Security Audit Report

Polygon Q2 2025:
Bor Updates for Heimdall Decoupling

Last Revised
2025/06/05

Authors:
Aleksandar Stojanovic, Ivan Golubovic, Darko Deuric

Contents

Audit overview

The Project

This audit focuses on two pull requests (#212↗ and #1544↗) aimed at enabling Bor to operate independently when Heimdall is unavailable. The changes also introduce a recovery mechanism through the Backfill Spans strategy, allowing Heimdall to catch up with Bor after a period of downtime.

Scope of this report

The scope covers:

- The decoupling logic allowing Bor to continue block production without Heimdall.
- The recovery process where Heimdall resynchronizes with Bor via span backfilling (`MsgBackfillSpans`).
- The correctness and safety of communication between Bor, Heimdall, and the bridge processor during these processes.

Audit plan

Informal Systems conducted the audit from May 19 to May 28, 2025. The review was carried out by three auditors: Darko Deuric, Ivan Golubovic, and Aleksandar Stojanovic.

Prior to diving into manual code review, the team defined key system properties. The audit then evaluated whether these properties held throughout the implementation. While most properties were preserved, several notable issues were identified.

Conclusions

- Communication between Bor & Heimdall and between the Bridge Processor & Heimdall is not consistently wrapped with timeout contexts. This can lead to indefinite stalls—for example, Bor halting block production (finding "Missing timeout in gRPC span fetch can halt Bor"), or the span processor hanging (finding "Missing timeout on Heimdall span fetch can stall processor") during span proposal or backfill procedures.
- There's a risk of non-deterministic behavior in determining the span proposer (finding "Non-deterministic and context agnostic proposer check"), especially during span or checkpoint proposals. This may lead to inconsistent behavior across nodes. (Note: Checkpoint proposal logic was outside the audit scope.)
- We recommend using `latestMilestoneEndBlock` when performing backfills (finding "Use of non-finalized block number in span backfill"). This ensures that only finalized spans (based on milestones) are subject to backfill, avoiding the risk of overriding unfinalized or still-progressing spans.
- A high severity issue (finding "Weak validation in `MsgBackfillSpans` and `SideHandleMsgBackfillSpans` allows forged or inconsistent spans") was identified in the validation logic for `MsgBackfillSpans`—both in the standard and side message servers. Currently, the message can be crafted and submitted by any actor, not just the span processor. The minimal field validation and lack of access control make it possible for a malicious actor to overwrite valid spans (including validator and producer sets) with forged data, up to the latest milestone span.

Audit Dashboard

Target Summary

- **Type: Protocol & Implementation**
- **Platform: Go**
- **Artifacts:**
  - Heimdall-v2, Msg to backfill spans #212↗, commit hash↗
  - Bor, Bor without heimdall #1544↗, commit hash↗

Engagement Summary

- **Dates**: May 19, 2025 - May 28, 2025
- **Method**: Manual code review

Severity Summary

Finding Severity	Number
Critical	1
High	2
Medium	3
Low	2
Informational	5
Total	**13**

System Overview

The Heimdall–Bor architecture is designed to ensure liveness and seamless synchronization, even when Heimdall is temporarily offline.

Decoupling Behavior

When Heimdall is unavailable, **Bor continues to produce blocks and commit spans autonomously**, using the latest validator and producer set it received from Heimdall prior to the outage. During this fallback mode, Bor maintains span commitments locally without relying on Heimdall. While in fallback, Bor does not modify validator or producer sets, preserving the last known state.

Resync on Heimdall Recovery

Once Heimdall comes back online, it initiates a graceful resynchronization using the Backfill Spans mechanism. The process involves:

1. Bridge Processor on Heimdall fetches the latest span info from Bor.
2. It generates a `MsgBackfillSpans` containing the missing span range.
3. The proposer broadcasts this message to Heimdall.
4. Heimdall's `MsgServer` validates the message against expected chain state.
5. The `SideMsgServer` performs side-tx validation, fetching Bor headers, checking milestone consistency and voting.
6. If 2/3 of validators vote YES, new spans committed by Bor are added to Heimdall's state via `PostHandleMsg-BackfillSpans`.

Normal Mode Resumption

After catching up, Heimdall resumes normal span coordination, and Bor seamlessly returns to synced operation, now relying on Heimdall once again for span validation and updates.

Threat Model

Property-01: Bor isn't stuck if Heimdall doesn't respond

Findings: Missing timeout in gRPC span fetch can halt Bor

Violation consequences

There was an initial concern that Bor could get stuck in a retry loop or hang entirely when trying to fetch span data from Heimdall using methods like `GetSpanV1/V2` or `GetLatestSpanV1/V2`. However, after reviewing the code, it turns out that **in most cases, Bor is protected from hanging**, especially when using the REST-based client or when calling `updateLatestHeimdallSpanV1/V2`.

Conclusion

Property holds.

The REST version of `GetSpanV1/V2` wraps the request in a timeout (`internalFetchWithTimeout`), even if the incoming context has no deadline. This ensures that Heimdall being unresponsive won't block the call forever.

If the REST request fails, Bor falls back to the latest locally stored span (`getLatestHeimdallSpanV1/V2`), allowing block production to continue.

