockNumberPrefixKey` - last block height when the milestone was finalized.

Milestone Proposals Lack Proper Validation of Block Hashes

ID	IF-FINDING-026
Severity	Informational
Impact	0 - None
Exploitability	3 - High
Type	Implementation
Status	Acknowledged

Involved artifacts

- x/milestone/abci/abci.go

Description

It is assumed that more than two-thirds of Bor validators' voting power will not submit invalid block hashes, and that the Bor chain is operational. Currently, Heimdall v2 does not perform any validation to confirm whether the block hashes returned from Bor are correct and correspond to actual existing blocks.

Prior to:

1. validator node including the milestone proposal in the Vote Extension (`ExtendVoteHandler`)
2. finalizing majority milestone (`FinalizeBlock` → `PreBlocker` execution)

the `ValidateMilestoneProposition` validation is performed (code ref):

```
func ValidateMilestoneProposition(ctx sdk.Context, milestoneKeeper *keeper.Keeper,
↪  milestoneProp *types.MilestoneProposition) error {
  if milestoneProp == nil {
    return nil
  }

  params, err := milestoneKeeper.GetParams(ctx)
  if err != nil {
    return err
  }

  if len(milestoneProp.BlockHashes) > int(params.MaxMilestonePropositionLength) {
    return fmt.Errorf("too many blocks in proposition")
  }

  for _, blockHash := range milestoneProp.BlockHashes {
    if len(blockHash) == 0 || len(blockHash) > common.HashLength {
      return fmt.Errorf("invalid block hash length")
    }
  }

  return nil
}
```

The function performs light validations intentionally, to simplify the code implementation:

- milestones must contain at most `MaxMilestonePropositionLength` block hashes
- each block hash must be of **valid block hash exactly 32 bytes length and** it must **come from a keccak256 hash** of properly structured data.

Problem Scenarios

Block hash validation is currently incorrect. Any hash with a length of *32 bytes or fewer* is improperly accepted. Additionally, even when a hash has the correct length, it may correspond to a non-existent block on Bor. However, Heimdall v2 intentionally simplifies validations and relies on the assumptions outlined in the Description.

Recommendation

Strengthen the validation condition.

Replace the current weak check

```
len(blockHash) == 0 || len(blockHash) > common.HashLength
```

with a strict check:

```
len(blockHash) != common.HashLength
```

Consider the potential impact and evaluate introducing stricter block hash validations.

Alternatively, ensure that the Bor chain can reliably handle cases where invalid or non-existent block hashes are present, since milestones are later queried by Bor to determine the last finalized block hash. This code was not a part of the audit, so the impact is unknown.

Malicious Proposer Can Inject Invalid Milestone

ID	IF-FINDING-027
Severity	High
Impact	2 - Medium
Exploitability	3 - High
Type	Implementation
Status	Acknowledged

Involved artifacts

- app/abci.go
- app/vote_ext_utils.go

Description

Milestones proposition, validations and finalization flow is as concluded:

1. `PreCommit` phase, block `N` - milestones proposal: using the `ExtendVoteHandler`.
   - The milestones are injected into the `VoteExtension` structure.
   - `ValidateMilestoneProposition` is called to validate the milestone before including it in the vote extension.
2. `VerifyVoteExtension` phase, block `N`: the milestone inside the vote extension is validated again using `ValidateMilestoneProposition`.
3. Block proposal and validation, block `N+1`:
   - `PrepareProposalHandler`: The proposer prepares the block that includes the canonical vote extension (with the milestone).
   - `ProcessProposalHandler`: Other nodes validate the proposed block and VE.
     - These steps **miss** re-validating the `VoteExtension` milestones during the `ValidateVoteExtensions` (code ref) executed in both handlers (code ref1, ref2).
4. Finalization during `PreBlocker`, block `N+1`: milestones from all votes are processed, that may be invalid or maliciously crafted, because validation at earlier stages was incomplete.

Problem Scenarios

The malicious validator/proposer can add/replace it's own milestone and add a really huge milestone proposition with e.g. a million (or more) of 1-byte `milestoneProp.BlockHashes` during the `PrepareProposalHandler`. It is not possible for a malicious proposer to alter other validators' milestones since the data is signed and the signature is verified (code ref). Since there are no milestone validations present in the `ProcessProposalHandler` (code ref) within the `ValidateVoteExtension` function the altered, but signed milestone injected by the malicious proposer would make it through and reach the `FinalizeBlock` when `PreBlocker` is called.

It seems like it would slow down substantially the `PreBlocker` (code ref) when executing the `GetMajorityMilestoneProposition`:

```
for i, blockHash := range prop.BlockHashes {
  ...
}
```

This malicious validator cannot stop transactions from getting added, because the malicious validator can only do this trick once, when it proposes. If another honest validator proposes a block, and the malicious validator gives a bogus vote extension with a huge number of `milestoneProp.BlockHashes`, then the honest validator would have rejected this vote extension during `VerifyVoteExtensionHandler` that calls `ValidateMilestoneProposition` and hence the honest validator wouldn't have added that malicious vote extension to its block.

Recommendation

Add the missing validations, both in Prepare and Process phases, to confirm that only valid VE milestones can be found in the proposed and finalized block.

