Audit Report

Dymension Phase 1 Stream 3: Dymint

v1.0

July 9, 2024

Table of Contents

License

Disclaimer

This audit has been performed by

Oak Security GmbH

https://oaksecurity.io/
info@oaksecurity.io

Introduction

Purpose of This Report

Oak Security has been engaged by Dymension Technologies Ltd to perform a security audit of Dymint.

The objectives of the audit are as follows:

1. Determine the correct functioning of the protocol, in accordance with the project specification.
2. Determine possible vulnerabilities, which could be exploited by an attacker.
3. Determine smart contract bugs, which might lead to unexpected behavior.
4. Analyze whether best practices have been applied during development.
5. Make recommendations to improve code safety and readability.

This report represents a summary of the findings.

As with any code audit, there is a limit to which vulnerabilities can be found, and unexpected execution paths may still be possible. The author of this report does not guarantee complete coverage (see disclaimer).

Codebase Submitted for the Audit

The audit has been performed on the following target:

Repository	https://github.com/dymensionxyz/dymint
Commit	`19f8877fe96fb8d3f8b08bd541033b0d9536c7ce`
Scope	All contracts were in scope.
Fixes verified at commit	`7290af6f00887d09862f32bf43169f42a416e87e` Note that only fixes to the issues described in this report have been reviewed at this commit. Any further changes such as additional features have not been reviewed.

Methodology

The audit has been performed in the following steps:

1. Gaining an understanding of the code base's intended purpose by reading the available documentation.
2. Automated source code and dependency analysis.
3. Manual line-by-line analysis of the source code for security vulnerabilities and use of best practice guidelines, including but not limited to:
    a. Race condition analysis
    b. Under-/overflow issues
    c. Key management vulnerabilities
4. Report preparation

Functionality Overview

Dymint is an ABCI-client implementation for Dymension's autonomous RollApps, which can be used as a replacement for CometBFT or Tendermint in any ABCI-compatible blockchain application.

How to Read This Report

This report classifies the issues found into the following severity categories:

Severity	Description
Critical	A serious and exploitable vulnerability that can lead to loss of funds, unrecoverable locked funds, or catastrophic denial of service.
Major	A vulnerability or bug that can affect the correct functioning of the system, lead to incorrect states or denial of service.
Minor	A violation of common best practices or incorrect usage of primitives, which may not currently have a major impact on security, but may do so in the future or introduce inefficiencies.
Informational	Comments and recommendations of design decisions or potential optimizations, that are not relevant to security. Their application may improve aspects, such as user experience or readability, but is not strictly necessary. This category may also include opinionated recommendations that the project team might not share.

The status of an issue can be one of the following: **Pending, Acknowledged**, or **Resolved**.

Note that audits are an important step to improving the security of smart contracts and can find many issues. However, auditing complex codebases has its limits and a remaining risk is present (see disclaimer).

Users of the system should exercise caution. In order to help with the evaluation of the remaining risk, we provide a measure of the following key indicators: **code complexity**, **code readability**, **level of documentation**, and **test coverage**. We include a table with these criteria below.

Note that high complexity or low test coverage does not necessarily equate to a higher risk, although certain bugs are more easily detected in unit testing than in a security audit and vice versa.

Code Quality Criteria

The auditor team assesses the codebase's code quality criteria as follows:

Criteria	Status	Comment
Code complexity	Medium-High	The extensive use of concurrency and the need for synchronization of shared states increase the difficulty of reasoning about the correctness of the code.
Code readability and clarity	High	-
Level of documentation	High	-
Test coverage	Medium	`go test` reports an average test coverage of 61%.

Summary of Findings

No	Description	Severity	Status
1	Peer nodes could receive blocks that are not applied by the sequencer	Critical	Resolved
2	HTTP server misconfiguration allows Slowloris DoS attacks	Major	Resolved
3	CONTINUATION frames flood vulnerability in `x/net` allows attackers to DoS the node	Major	Resolved
4	Incorrect mempool initialization height leads to discarded transactions	Major	Resolved
5	Submitting blocks to the data availability and settlement layer can result in a deadlock	Major	Resolved
6	Invalid blocks can be received via p2p gossip	Major	Resolved
7	Non-atomic batch submission to data availability and settlement layers causes repayment for the same data and potential indefinite failure	Major	Resolved
8	Applying blocks concurrently can lead to unexpected errors	Major	Resolved
9	Stopped block production is immediately resumed after receiving a health status event from the data availability or settlement layer	Minor	Resolved
10	Pending blocks are repeatedly gossiped	Minor	Acknowledged
11	Validating gossiped transactions may block the libp2p validator queue	Minor	Resolved
12	Mempool is initialized at genesis without the `preCheck` and `postCheck` functions, leading to not validating the transaction's size and wanted gas until the first block is committed	Minor	Resolved
13	Maximum number of peer IDs can potentially be reached, preventing gossiped transactions from being added to the mempool	Minor	Acknowledged
14	Incomplete validation of configuration parameters	Minor	Resolved
15	Node can be started without providing the genesis	Minor	Resolved