`updateLatestHeimdallSpanV1/V2` explicitly creates a timeout context when fetching the latest span:

```
ctxWithTimeout, cancel := context.WithTimeout(context.Background(),
↪  getSpanTimeout)
defer cancel()
respSpan, err := c.HeimdallClient.GetLatestSpanV2(ctxWithTimeout)
```

Since the context includes a timeout, these calls cannot hang indefinitely, even if Heimdall is unresponsive.

The only remaining risk comes from the gRPC client path, where the context is not wrapped with a timeout. This is detailed in finding "Missing timeout in gRPC span fetch can halt Bor".

Property-02: Fallback mode must be entered only after confirmed unavailability

Violation consequences

- If fallback mode is triggered even when Heimdall is still available, Bor might use outdated span data from its local database instead of the correct, fresh data from Heimdall.
- This could cause Bor to work with wrong or stale span info, potentially leading to incorrect block validation or issues with syncing the chain.
- In the worst case, it may allow the chain to diverge or operate on assumptions that are no longer valid, risking consensus problems.

Conclusion

Property holds.

The intention is to use fallback mode only when Heimdall is unavailable. There are two possible approaches here: one is to retry Heimdall a few times before falling back, and the other — which is currently implemented — is to fall back to the local DB immediately after a single failed attempt to fetch a span from Heimdall. This shows that availability was prioritized over strict retry behavior — which makes sense in this context, as it ensures Bor can continue operating smoothly even if Heimdall has a brief outage.

Property-03: Correct span is derived if Heimdall is not available

Violation consequences

- Span Id or block range can be wrong.
- If Bor nodes construct fallback spans differently (e.g., different start block, span length, validator set), they'll fork.

Conclusion

Property holds.

Even if Heimdall is temporarily unavailable, Bor is able to construct the correct new span using fallback logic. The span that Bor last used is always tracked via the smart contract (`GetCurrentSpan`), and this value is used as the base when committing a new span (`span.Id + 1`).

If Heimdall is unreachable, Bor falls back to the latest span stored locally in the DB (updated earlier via `updateLatestHeimdallSpanV1/V2` when Heimdall was available). The fallback span is then patched to create a new one:

- `Id` is set to `currentSpan.Id + 1`,
- `StartBlock` is calculated as `header.Number + sprintLength`,
- `EndBlock` is derived as `StartBlock + spanLength`, where `spanLength` is based on the previous span.

These calculations are deterministic and follow the expected span structure, ensuring correctness even during fallback. Since the source span for both DB and smart contract is expected to be in sync (from previous updates), this process ensures Bor can continue producing valid spans without Heimdall — maintaining both **availability and correctness** in the absence of a live span response.

Property-04: Heimdall v1 and v2 APIs must be handled separately and safely selected

Violation consequences

- If the wrong versioned API (V1 or V2) is called due to incorrect flag handling, Bor could try to decode an incompatible response format or operate on incorrect data.
- This may lead to span misconfigurations, invalid block production, or a crash if unmarshalling fails.
- Mistakes in version selection could also silently break span consistency without obvious failure.

Conclusion

Property holds.

The code clearly separates Heimdall V1 and V2 logic and uses the `IsHeimdallV2` flag to choose the correct API paths (`GetSpanV1` vs `GetSpanV2`, `updateLatestHeimdallSpanV1` vs `updateLatestHeimdallSpanV2`, etc). This flag is determined based on chain version (e.g., `minor >= 38`) and stored persistently via `storeIsHeimdallV2Flag`. From

the audit scope, it appears that each code path consistently respects the flag and no cross-version calls are made. This ensures that the correct API is used in all cases, avoiding version mismatches and maintaining correctness through the transition.

Property-05: Bor continues producing blocks using the last known validator and producer set when Heimdall is off

Conclusion

Property holds.

When Heimdall is temporarily unavailable, Bor doesn't try to change or update the validator and producer set—it simply reuses the most recent set it already has. This is intentional and the correct behavior to keep the chain running smoothly. Since Heimdall is the source of truth for validator updates, and those updates can't be fetched during downtime, Bor using the last known set is the safest way to continue operating without introducing inconsistencies or making unauthorized changes.

Property-06: Bor smoothly re-syncs with Heimdall once it comes back

Threats

Bor reuses an old span from the DB even after Heimdall comes back, leading to inconsistent block production.

Conclusion

Property holds.

Bor will not prematurely accept stale spans from Heimdall, because it always asks for the next expected span (`GetSpanV2(ctx, newSpanID)`).

If Heimdall hasn't reached that span yet, Bor remains in fallback mode and continues committing spans using its internal logic.

Once Heimdall recovers and reaches the expected span ID, Bor will automatically transition back to trusting Heimdall.

Property-07: No gaps in span IDs during normal and backoff mode

Findings: Incorrect Span ID Calculation Due to Exclusive Range and Offset Logic

Conclusion

Property holds.

Bor ensures there are no gaps in span IDs—regardless of whether Heimdall is available (normal mode) or unavailable (backoff mode):

- In backoff mode, Bor uses `FetchAndCommitSpan` with `span.Id + 1`, ensuring a strictly increasing span ID sequence and continuous span window.