Additionally, as a safety measure, milestone validation could be introduced in `GetMajorityMilestoneProposition` when collecting valid voted milestones for processing.

x/milestone Various Minor Code Improvements

ID	IF-FINDING-028
Severity	Informational
Impact	0 - None
Exploitability	0 - None
Type	Implementation
Status	Acknowledged

Involved artifacts

- x/milestone/keeper/grpc_query.go
- x/milestone/abci/abci.go

Description

Here is the list of aesthetic and minor code improvements, refactoring suggestions and issues around logging found during the code inspection of the `x/milestone` Heimdall v2 module. They do not pose a security threat nor do they introduce an issue, but the following suggestions are shared to improve the code readability, keep consistency, optimize, and improve logging.

- Check if `milestone` is `nil` before dereferencing (code ref1, ref2):

  ```
  return &types.QueryMilestoneResponse{Milestone: *milestone}, nil
  ```

  The code *assumes* that `milestone` is non-nil. But what if `q.k.GetMilestoneByNumber()` is potentially changed and can **return `(nil, nil)`** Suggestion: defensively check `milestone != nil`:

  ```
  if milestone == nil {
    return nil, status.Error(codes.NotFound, "milestone not found")
  }
  ```

- Error wrapping in `getBlockHashes` is weak (code ref): It currently returns just `fmt.Errorf("failed to get headers")`, losing the actual reason. Include the original error (`fmt.Errorf("failed to get headers: %w", err)`).
- Context reuse **i**n `GetBorChainBlocksInBatch`, you override `ctx` with a timeout context. It's usually clearer to name the new context differently (`timeoutCtx`), to avoid confusing shadowing.
- Minor optimization In `getBlockHashes` (code ref) Instead of `make([][]byte, 0)`, prefer pre-allocating with an expected capacity: `make([][]byte, 0, len(headers))`. This is a small optimization, but useful if performance matters.
- The `GetMajorityMilestoneProposition` function (code ref) could benefit from refactoring to improve readability and testability. It can be split into smaller, more manageable helper functions, such as: Vote Collection, Vote Aggregation, Majority Filtering, Range Detection, Validator Filtering...

Problem Scenarios

Findings listed above could not introduce any issues, they are suggestions for code improvements.

Recommendation

As explained in the *Description* section.

Proposer Can Submit Milestone Without Block Hashes

ID	IF-FINDING-029
Severity	Informational
Impact	1 - Low
Exploitability	0 - None
Type	Implementation
Status	Acknowledged

Involved artifacts

- x/milestone/abci/abci.go

Description

It is assumed that more than two-thirds of Bor validators' voting power will not submit invalid block hashes, and that the Bor chain is operational. Currently, Heimdall v2 does not perform any validation to confirm whether the block hashes returned from Bor are correct and correspond to actual existing blocks.

However, we noticed one improvement, that could be easily implemented on Heimdall v2.

Problem Scenarios

If the `GetBorChainBlocksInBatch` function fails during execution (code ref), an error is returned, and `getBlockHashes` returns a `nil` milestone (code ref) - which is expected.

However, if the `GetBorChainBlocksInBatch` function executed on Bor does not return any block headers (e.g., if Bor is down or not producing blocks), the result is simply an empty slice (code ref). This empty slice is still used to create a milestone proposition without any blocks (code ref).

Subsequently, the `ValidateMilestoneProposition` function considers this empty milestone as valid (code ref), even though it does not contain any valid block data.

This could potentially occur in situations where Bor is down or not producing blocks, leading to the finalization process accepting an effectively empty milestone.

Recommendation

Do not propose milestones if no block hashes are returned from Bor. Processing of empty milestones is not needed.

Missing Input Data Validations in x/milestone Queries

ID	IF-FINDING-030
Severity	Informational
Impact	0 - None
Exploitability	0 - None
Type	Implementation
Status	Acknowledged

Involved artifacts

- x/milestone/keeper/grpc_query.go

Description

Two potential improvements in logging and validations are noticed:

- `GetLatestMilestone`: Logs error in cases of no milestones existing.
- `GetMilestoneByNumber`: Validation of the requested milestone number is missing.

Problem Scenarios

Minor improvement in `GetLatestMilestone` when it comes to logging is possible. Is it really a *runtime error* that no milestones exist yet? Or is it a normal/expected situation (e.g., fresh chain) (code ref).

`GetMilestoneByNumber` is missing the validation of the requested milestone number.

Recommendation

`GetLatestMilestone` Suggestion: use `Info` or `Warn` log level instead of `Error`:

```
k.Logger(ctx).Info("no milestones found in store yet")
k.Logger(ctx).Warn("no milestones found in store yet")
```

`GetMilestoneByNumber`: Validate the requested number earlier - add an *optional early check* like:

```
count, err := q.k.GetMilestoneCount(ctx)
if err != nil {
  return nil, status.Error(codes.Internal, "failed to get milestone count")
}
if req.Number == 0 || req.Number > count {
  return nil, status.Error(codes.NotFound, "milestone number out of range")
}
```

This avoids asking storage to fetch something we know doesn't exist and provides information for better logging.

Panic Caused by Invalid Validator Public Key in EndBlocker

ID	IF-FINDING-031
Severity	Informational
Impact	3 - High
Exploitability	0 - None
Type	Implementation
Status	Acknowledged

Involved artifacts

- /x/stake/keeper/

Description

In the `x/stake` module's `EndBlocker`, the method `CmtConsPublicKey()` is called for each validator update when preparing Tendermint-compatible validator updates.

If the validator's public key is invalid (e.g., incorrectly formatted, corrupted, or unexpected type), this method will cause a **panic** instead of returning an error (validator.go#L552).