- In normal mode, Bor trusts Heimdall's response for the next span. However, Heimdall itself enforces strict validation on the proposed span through the `ProposeSpan` handler.

The `ProposeSpan` function includes some of the critical checks like:

- Span/block continuity: `msg.SpanId == lastSpan.Id + 1`, and `msg.StartBlock == lastSpan.EndBlock + 1`
- Start/end block validity: `msg.EndBlock > msg.StartBlock`

These validations ensure that Heimdall only accepts properly formed and sequential spans. Thus, even though Bor doesn't verify span continuity when Heimdall is available, it remains safe to do so because Heimdall enforces these checks at the source.

While there is an issue with `CalcCurrentBorSpanId` function's span ID calculation logic, fortunately this function is only used in validation paths (ref1↗, ref2↗) and not in critical span generation or commitment flows, limiting its impact on the system's span continuity guarantees.

Overall, the combined logic in both Bor and Heimdall guarantees gap-free span ID progression.

Property-08: Span immutability in normal operation

Threats

If Heimdall returns the same span ID but with different validator/producers, the code silently skips the update.

Conclusion

Property holds.

Heimdall provides a strong guarantee that once a span is proposed, it will not change in any way — including its validator set or selected producers. Therefore, when Bor repeatedly calls `c.HeimdallClient.GetLatestSpanV2(ctxWithTimeout)` during the execution of a given span, it is safe and expected that Heimdall will always return the same span data. Based on this immutability guarantee, Bor's logic to reject updates for spans with the same ID is correct and intentional. The system relies on this invariant, and no action is needed if the span ID has not changed.

The only exception to this rule is through the explicitly designed Backfill message, which allows span modification only under controlled backfilling scenarios. Outside of this mechanism, span data remain fixed.

Property-09: Heimdal maintains 2/3+ validator voting for backfill spans

Violation consequences

- Potential acceptance of invalid or malicious span backfills.
- Inconsistent validator set state between Heimdall and Bor.
- Risk of network fork if spans are incorrectly backfilled.

Threats

- Malicious validators could attempt to backfill spans without proper consensus.
- Network partitions could lead to conflicting span backfills.

Conclusion

Property holds.

Using side transactions ensures 2/3+ validator agreement through the standard consensus mechanism, effectively preventing malicious span backfills.

Property-10: Proposer determination during span/backfill proposal must be deterministic and context-aware

Findings: Non-deterministic and context agnostic proposer check

Violation consequences

- Inconsistent proposer selection across nodes.
- Potential duplicate span proposals.

Threats

- Network latency causing different nodes to see different "current" proposers.
- Heimdall API responses varying based on node sync state.
- Lack of block/span context leading to ambiguous proposer selection.

Conclusion

Property does not hold.

The current proposer determination mechanism lacks proper context anchoring. The `IsProposer` check relies on querying the "current" proposer through `GetProposersByTimes` without tying it to a specific block height, span, or sprint context. This creates a non-deterministic environment where different nodes might disagree on who the valid proposer is at any given moment. While this works in practice most of the time, it violates the principle that consensus-critical operations must be fully deterministic.

Property-11: No override of existing valid spans

Findings: Weak validation in MsgBackfillSpans and SideHandleMsgBackfillSpans allows forged or inconsistent spans

Violation consequences

- Historical validator sets could be maliciously altered.
- Chain security compromised by rewriting validator history.
- Potential for validator set manipulation extending into future spans.

Threats

- Attacker can specify an older span as latest span in `MsgBackfillSpans` and by doing that spans between specified "latest" and actual latest could be overwritten.

Conclusion

Property does not hold.

It is possible to overwrite spans from the past by manipulating latest span id when sending `MsgBackfillSpans` .

Property-12: All input parameters must be validated

Findings: Weak validation in MsgBackfillSpans and SideHandleMsgBackfillSpans allows forged or inconsistent spans

Conclusion

Property does not hold.

The codebase implements comprehensive parameter validation across all span-related messages. All critical parameters undergo multiple validation layers through message handlers and side-tx processing.

The only identified weakness is in the validation of the last span ID parameter, which could potentially allow overwriting of past spans due to relying on the message parameter instead of deriving it from chain state.

While the validation system is functional, it could be optimized by reducing redundant checks across handlers and minimizing message fields to only those that cannot be derived from chain state.

Property-13: Bor and Heimdall must agree on validator/proposer set for a span

Violation consequences

- Consensus failures.
- Block production issues.

Threats

- Malicious actors could try to manipulate validator sets during backfill process.
- Incorrect span synchronization during backfill could propagate wrong validator sets.

Conclusion

Property holds.

The backfill mechanism ensures validator set agreement through:

- Using the last span as a template for generating backfill spans
- Preserving the original validator set and selected producers when creating synthetic spans.
- Side-tx validation requiring 2/3+ validator agreement for span proposal.

This approach is safe as long as the last valid span's validator set is correct, since backfilled spans inherit their validator sets from it. However, this also means that any error in the source span's validator set would propagate through all backfilled spans.

Property-14: Heimdall must recover gracefully from Bor errors

Heimdall should tolerate a Bor node generating bad blocks while offline.

Violation consequences

- Network partition if Heimdall nodes accept conflicting span information.
- Potential chain halt if consensus cannot be reached on the valid proposer set.

Threats

- A Bor node with a failed Heimdall instance could generate blocks with backfilled spans that don't match the network consensus.
- Malicious node could intentionally go offline and attempt to propose blocks with manipulated proposer sets.
- Network split if some nodes accept backfilled blocks while others reject them.
- If the failed node was both Heimdall proposer and Bor block producer, it could try to enforce its local (incorrect) view of the spans.

Conclusion

Property holds.

- Other Heimdall nodes will reject backfilled spans from the failed node since it will not be able to reach consensus for them.
- Bor nodes validate blocks against proposer sets from their synced Heimdall nodes.
- Blocks with incorrect proposer sets (from backfilled spans) are automatically rejected by the network majority.
- The system maintains consistency by requiring consensus on span information across the network.
- Heimdall has a mechanism that automatically select new proposers if the current one fails.

Property-15: External API calls must be time-bounded to ensure processor liveness

Findings: Missing timeout in gRPC span fetch can halt Bor, Missing Timeout on Heimdall Span Fetch Can Stall Processor

Violation consequences

- Bridge processor stalling due to indefinite blocking on API calls.
- System-wide liveness degradation.
- Block production delays or complete halts.

Conclusion

Property does not hold.

The codebase currently lacks consistent timeout enforcement in several critical API calls:

- Bor's gRPC span fetching operations lack timeouts.
- Heimdall's REST API calls in span processor don't enforce timeouts.
- Multiple span-related operations lack timeout bounds.

This creates a significant liveness risk as any network disruption or service unresponsiveness could cause indefinite blocking. While some parts of the system properly implement timeouts using `context.WithTimeout`, this practice

needs to be consistently applied across all external API calls to ensure system resilience.

Property-16: Span operations must only reference finalized blocks

Findings: Use of Non-Finalized Block Number in Span Backfill

Violation consequences

- Spans could be built on reorged/orphaned blocks.

Threats

- Malicious actors attempting to exploit non-finalized block references.

Conclusion

Property does not hold.

The codebase currently has mixed handling of block finality in span operations. While it correctly uses milestone end blocks for validation boundaries, it inconsistently uses non-finalized block numbers (`latestBorBlockNumber`) in backfill operations. This violates the principle that span operations should only reference blocks that have achieved milestone finality. The violation is particularly concerning in backfill scenarios where spans could be created using blocks that may later be reorged, potentially compromising the system's finality guarantees.

Property-17: Prevent spam of backfill requests

Findings: Weak validation in MsgBackfillSpans and SideHandleMsgBackfillSpans allows forged or inconsistent spans

Conclusion

Property does not hold.

The current implementation has insufficient spam protection for `MsgBackfillSpans` messages:

- Check if current process is a proposer (ref ↗) is only done in a bridge processor and (as Polygon team clarified) is used only to prevent spam coming from the other bridge processors that are not current proposer. There isn't implemented any prevention of third party user sending `MsgBackfillSpans` messages to the message server.
- While `msg.Proposer` is validated to be a valid address format inside message server (ref ↗), there's no check that the sender is actually an authorized proposer.
- The side-tx validation provides some protection through 2/3+ validator agreement, but doesn't prevent the initial spam of requests.

Findings

Finding	Type	Severity	Status
Weak validation in MsgBackfillSpans and SideHandleMsgBackfillSpans allows forged or inconsistent spans	Implementation	Critical	Acknowledged
Non-deterministic and context agnostic proposer check	Implementation	High	Acknowledged
Missing timeout in gRPC span fetch can halt Bor	Implementation	High	Acknowledged
Incorrect span ID calculation due to exclusive range and offset logic	Implementation	Medium	Acknowledged
Missing timeout on Heimdall span fetch can stall processor	Implementation	Medium	Acknowledged
Use of non-finalized block number in span backfill	Implementation	Medium	Acknowledged
Unnecessary RPC calls in non-proposer nodes during span proposal check	Implementation	Low	Acknowledged
Unnecessary Heimdall span fetch calls on every sprint cause redundant load	Implementation	Low	Acknowledged
Overuse of maxBlockNumber variable reduces clarity	Implementation	Informational	Acknowledged
Unhandled errors and panics in goroutines	Implementation	Informational	Acknowledged
Lack of documentation for span backfill finalization condition	Documentation	Informational	Acknowledged
Redundant call to CalculateSprint in needToCommitSpan	Implementation	Informational	Acknowledged

Finding	Type	Severity	Status
Redundant sprint start check in checkAndCommitSpan	Implementation	Informational	Acknowledged

Weak validation in MsgBackfillSpans and SideHandleMsgBackfillSpans allows forged or inconsistent spans

Severity Critical **Impact** 3 - High **Exploitability** 3 - High

Type Implementation **Status** Acknowledged

Involved artifacts

- heimdall-v2/x/bor/keeper/msg_server.go↗
- heimdall-v2/x/bor/keeper/side_msg_server.go↗

Description

The `MsgBackfillSpans` flow, used to synchronize Heimdall with Bor spans after downtime, relies heavily on input values forwarded by the message sender, without verifying their authenticity or origin. Both `msgServer.BackfillSpans` and `sideMsgServer.SideHandleMsgBackfillSpans` contain minimal validation logic, making it possible for a specially crafted message to pass through and cause Heimdall to commit frged or inconsistent spans, even if Bor never used them.