As a result, a single invalid validator (potentially due to a corrupted state or malicious submission from L1) can trigger a chain-wide halt during EndBlocker execution, leading to a DoS attack on Heimdall v2.

Problem Scenarios

A corrupted or incorrectly bridged validator (from L1 contracts) may enter the Heimdall state with a malformed public key, which causes `CmtConsPublicKey()` to panic during EndBlocker processing and halt the entire Heimdall chain.

Recommendation

- Refactor `CmtConsPublicKey()` and `EndBlocker` handling to avoid panic in case of invalid public keys.
- Instead, return an error, log the problem, and skip processing the invalid validator update.
- Optionally, mark invalid validators for forced removal or quarantine to prevent reprocessing in subsequent blocks.

TopUp - Unsafe Queries

ID	IF-FINDING-032
Severity	Informational
Impact	3 - High
Exploitability	0 - None
Type	Implementation
Status	Acknowledged

Involved artifacts

- proto/heimdallv2/topup/query.proto

Description

`IsTopupTxOld` , `GetTopupTxSequence` and `GetAccountProofByAddress`are queries in Heimdall's `topup` module perform calls to a contract to get transaction information from main-net or staking information. Since `module_query_safe` is set to `true`, the query result and gas consumption must be deterministic to allow non-state-breaking calls between modules, keepers or CosmWasm contracts. This cannot be guaranteed due to the external dependency of the call to a contract which can timeout. This can lead to state inconsistencies when other keepers or modules use these queries.

Problem Scenarios

Another module or keeper is using one of the queries to track their state. This could lead to state breakages because of the non-determinism of the contract interaction of the query and result in a chain halt.

Recommendation

Remove the `module_query_safe` option in the protobuf definition for the queries where deterministic behaviour can't be guaranteed.

Missing Validation for Negative FeeToken Amounts During Transfers

ID	IF-FINDING-033
Severity	Informational
Impact	0 - None
Exploitability	0 - None
Type	Implementation
Status	Acknowledged

Involved artifacts

- /x/stake/keeper

Description

In the `PostHandleMsgSignerUpdate` method, the absolute value of the `FeeToken` balance is taken using `.Abs()`(side_msg_server.go#L713), but then the original `coins.Amount` is transferred without an explicit check for whether the initial amount was negative (side_msg_server.go#L717-L718).

According to the project team, the `FeeToken` amount is never intended to be negative at any point in the system. However, this invariant is currently not enforced at the contract or application level, leaving the system vulnerable to potential misuse if a negative balance were ever to arise due to bugs or external manipulation.

Problem Scenarios

If, due to an unexpected bug or system misbehavior, a negative `FeeToken` amount appears, it would be silently converted to a positive value and transferred with a negative value, resulting in incorrect state transitions and potential inconsistencies in the system.

Lack of validation could enable a future attacker (or faulty upstream component) to inject invalid states into the balance handling logic.

Recommendation

Introduce an explicit validation step to ensure that `FeeToken` amounts are non-negative before taking the absolute value and transferring coins.

Topup - Query Input Validation Missing

ID	IF-FINDING-034
Severity	Informational
Impact	0 - None
Exploitability	0 - None
Type	Implementation
Status	Acknowledged

Involved artifacts

- x/topup/keeper/grpc_query.go

Description

`GetTopupTxSequence` and `IsTopupTxOld`do not validate `TxHash`parameter. Accepting any string content and length does not comply to the expectations from underlying decoding function `DecodeString()` which is expecting only hexadecimal characters. Caller `common.FromHex()` does not guarantee this expectation. Errors returned from decoding function are ignored and content is converted to `Hash` and passed to the contract caller.

`VerifyAccountProofByAddress` query does not validate input `Address` nor `Proof`. Underlying `GetAccountProof()` does not do any user address check either before trying to get the merkle path. Note that `GetAccountProof()` is ignoring error returned from `tree.GetMerklePath()`.

`GetAccountProofByAddress` query does not validate input `Address` accepting any string content and passing it to `GetAccountProof()` .

Problem Scenarios

Input data is not validated for the above queries leading to unnecessary consumption of system resources.

Recommendation

Add validation (including length check) for `Address` and `Proof` used as input parameters for above queries.

Validator Object Retains Non-Zero Voting Power After Exit

ID	IF-FINDING-035
Severity	Low
Impact	1 - Low
Exploitability	2 - Medium
Type	Implementation
Status	Acknowledged

Involved artifacts

- /x/stake/keeper
- /x/stake/types/validator_set.go

Description

When processing a validator's exit through the `PostHandleMsgValidatorExit` method, the `VotingPower` field of the validator object is not reset and retains its previous non-zero value.

Although the validator set correctly updates the `VotingPower` to zero during `EndBlocker` execution (validator_set.go#L622), the individual `Validator` object in the store remains inconsistent with this update.

Problem Scenarios

- Queries that fetch `validators` directly may inaccurately reflect that an exited validator still possesses voting power.
- Future modules or system components depending on validator's fields may misinterpret the validator's active status, potentially leading to incorrect behavior.
- Developers maintaining or extending the Heimdall v2 chain could introduce bugs by assuming that a non-zero `VotingPower` indicates an active validator.

Recommendation

Update the validator object's `VotingPower` field to zero within the `PostHandleMsgValidatorExit` method when processing an exit.

Unsafe x/stake IsStakeTxOld Query

ID	IF-FINDING-036
Severity	Informational
Impact	3 - High
Exploitability	0 - None
Type	Implementation
Status	Acknowledged

Involved artifacts

- x/stake/keeper/grpc_query.go
- proto/heimdallv2/stake/query.proto

Description

`IsStakeTxOld`: is x/stake module query performing the call to L1 contract in order to receive the receipt for the tx hash input (code ref):

```
// get main tx receipt
  receipt, err := q.k.contractCaller.GetConfirmedTxReceipt(common.HexToHash(req.TxHash),
↪  chainParams.MainChainTxConfirmations)
  if err != nil {
    return nil, status.Error(codes.Internal, err.Error())
  }
  if receipt == nil {
    return nil, status.Errorf(codes.NotFound, "receipt not found")
  }
```

Since the query contains the annotation `module_query_safe` set to `true` (code ref) it is expected for determinism to be guaranteed.

```
rpc IsStakeTxOld(QueryStakeIsOldTxRequest)
      returns (QueryStakeIsOldTxResponse) {
    option (cosmos.query.v1.module_query_safe) = true;
    option (google.api.http).get = "/stake/is-old-tx";
  }
```

As explained in the the *TopUp: Unsafe Queries* issue: Query result and gas consumption must be deterministic to allow non-state-breaking calls between modules, keepers or CosmWasm contracts. This cannot be guaranteed due to the external dependency of the call to a contract which can timeout. This can lead to state inconsistencies when other keepers or modules use these queries.

Problem Scenarios

Another module or keeper is using one of the queries to track their state. This could lead to state breakages because of the non-determinism of the contract interaction of the query and result in a chain halt.

Recommendation

Remove the `module_query_safe` option in the protobuf definition for the queries where deterministic behaviour can't be guaranteed.

Missing Input Data Validations in x/stake Queries

ID	IF-FINDING-037
Severity	Informational
Impact	0 - None
Exploitability	0 - None
Type	Implementation
Status	Acknowledged

Involved artifacts

- x/stake/keeper/grpc_query.go

Description

Several improvements in logging and validations are noticed for the x/stake queries:

- `GetCurrentValidatorSet` always returns a response, even if there's an error from `GetValidatorSet` (code ref). Early return on error would make the code slightly cleaner:
- Additional validation checks could be introduced for `GetSignerByAddress` to check if `len(req.ValAddress) != 0` (code ref).
- `GetValidatorById` (code ref): could be improved with checking if the req.Id is valid (non 0?) or existing validator Id in the store and with returning the appropriate error in cases when the input ValId is not an existing validator id. Propagate error from `GetValidatorFromValID` (code ref).
- `GetValidatorStatusByAddress`: Validation of the `req.ValAddress != 0` could be introduced. (code ref).
- `GetProposersByTimes`: Input validation for `req.Times`: Correctly checking if `req.Times` exceeds `MaxInt64` (code ref), but we must also ensure that `req.Times` is a valid positive number - if it's `0`, it might not be meaningful for the query.
- `IsStakeTxOld`: `txHash/LogIndex` validation for expected format is also missing. This data is indirectly validated throughout the Heimdall with the ability to receive data from L1. If this is possible, the data is valid.

Problem Scenarios

The lack of proper input validation and error handling in some queries can lead to unnecessary processing and unclear execution errors. Without early returns on invalid input, the queries might perform unnecessary operations, leading to inefficient execution and harder-to-trace issues. Additionally, missing validations can result in misleading error messages or obscure the actual cause of a failure, making debugging more difficult. For example, queries like `GetValidatorById` and `GetValidatorStatusByAddress` may silently process invalid or empty inputs, which can lead to unexpected results.

Recommendation

Introduce input validations early in the query functions to ensure that invalid requests are caught before executing any complex logic.

Additionally, return early on error conditions, such as when the request is invalid, to simplify the flow and avoid unnecessary processing. By implementing these improvements, the code will be cleaner, more efficient, and easier to debug, as errors will be more explicit and encountered earlier in the process.

Improve Error Handling in x/stake Queries

ID	IF-FINDING-038
Severity	Informational
Impact	0 - None
Exploitability	0 - None
Type	Implementation
Status	Acknowledged

Involved artifacts

- x/stake/keeper/grpc_query.go

Description

- `GetSignerByAddress` could be improved for better error propagation: In `GetValidatorInfo`, the store error is wrapped, but then in `GetSignerByAddress`, we lose that detail and just report "error in getting validator" (code ref).
- For the `GetValidatorStatusByAddress`,there is an error suppression inside `IsCurrentValidatorByAddress` (code ref): If `GetAckCount` or `GetValidatorInfo` fails, it just silently returns `false`.

Problem Scenarios

Failing to propagate errors can obscure the root cause of failures. It can be difficult to distinguish between genuine issues (e.g., store corruption, database failure) and expected conditions (e.g., a validator simply not existing). This lack of transparency can lead to unreliable debugging and troubleshooting.

Recommendation

Errors should be propagated upwards and wrapped in upstream functions. This will provide more context about the error and enable better differentiation between different types of failures (e.g., "not found" vs. real system failures).

Implemented Validate() Methods Instead of ValidateBasic() for x/stake Messages

ID	IF-FINDING-039
Severity	Low
Impact	1 - Low
Exploitability	1 - Low
Type	Implementation
Status	Acknowledged

Involved artifacts

- /x/stake/types/msg.go

Description

In the `x/stake` module, messages such as `MsgValidatorJoin` (msg.go#L40-L41), `MsgStakeUpdate`, and others implement a custom `Validate()` method instead of the standard `ValidateBasic()` method.