Problem Scenarios

On `msgServer.BackfillSpans`:

- Fields like `LatestSpanId`, `LatestBorSpanId`, and `LatestBorBlock` are accepted as-is from the message sender.
- Heimdall performs only basic consistency checks (e.g., span ordering, Bor block bounds, span ID math via `CalcCurrentBorSpanId`).
- There is no strict enforcement that the message extends the latest span currently stored in Heimdall state.
- Any actor (not just the designated proposer) can call this method directly.
- There is a mix of useful and less effective validations (example↗) across both message handlers, relying on message parameters for validation while they are accessible on chain.

On `sideMsgServer.SideHandleMsgBackfillSpans`:

- The side-tx handler performs a few additional runtime checks:

```
header, err := s.k.contractCaller.GetBorChainBlock(...)
if header == nil || err != nil { ... }

if _, err := s.k.GetSpan(...); err != nil { ... }
```

These checks only ensure that the block exists in Bor RPC and that the referenced span exists in Heimdall, both of which are trivially satisfiable with crafted inputs.

- The strongest protection comes from this check:

```
latestMilestone := s.k.mk.GetLastMilestone(...)
if msg.LatestBorBlock > latestMilestone.EndBlock {
  return VOTE_NO
}
```

This prevents proposers from forging spans into the future by capping accepted Bor blocks at the milestone end height, which is retrieved from a trusted on-chain source (`x/milestone`).

However, the system remains vulnerable to potential past span forgery. Heimdall may accept and commit spans

that diverge from Bor's real span usage, leading to long-term consensus inconsistencies. Since any actor can call `MsgBackfillSpans`, an attacker could flood Heimdall with misleading or confusing backfill attempts, since most of the validation checks rely on message parameters that could be manipulated by the sender.

For example, when a `MsgBackfillSpans` is sent, Heimdall fails to verify if that span id is actually the latest one in Heimdall. This oversight allows an attacker to specify an older span as the latest span, effectively overwriting all spans from that point up to the latest milestone.

The impact is magnified if Heimdall fails again afterwards, as Bor will use the most recent span (which now contains validators from the older span) as a template for generating synthetic spans. This creates a cascading effect where validators from a historical span can extend their control far into the future.

Recommendation

- Instead of fetching a span using an arbitrary span ID from the message, use Heimdall's latest span ID as source of truth.
- Minimize the `MsgBackfillSpans` structure to only essential fields.
- Remove redundant checks that don't enhance security.

Non-deterministic and context agnostic proposer check

Severity High **Impact** 3 - High **Exploitability** 2 - Medium

Type Implementation **Status** Acknowledged

Involved artifacts

- `heimdall-v2/bridge/util/common.go`↗

Description

The bridge processor currently determines proposer eligibility (`IsProposer`↗) by querying the Heimdall API for the "current" proposer, using `GetProposersByTimes` with a simple count parameter. This approach is non-deterministic, as API responses can vary depending on node sync and timing, and is not tied to any specific block height, span, or sprint.

Problem scenarios

- Possible proposer disagreement or race conditions, especially if bridge nodes or Heimdall instances are not perfectly in sync.
- Risk of duplicate or missed / rejected span proposals.

Recommendation

Proposer check should be anchored to a specific block height, span ID, or sprint. The API and bridge logic must determine proposership for the exact consensus context, ensuring all nodes compute the same proposer for a given block or span and eliminating race conditions.

Missing timeout in gRPC span fetch can halt Bor

Severity High **Impact** 3 - High **Exploitability** 2 - Medium

Type Implementation **Status** Acknowledged

Involved artifacts

- consensus/bor/bor.go↗
- consensus/bor/heimdallgrpc/span_v2.go↗

Description

When Bor uses the gRPC version of `GetSpanV1/V2`↗, it passes a context to the call without enforcing a timeout. The relevant code looks like this:

```
retryWithBackoff(func() bool {
    var err error
    response, err = c.HeimdallClient.GetSpanV2(ctx, newSpanID)
    if err == nil {
        return true
    }
  ...
    res, err := h.borQueryClient.GetSpanById(ctx, req) // inside func (h
↪   *HeimdallGRPCClient) GetSpanV2
    if err != nil {
        return nil, err
    }
```

Unlike the REST client, the gRPC client does not override or wrap the context with a timeout. If the passed-in context is `context.Background()` (as it is in `checkAndCommitSpan`), then the call will **block indefinitely** if Heimdall is unresponsive.

Because this call blocks, Bor never reaches the fallback logic (`getLatestHeimdallSpanV2()`), and block production can stall.

Problem Scenarios

- Bor is running with the gRPC-based Heimdall client.
- Heimdall becomes unreachable or stops responding to gRPC calls.
- Bor calls `GetSpanV1/V2()` using a context without a timeout.
- The call hangs indefinitely and never returns.
- Fallback to cached span is never executed.
- Bor gets stuck waiting and can't continue block production.