In Cosmos SDK architecture, `ValidateBasic()` is automatically invoked during transaction processing through the `BaseApp` interface.

However, the custom `Validate()` methods implemented in `x/stake` are not called anywhere in the current Heimdall v2 codebase, meaning important message field validations could be accidentally skipped unless explicitly invoked.

It is worth noting that although `ValidateBasic()` was deprecated in Cosmos SDK v0.50 (link), its behavior is still supported and expected within the standard transaction processing flow. Therefore, a consistent decision should be made across modules - either fully migrate to manual validation or continue relying on `ValidateBasic()` invocation.

Problem Scenarios

Message field validations could be skipped, allowing potentially invalid messages to pass initial checks and be detected and rejected only at later processing stages.

Inconsistent use of validation methods across modules could confuse developers and lead to missed validation in future updates.

Recommendation

Rename and adjust the existing `Validate()` methods to conform to the `ValidateBasic()` method signature and ensure they are automatically invoked by the transaction flow. If choosing to maintain manual validation, make sure `Validate()` is explicitly called within every `MsgServer` handler or side handler, and document this requirement clearly to prevent missed validations.

x/stake Various Minor Code Improvements

ID	IF-FINDING-040
Severity	Informational
Impact	0 - None
Exploitability	0 - None
Type	Implementation
Status	Acknowledged

Involved artifacts

- /x/stake/keeper
- /bridge/processor

Description

Here is the list of aesthetic and minor code improvements and issues around logging found during the code inspection of the `x/stake` Heimdall v2 module. They do not pose a security threat nor do they introduce an issue, but the following suggestions are shared to improve the code readability, keep consistency, optimize, and improve logging.

- In the unstake initialization flow, the logger incorrectly prints `unstakeInit` instead of the appropriate context `joiningValidator` - /bridge/processor/stake.go#L150,
- The `PanicIfSetupIsIncomplete` method is redundantly called multiple times within a short execution path here and here,
- The `err != nil` case is not handled after calling `GetValidatorInfo()` in `MsgValidatorJoin` (msg_server.go#L66),
- The `GetValidatorFromValID()` function is called twice unnecessarily inside `StakeUpdate()`. The first call appears redundant and could be removed (msg_server.go#L112),
- The signer address validation (side_msg_server.go#L147-L155) could be performed earlier, immediately after `eventLogSignerBytes` assignment, improving readability and reducing indentation depth.

Problem Scenarios

Findings listed above could not introduce any issues, they are suggestions for code improvements.

Recommendation

As explained in the *Description* section.

Lack of Authorization Checks for From Field in x/stake Messages

ID	IF-FINDING-041
Severity	Medium
Impact	1 - Low
Exploitability	3 - High
Type	Protocol
Status	Acknowledged

Involved artifacts

- /x/stake/keeper

Description

In the `x/stake` module, the `From` field is used as a sender (and signer) of staking-related messages (e.g., `MsgValidatorJoin`, `MsgStakeUpdate`, etc.). The system currently only validates the format of the `From` address but does not perform any authorization checks to ensure that the sender belongs to an approved group. As a result, any Heimdall v2 user could theoretically submit `x/stake` messages, regardless of their role in the system.

Although the design assumption seems to be that only validators or bridge processors interact with these messages — and although validation against L1 event logs provides a second layer of defense — the current implementation leaves open a window for malicious actors to attempt message injection, spamming, or state manipulation.

Problem Scenarios

Without authorization checks, a malicious user (proposer, validator,...) could submit a crafted staking-related message that passes format validation, leading to unnecessary consumption of processing resources before rejection at later validation stages.

Additionally, if future code changes weaken or bypass event log validation, there is a risk that a malicious message could be incorrectly processed, resulting in x/stake state corruption.

Recommendation

It is recommended to introduce an authorization layer for `x/stake` messages, restricting eligible senders to a predefined set of authorized users, such as active validators or trusted system components like the bridge processor.

Stake Module State is not Entirely Exported and Initialized

ID	IF-FINDING-042
Severity	Low
Impact	2 - Medium
Exploitability	1 - Low
Type	Implementation
Status	Acknowledged

Involved artifacts

- proto/heimdallv2/stake/genesis.proto
- x/stake/keeper/keeper.go
- x/stake/types/keys.go
- x/stake/keeper/genesis.go

Description

`x/stake` genesis state is defined (code ref) to contain validators, current validator set and staking sequences:

```
message GenesisState {
  repeated Validator validators = 1
      [ (gogoproto.nullable) = true, (amino.dont_omitempty) = true ];
  ValidatorSet current_validator_set = 2
      [ (gogoproto.nullable) = false, (amino.dont_omitempty) = true ];
  repeated string staking_sequences = 3 [ (amino.dont_omitempty) = true ];
}
```

while the `x/stake` state contains additional data (code ref)

```
validators:   collections.NewMap(sb, types.ValidatorsKey, "validator", collections.StringKey,
↪  codec.CollValue[types.Validator](cdc)),
validatorSet: collections.NewMap(sb, types.ValidatorSetKey, "validator_set",
↪  collections.BytesKey, codec.CollValue[types.ValidatorSet](cdc)),
sequences:    collections.NewMap(sb, types.StakeSequenceKey, "stake_sequence",
↪  collections.StringKey, collections.BoolValue),
signer:       collections.NewMap(sb, types.SignerKey, "signer", collections.Uint64Key,
↪  collections.StringValue),
lastBlockTxs: collections.NewItem(sb, types.LastBlockTxsKey, "last_block_txs",
↪  codec.CollValue[types.LastBlockTxs](cdc)),
```

stored under appropriate keys (code ref):

```
ValidatorsKey                  = []byte{0x21} // prefix for each key to a validator
ValidatorSetKey                = []byte{0x22} // prefix for each key for validator map
CurrentValidatorSetKey         = []byte{0x23} // key to store current validator set
StakeSequenceKey               = []byte{0x24} // prefix for each key for staking sequence map
SignerKey                      = []byte{0x25} // prefix for signer address for signer map
CurrentMilestoneValidatorSetKey = []byte{0x25} // Key to store current validator set for
↪  milestone
LastBlockTxsKey                = []byte{0x26} // key to store last block's txs
PreviousBlockValidatorSetKey   = []byte{0x27} // key to store the previous block's validator
↪  set
```