Recommendation

Make sure to wrap the context passed to `GetSpanV2()` with a timeout, like so:

```
ctxWithTimeout, cancel := context.WithTimeout(ctx, **X***time.Second)
defer cancel()

response, err := c.HeimdallClient.GetSpanV2(ctxWithTimeout, newSpanID)
```

The same for `GetSpanV1()` .

Incorrect span ID calculation due to exclusive range and offset logic

Severity Medium **Impact** 1 - Low **Exploitability** 3 - High

Type Implementation **Status** Acknowledged

Involved artifacts

- `heimdall-v2/x/bor/types/util.go`↗

Description

The current span ID calculation↗ computes `spanLength` as `EndBlock - StartBlock`, which treats the span as an exclusive range. In practice, block spans are inclusive of both the start and end block, so the correct calculation should be `spanLength = EndBlock - StartBlock + 1`. Additionally, the offset for determining span progression should be based on `EndBlock`, not `StartBlock`, to avoid incrementing the span ID prematurely.

Problem scenarios

Using an exclusive range and offset from `StartBlock` causes the span ID to increment too early at boundaries, leading to off-by-one errors. This can result in Bor blocks being misattributed to the wrong span, introducing inconsistencies in span assignment, block validation, and potentially triggering protocol-level discrepancies between nodes.

Recommendation

Update the span calculation to use an inclusive block range (`spanLength = EndBlock - StartBlock + 1`) and compute the offset from `EndBlock` rather than `StartBlock`. This will ensure correct span assignment for all blocks, especially at boundaries, and maintain consistency with protocol expectations.

Missing timeout on Heimdall span fetch can stall processor

Severity Medium **Impact** 3 - High **Exploitability** 1 - Low

Type Implementation **Status** Acknowledged

Involved artifacts

- `heimdall-v2/bridge/processor/span.go`↗

Description

The `getSpanById` function↗ fetches span data from Heimdall via a REST API call without enforcing a timeout. The same is true for the `GetChainmanagerParams` call↗ in `backfillSpans` , as well as `getLastSpan` and `GetLatestMilestone` in `checkAndPropose` .

Problem scenarios

Without a timeout, these functions can block indefinitely if the remote Heimdall API hangs or is slow to respond. This can degrade liveness, stall the span processor, and lead to resource exhaustion or deadlocks, particularly under network failure or denial-of-service conditions. Lack of timeout handling reduces system resilience and increases operational risk.

Recommendation

Add a context or hard timeout to all remote API calls in `getSpanById` to ensure that network issues cannot stall span processing logic, similarly to what has been done in `GetStartBlockHeimdallSpanID` (using `context.WithTimeout`).

Use of non-finalized block number in span backfill

Severity Medium **Impact** 2 - Medium **Exploitability** 2 - Medium

Type Implementation **Status** Acknowledged

Involved artifacts

- `heimdall-v2/bridge/processor/span.go`↗

Description

The current implementation is correctly checking if the `latestMilestoneEndBlock` is past the last span↗ from Heimdall, but then invokes↗ `backfillSpans` with `latestBorBlockNumber`, which may reference blocks that have not yet been finalized.

Problem scenarios

If Heimdall proposes or backfills spans based on non-finalized Bor blocks, there is a risk that these blocks could be re-organized or orphaned before milestone finality is reached. This can result in Heimdall building spans over invalid or abandoned blocks, undermining the finality guarantees of the system and potentially leading to inconsistent state or replay vulnerabilities.

Recommendation

Invoke `backfillSpans` with `latestMilestoneEndBlock` to ensure only milestone-finalized blocks are included in backfill operations.

Unnecessary RPC calls in non-proposer nodes during span proposal check

Severity Low **Impact** 1 - Low **Exploitability** 1 - Low

Type Implementation **Status** Acknowledged

Involved artifacts

- `bridge/processor/span.go`↗

Description

In the `checkAndPropose` function↗ of the `SpanProcessor`, the proposer check (`util.IsProposer`) is performed only after multiple RPC calls are made—such as retrieving the latest span from Heimdall, the latest block from Bor, and the latest milestone. However, only the proposer node proceeds with proposing or backfilling spans, making these early calls unnecessary for non-proposer nodes.

Problem Scenarios

When multiple nodes run `checkAndPropose`, each performs several expensive RPC calls:

- `getLastSpan()` → Heimdall
- `GetBorChainBlock()` → Bor
- `GetLatestMilestone()` → Heimdall

Even though only the actual proposer will proceed, all other nodes perform these calls needlessly. This results in:

- Increased load on Heimdall and Bor RPC services
- Wasted compute and network resources
- Slower responsiveness and efficiency on non-proposer nodes

Recommendation

Move the `isProposer` check to the top of the `checkAndPropose` function. This will short-circuit execution early for non-proposer nodes and prevent unnecessary external call.

Unnecessary Heimdall span fetch calls on every sprint cause redundant load

Severity Low **Impact** 1 - Low **Exploitability** 1 - Low

Type Implementation **Status** Acknowledged

Involved artifacts

- bor/consensus/bor/bor.go↗

Description

In the `checkAndCommitSpan` function, the code calls `updateLatestHeimdallSpanV1/V2` on every sprint, even though the span on Heimdall changes only once every 200 sprints. These update functions fetch the latest span from Heimdall and store it locally only if the span ID has advanced.