Problem Scenarios

The entire `x/stake` module state should be exported and the implementation must support the full state import, together with the validations.

The genesis state as currently defined does not contain entire information about the `x/stake` state. Missing data:

- `signers`
- `lastBlockTxs`
- validator set → `CurrentMilestoneValidatorSetKey`, `PreviousBlockValidatorSetKey`

However, `signer` can be indirectly initialized from the `validators` (exported in Genesis) even though it is not exported, which is done during `InitGenesis` (code ref) with the `AddValidator` function (code ref).

Also, previous block validator set is set to current validator set (code ref) but the current milestone validator set is remaining uninitialized.

The inability to export and initialize the current state of the checkpoint module as necessary poses a significant risk of potential data loss. Moreover, this limitation makes it impossible to create backups of the state, further complicating data management and security measures.

Also the data missing is used during `PreBlocker` execution:

- `lastBlockTxs` are used in order to trigger post-handler executions for the side-txs that are voted on.
- Previous block validator set is retrieved with `GetPreviousBlockValidatorSet` during:
  - `ValidateVoteExtensions`
  - `checkNonRpVoteExtensionsSignatures`
  - `PreBlocker` when propagating the validator set to `getMajorityNonRpVoteExtension`

  Since the previous block validator set is initialized to the current validator set - this means that in this case, the functions listed above will not work with previous block validator set.
- Milestone validator set is not initialized at all. But currently, we did not find it has been read and used - only updated in stored (code ref).

Recommendation

Implement proper genesis state with possibility of exporting and initializing the entire `x/stake` state. Consider if exporting and importing of the correct previous block validator set and milestone validator set is needed.

Heavy Reliance on L1 Event Logs Without Sufficient Local Validation in x/stake Messages

ID	IF-FINDING-043
Severity	Informational
Impact	3 - High
Exploitability	0 - None
Type	Implementation
Status	Acknowledged

Involved artifacts

- /x/stake/keeper

Description

The implementation of the `x/stake` module heavily relies on comparing message data with event logs from the Ethereum L1 chain for validation. While cross-checking with L1 events provides a strong defense, there is a noticeable lack of internal validation for many critical fields at the message server (MsgServer) and side transaction handler levels.

This design means that if the event logs are corrupted, incorrectly emitted, or misinterpreted (due to bugs or malicious interference), the Heimdall v2 chain could process invalid staking-related messages, leading to potential system inconsistencies.

Problem Scenarios

If event logs are corrupted or unavailable, here are a few examples of how they could be exploited:

- In `MsgValidatorExit`, the `DeactivationEpoch` field could be set to `0`, allowing a validator to bypass exit procedures and continue participating in consensus.
- `BlockNumber` and `LogIndex` could be manipulated to create a "sequence" that reorders validator transactions, disrupting the intended lifecycle of validators.
- In `MsgValidatorJoin` and `MsgStakeUpdate`, the `Amount` and `NewAmount` fields could potentially be tampered with, resulting in incorrect voting power assignments for validators.

Without additional internal safeguards, Heimdall becomes highly dependent on the correctness and finality of L1 event logs.

Recommendation

It is recommended to strengthen local validation within the `x/stake` message handlers, ensuring that critical fields are checked ilike DeactivationEpoch, BlockNumber, LogIndex, Amount, and NewAmountndependently wherever feasible, even before cross-referencing L1 event logs.

Incomplete Data Validation in Stake ValidateGenesis Function

ID	IF-FINDING-044
Severity	Low
Impact	2 - Medium
Exploitability	1 - Low
Type	Implementation
Status	Acknowledged

Involved artifacts

- x/stake/types/msg.go
- x/stake/types/validator.go
- x/stake/genesis.go
- x/stake/keeper/genesis.go

Description

`InitGenesis` (code ref) can be invoked during chain initialization or during a chain migration (i.e., hard fork) using the genesis file provided by the operator. The resulting state must be validated by calling `ValidateGenesis` (code ref).

If validations are missing or not aligned between genesis validation and `ValidateBasic` functions, then during a chain upgrade or migration, calling `InitGenesis` can allow operator mistakes (or malicious modifications) in the genesis JSON to cause a chain halt or initialize an invalid (corrupted) state in the store.