Since Heimdall does not modify a span once it is proposed, and span IDs are strictly increasing and immutable mid-span, calling the update logic on every sprint results in ~199 redundant Heimdall API calls per span.

Problem Scenarios

- The mainnet configuration uses 200 sprints per span.
- During each sprint, `checkAndCommitSpan` is triggered.
- Each call to `checkAndCommitSpan` leads to a call to `updateLatestHeimdallSpanV1/V2`, which attempts to fetch the latest span via the Heimdall client.
- As a result, for 199 out of 200 sprints, the fetched span ID remains unchanged and the function exits early, having no effect.
- Despite this, the API call is still executed, leading to:
  - Waste of computation and network resources
  - Increased log noise
  - Potential delays or failure propagation if Heimdall is slow or unreachable

Recommendation

1. Move the call to `updateLatestHeimdallSpanV1/V2` inside the conditional block that checks `needToCommitSpan`. This condition only becomes true near the end of a span, when the next span is actually needed. This change would eliminate ~199 unnecessary Heimdall API calls per span while preserving correctness and system behavior.

2. Once a new span is successfully fetched from Heimdall and stored locally (e.g., span 101), suppress further Heimdall span fetch attempts until that span is committed on-chain via `FetchAndCommitSpan`.

   This improvement targets how Heimdall operates—once it detects Bor is executing span X, it begins creating span X+1 and may propose it early within the 1–200 sprint window. The recommendation is to allow Heimdall fetches every sprint, but only until the new span (X+1) is successfully fetched and stored. After that, skip further Heimdall calls until the span is committed, reducing unnecessary fetch attempts while preserving early responsiveness.

Overuse of maxBlockNumber variable reduces clarity

Severity Informational **Impact** 0 - None **Exploitability** 0 - None

Type Implementation **Status** Acknowledged

Involved artifacts

- heimdall-v2/bridge/processor/span.go↗
- heimdall-v2/x/bor/keeper/side_msg_server.go↗

Description

The code uses a `maxBlockNumber` variable (the maximum of the latest milestone end block and Bor block number) in multiple control paths, even though only one of these values is relevant in each context. This reduces readability and may confuse the distinction between milestone-finalized and simply produced blocks.

Recommendation

Use `latestMilestoneEndBlock` exclusively for span backfill checks, and `latestBorBlockNumber` for span proposal and "close to span end" logic. This improves code clarity and intent.

```
if latestMilestoneEndBlock > lastSpan.EndBlock {
  sp.Logger.Debug("Backfilling spans up to milestone-finalized block",
↪  "milestoneEndBlock", latestMilestoneEndBlock)
  go sp.backfillSpans(ctx, latestBorBlockNumber, lastSpan)
  return
}

if latestBorBlockNumber > lastSpan.EndBlock {
  if types.IsBlockCloseToSpanEnd(latestBorBlockNumber, lastSpan.EndBlock) {
    sp.Logger.Debug("Current bor block is close to last span end block, skipping
↪  proposing span", "currentBlock", latestBorBlockNumber, "lastSpanEndBlock",
↪  lastSpan.EndBlock)
    return
  }

  nextSpanMsg, err := sp.fetchNextSpanDetails(lastSpan.Id+1, lastSpan.EndBlock+1)
  if err != nil {
    sp.Logger.Error("Unable to fetch next span details", "error", err,
↪  "lastSpanId", lastSpan.Id)
    return
  }

  go func() {
    defer func() {
      if r := recover(); r != nil {
        sp.Logger.Error("Recovered panic in propose goroutine", "panic", r)
      }
    }()
    if err := sp.propose(ctx, lastSpan, nextSpanMsg); err != nil {
      sp.Logger.Error("Error in propose", "error", err)
    }
  }()
}
```

Unhandled errors and panics in goroutines

Severity Informational **Impact** 0 - None **Exploitability** 0 - None

Type Implementation **Status** Acknowledged

Involved artifacts

- heimdall-v2/bridge/processor/span.go↗

Description

The current implementation launches `sp.backfillSpans(...)` and `sp.propose(...)` as goroutines without handling returned errors or potential panics. This can cause critical failures to go undetected, reducing system observability and complicating debugging.

Recommendation

Wrap each goroutine in a `defer`-based recovery block and ensure errors are logged.

```
if maxBlockNumber > lastSpan.EndBlock {
  if latestMilestoneEndBlock > lastSpan.EndBlock {
    sp.Logger.Debug("Bor self commmitted spans, backfill heimdall to fill missing
↪  spans", "currentBlock", latestBlock.Number.Uint64(), "lastSpanEndBlock",
↪  lastSpan.EndBlock)
    go func() {
      defer func() {
        if r := recover(); r != nil {
            sp.Logger.Error("Recovered panic in backfillSpans goroutine",
↪  "panic", r)
        }
      }()
      if err := sp.backfillSpans(ctx, latestBorBlockNumber, lastSpan); err != nil
       ↪  {
        sp.Logger.Error("Error in backfillSpans", "error", err)
      }
    }()
    return
  } else {
    sp.Logger.Debug("Will not backfill heimdall spans, as latest milestone end
↪  block is less than last span end block", "currentBlock",
↪  latestBlock.Number.Uint64(), "lastSpanEndBlock", lastSpan.EndBlock)
    return
  }
} else {
  if types.IsBlockCloseToSpanEnd(maxBlockNumber, lastSpan.EndBlock) {
    sp.Logger.Debug("Current bor block is close to last span end block, skipping
↪  proposing span", "currentBlock", latestBlock.Number.Uint64(),
↪  "lastSpanEndBlock", lastSpan.EndBlock)
    return
  }

  nextSpanMsg, err := sp.fetchNextSpanDetails(lastSpan.Id+1, lastSpan.EndBlock+1)
  if err != nil {
```

```
    sp.Logger.Error("Unable to fetch next span details", "error", err,
↪   "lastSpanId", lastSpan.Id)
    return
  }

  go func() {
    defer func() {
      if r := recover(); r != nil {
          sp.Logger.Error("Recovered panic in propose goroutine", "panic", r)
      }
    }()
    if err := sp.propose(ctx, lastSpan, nextSpanMsg); err != nil {
      sp.Logger.Error("Error in propose", "error", err)
    }
  }()
}
```

Lack of documentation for span backfill finalization condition

Severity Informational **Impact** 0 - None **Exploitability** 0 - None

Type Documentation **Status** Acknowledged

Involved artifacts

- `heimdall-v2/bridge/processor/span.go`↗

Description

The span backfill logic only triggers when `latestMilestoneEndBlock > lastSpan.EndBlock`, thereby ensuring that spans are only backfilled for blocks already finalized by a milestone. This is a sound safety check, but the rationale is not explicitly documented in the code.

Problem Scenarios

Future maintainers or auditors may misinterpret the logic or unintentionally weaken this important finality guarantee.

Recommendation

Add a clear inline comment explaining that span backfill should only occur for blocks finalized by a milestone, and not merely based on Bor's latest block number.

Redundant call to CalculateSprint in needToCommitSpan

Severity Informational **Impact** 0 - None **Exploitability** 0 - None

Type Implementation **Status** Acknowledged

Involved artifacts

- `consensus/bor/bor.go`↗

Description

The function `needToCommitSpan` calls `c.config.CalculateSprint(headerNumber)` twice within the same conditional block↗. Since the result is deterministic for a given `headerNumber`, this value can be computed once and reused.

Problem Scenarios

Calling `CalculateSprint` twice with the same input adds unnecessary computation and slightly reduces readability. While not a major performance issue, it is an avoidable inefficiency.

Recommendation

Cache the result of `CalculateSprint(headerNumber)` in a local variable before the conditional check:

```
sprint := c.config.CalculateSprint(headerNumber)
if currentSpan.EndBlock > sprint && currentSpan.EndBlock-sprint+1 == headerNumber
↪  {
  return true
}
```

Redundant sprint start check in checkAndCommitSpan

Severity Informational **Impact** 0 - None **Exploitability** 0 - None

Type Implementation **Status** Acknowledged

Involved artifacts

- consensus/bor/bor.go↗

Description

The function `checkAndCommitSpan` includes a check↗ to see if the current block is the start of a sprint:

```
if header.Number.Uint64()%c.config.CalculateSprint(header.Number.Uint64()) == 0 {
  ...
}
```

However, this exact check is already performed in both `Finalize`↗ and `FinalizeAndAssemble`↗ before calling `checkAndCommitSpan`.

Problem Scenarios

This duplicated condition adds unnecessary logic and may confuse future maintainers. It gives the impression that `checkAndCommitSpan` is expected to be called in other contexts too — but in reality, it's only ever called after the sprint-start check has already passed.

Recommendation

Remove the sprint-start check from `checkAndCommitSpan` to avoid redundant logic and make the control flow clearer. The check should remain only in `Finalize` and `FinalizeAndAssemble`, where it is correctly used to gate the call to `checkAndCommitSpan`.

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)

ImpactScore	Examples
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.

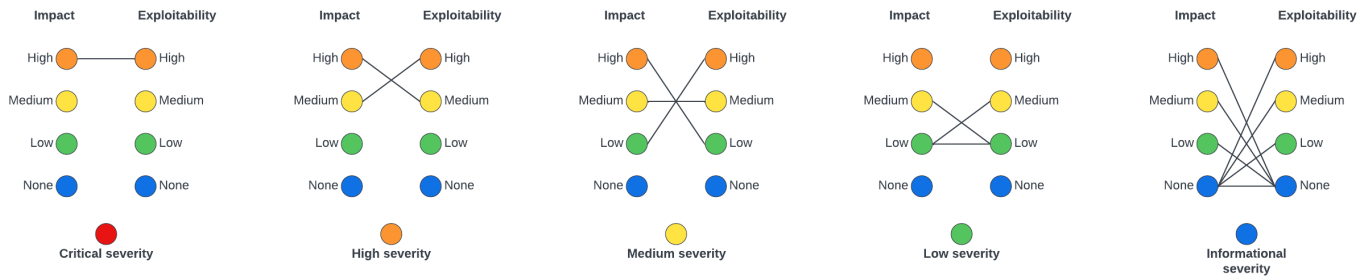

Figure 1: Severity classification

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.