Problem Scenarios

Currently, the genesis state validation for the `x/stake` module does not cover critical validators state data:

- stake genesis data triggers validations for the following genesis data:
  - `Validators`: each validator from validators field is validated with the function named `ValidateBasic` (code ref)
  - `StakingSequences` - validated for invalid empty string sequence.
- Validations must be aligned: genesis import and all the existing x/stake message validations e.g. `MsgValidatorJoin` should validate the following data (code ref):

```
func (msg MsgValidatorJoin) Validate(ac address.Codec) error {
  if msg.ValId == uint64(0) {
    return ErrInvalidMsg.Wrapf("invalid validator id %v", msg.ValId)
  }

  addrBytes, err := ac.StringToBytes(msg.From)
  if err != nil {
    return ErrInvalidMsg.Wrapf("invalid proposer %v", msg.From)
  }

  accAddr := sdk.AccAddress(addrBytes)

  if accAddr.Empty() {
    return ErrInvalidMsg.Wrapf("invalid proposer %v", msg.From)
  }

  if msg.SignerPubKey == nil {
```

```
    return ErrInvalidMsg.Wrap("signer public key can't be nil")
  }

  if bytes.Equal(msg.SignerPubKey, EmptyPubKey[:]) {
    return ErrInvalidMsg.Wrap("signer public key can't be of zero bytes")
  }

  return nil
}
```

Recommendation

Add all missing validations to the `InitGenesis` flow and ensure they are aligned with those performed during `x/stake` transaction processing. Currently, validations for `x/stake` messages are entirely absent, as outlined in the issue: *Used Validate() methods instead of ValidateBasic() for x/stake messages.*

Incorrect SignerPubKey Validation Prevents Exited Validators to Rejoin

ID	IF-FINDING-045
Severity	Low
Impact	1 - Low
Exploitability	1 - Low
Type	Implementation
Status	Acknowledged

Involved artifacts

- /x/stake/keeper/msg_server.go

Description

In the current implementation of `MsgValidatorJoin`, a validator attempting to rejoin Heimdall with the same `SignerPubKey` after previously exiting is incorrectly rejected. The system checks whether a validator with the provided `SignerPubKey` exists in the state and immediately blocks the Join if found, regardless of the validator's active or inactive status (msg_server.go#L44-L69). This behavior prevents legitimate rejoining scenarios where a validator exits properly and wishes to reenter the validator set with the same key.

Problem Scenarios

If a validator exits the Heimdall validator set and later attempts to rejoin using the same `SignerPubKey`, the join will be rejected even though the validator is no longer active (i.e., has an `EndEpoch` set). This unnecessarily prevents validators from reusing their previous cryptographic identity after a proper exit and forces them to generate a new keypair for rejoining.

Recommendation

Update the `MsgValidatorJoin` logic to allow rejoining with the same `SignerPubKey` if the validator corresponding to that key is no longer active (i.e., has a non-zero `EndEpoch`). Ensure that validators who are still active are correctly blocked from creating duplicate entries, but allow inactive validators to rejoin without requiring a new key.

Early Validation Missing for Negative Withdrawal Amounts in MsgWithdrawFeeTx

ID	IF-FINDING-046
Severity	Informational
Impact	0 - None
Exploitability	3 - High
Type	Implementation
Status	Acknowledged

Involved artifacts

- x/topup/keeper/msg_server.go
- x/topup/types /tx.pb.go

Description

The implementation of the `WithdrawFeeTx` handler lacks an early explicit validation check for negative withdrawal amounts. Although the transaction will fail during execution when it attempts to use `SendCoins` and the subsequent `subUnlockedCoins` operations, this occurs deeper in the execution flow rather than being validated at the beginning of processing.

Problem Scenarios

Without early validation, the system processes invalid transactions through multiple functions and modules before failing. This consumes unnecessary computational resources.

Recommendation

Add explicit validation for negative amounts at the beginning of the `WithdrawFeeTx` handler.

Appendix: Vulnerability classification

For classifying vulnerabilities identified in the findings of this report, we employ the simplified version of Common Vulnerability Scoring System (CVSS) v3.1, which is an industry standard vulnerability metric. For each identified vulnerability we assess the scores from the *Base Metric Group*, the Impact score, and the Exploitability score. The *Exploitability score* reflects the ease and technical means by which the vulnerability can be exploited. That is, it represents characteristics of the *thing that is vulnerable*, which we refer to formally as the *vulnerable component*. The *Impact score* reflects the direct consequence of a successful exploit, and represents the consequence to the *thing that suffers the impact*, which we refer to formally as the *impacted component*. In order to ease score understanding, we employ CVSS Qualitative Severity Rating Scale, and abstract numerical scores into the textual representation; we construct the final *Severity score* based on the combination of the Impact and Exploitability sub-scores.

As blockchains are a fast evolving field, we evaluate the scores not only for the present state of the system, but also for the state that deems achievable within 1 year of projected system evolution. E.g., if at present the system interacts with 1-2 other blockchains, but plans to expand interaction to 10-20 within the next year, we evaluate the impact, exploitability, and severity scores wrt. the latter state, in order to give the system designers better understanding of the vulnerabilities that need to be addressed in the near future.

Impact Score

The Impact score captures the effects of a successfully exploited vulnerability on the component that suffers the worst outcome that is most directly and predictably associated with the attack.

ImpactScore	Examples
High	Halting of the chain; loss, locking, or unauthorized withdrawal of funds of many users; arbitrary transaction execution; forging of user messages / circumvention of authorization logic
Medium	Temporary denial of service / substantial unexpected delays in processing user requests (e.g. many hours/days); loss, locking, or unauthorized withdrawal of funds of a single user / few users; failures during transaction execution (e.g. out of gas errors); substantial increase in node computational requirements (e.g. 10x)
Low	Transient unexpected delays in processing user requests (e.g. minutes/a few hours); Medium increase in node computational requirements (e.g. 2x); any kind of problem that affects end users, but can be repaired by manual intervention (e.g. a special transaction)
None	Small increase in node computational requirements (e.g. 20%); code inefficiencies; bad code practices; lack/incompleteness of tests; lack/incompleteness of documentation

Exploitability Score

The Exploitability score reflects the ease and technical means by which the vulnerability can be exploited; it represents the characteristics of the vulnerable component. In the below table we list, for each category, examples of actions by actors that are enough to trigger the exploit. In the examples below:

- *Actors* can be any entity that interacts with the system: other blockchains, system users, validators, relayers, but also uncontrollable phenomena (e.g. network delays or partitions).
- *Actions* can be
  - *legitimate*, e.g. submission of a transaction that follows protocol rules by a user; delegation/redelegation/bonding/unbonding; validator downtime; validator voting on a single, but alternative block; delays in relaying certain messages, or speeding up relaying other messages;
  - *illegitimate*, e.g. submission of a specially crafted transaction (not following the protocol, or e.g. with large/incorrect values); voting on two different alternative blocks; alteration of relayed messages.
- We employ also a *qualitative measure* representing the amount of certain class of power (e.g. possessed tokens, validator power, relayed messages): *small* for $< 3\%$; *medium* for 3-10%; *large* for 10-33%, *all* for $>33\%$. We further quantify this qualitative measure as relative to the largest of the system components. (e.g. when two blockchains are interacting, one with a large capitalization, and another with a small capitalization, we employ *small* wrt. the number of tokens held, if it is small wrt. the large blockchain, even if it is large wrt. the small blockchain)

ExploitabilityScore	**Examples**
High	illegitimate actions taken by a small group of actors; possibly coordinated with legitimate actions taken by a medium group of actors
Medium	illegitimate actions taken by a medium group of actors; possibly coordinated with legitimate actions taken by a large group of actors
Low	illegitimate actions taken by a large group of actors; possibly coordinated with legitimate actions taken by all actors
None	illegitimate actions taken in a coordinated fashion by all actors

Severity Score

The severity score combines the above two sub-scores into a single value, and roughly represents the probability of the system suffering a severe impact with time; thus it also represents the measure of the urgency or order in which vulnerabilities need to be addressed. We assess the severity according to the combination scheme represented graphically below.

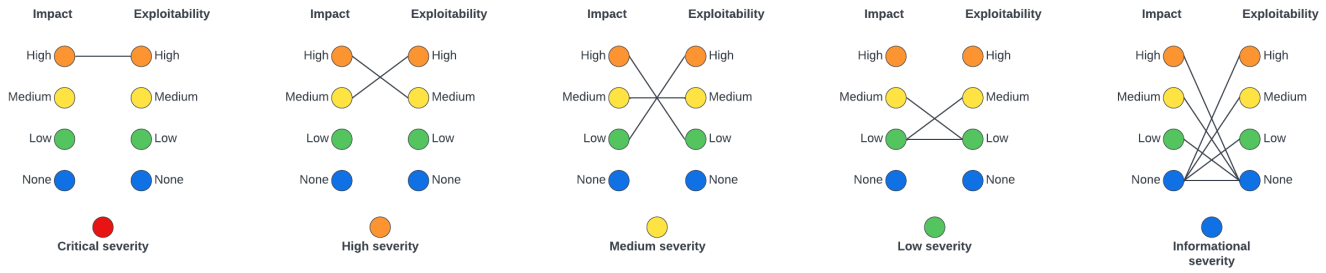

Figure 4: Severity classification

As can be seen from the image above, only a combination of high impact with high exploitability results in a Critical severity score; such vulnerabilities need to be addressed ASAP. Accordingly, High severity score receive vulnerabilities with the combination of high impact and medium exploitability, or medium impact, but high exploitability.

SeverityScore	Examples
Critical	Halting of chain via a submission of a specially crafted transaction
High	Permanent loss of user funds via a combination of submitting a specially crafted transaction with delaying of certain messages by a large portion of relayers
Medium	Substantial unexpected delays in processing user requests via a combination of delaying of certain messages by a large group of relayers with coordinated withdrawal of funds by a large group of users
Low	2x increase in node computational requirements via coordinated withdrawal of all user tokens
Informational	Code inefficiencies; bad code practices; lack/incompleteness of tests; lack/incompleteness of documentation; any exploit for which a coordinated illegitimate action of all actors is necessary

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability, etc.) set forth in the associated Services Agreement. This report provided in connection with the Services set forth in the Services Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement.

This audit report is provided on an "as is" basis, with no guarantee of the completeness, accuracy, timeliness or of the results obtained by use of the information provided. Informal has relied upon information and data provided by the client, and is not responsible for any errors or omissions in such information and data or results obtained from the use of that information or conclusions in this report. Informal makes no warranty of any kind, express or implied, regarding the accuracy, adequacy, validity, reliability, availability or completeness of this report. This report should not be considered or utilized as a complete assessment of the overall utility, security or bugfree status of the code.

This audit report contains confidential information and is only intended for use by the client. Reuse or republication of the audit report other than as authorized by the client is prohibited.

This report is not, nor should it be considered, an "endorsement", "approval" or "disapproval" of any particular project or team. This report is not, nor should it be considered, an indication of the economics or value of any "product" or "asset" created by any team or project that contracts with Informal to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor does it provide any indication of the client's business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should it be leveraged as investment advice of any sort.

Blockchain technology and cryptographic assets in general and by definition present a high level of ongoing risk. Client is responsible for its own due diligence and continuing security in this regard.