16	Suboptimal data retrieval due to static configurations	Minor	Acknowledged
17	Occurrence of "Tendermint" instead of "Dymint"	Informational	Acknowledged
18	Non-sequencer nodes unnecessarily register health status event listener	Informational	Resolved
19	Use of magic numbers decreases maintainability	Informational	Resolved

Detailed Findings

1. Peer nodes could receive blocks that are not applied by the sequencer

Severity: Critical

The `produceBlock` function, defined in `block/produce.go:75-161`, is responsible for producing and distributing new blocks to peers.

However, since in `block/produce.go:148`, the `gossipBlock` function is optimistically invoked before the block is actually applied via the `applyBlock` function, there is a risk of gossiping blocks to peers even if applying the block errors or if the process stops for any reason after the execution of `gossipBlock` but before the execution of `applyBlock`.

Such an error will lead to peers storing and applying a block that the proposer will not apply, leading to an inconsistent block and transaction execution.

Recommendation

We recommend applying the block before initiating the P2P gossip.

Status: Resolved

2. HTTP server misconfiguration allows Slowloris DoS attacks

Severity: Major

The `serve` function in `rpc/server.go:177-184` instantiates an HTTP server and enables listening for incoming requests on the `listener` address.

However, since there is no `ReadHeaderTimeout` to handle idle connections, the server is vulnerable to Slowloris Denial-of-Service (DoS) attacks.

This attack method operates by transmitting large amounts of data slowly, which succeeds in keeping the connection alive in the event of a timeout, ultimately resulting in a DoS of the node.

Recommendation

We recommend defining the `ReadHeaderTimeout` for the HTTP server.

Status: Resolved

3. CONTINUATION frames flood vulnerability in `x/net` allows attackers to DoS the node

Severity: Major

The golang.org/x/net package used in Dymint is vulnerable to CONTINUATION frames flood, as reported in https://pkg.go.dev/vuln/GO-2024-2687.

"An attacker may cause an HTTP/2 endpoint to read arbitrary amounts of header data by sending an excessive number of CONTINUATION frames. Maintaining HPACK state requires parsing and processing all HEADERS and CONTINUATION frames on a connection. When a request's headers exceed MaxHeaderBytes, no memory is allocated to store the excess headers, but they are still parsed. This permits an attacker to cause an HTTP/2 endpoint to read arbitrary amounts of header data, all associated with a request which is going to be rejected. These headers can include Huffman-encoded data which is significantly more expensive for the receiver to decode than for an attacker to send."

Consequently, attackers can leverage this vulnerability to perform a DoS attack against the RPC server.

Recommendation

We recommend updating the [`golang.org/x/net`](golang.org/x/net) package to `v0.23.0` and `go` to version `go1.22.2` or later, as recommended in the CVE.

Status: Resolved

4. Incorrect mempool initialization height leads to discarded transactions

Severity: Major

In `node/node.go:177`, during the execution of the `NewNode` function, the `NewTxMempool` function is called to instantiate a new mempool.

However, since the mempool height is set to zero, incoming transactions would get incorrectly assigned a height of zero. As a consequence, the mempool would incorrectly and prematurely purge valid transactions in the `purgeExpiredTxs` function defined in `mempool/v1/mempool.go:762`.

Recommendation

We recommend initializing the mempool with the actual height.

Status: Resolved

5. Submitting blocks to the data availability and settlement layer can result in a deadlock

Severity: Major

In `block/submit.go:13-57`, the `SubmitLoop` function, running in a goroutine, submits blocks to the data availability layer (DA) and settlement layer (SL). A mutex, `batchInProcess`, is used to ensure that only a single batch is processed at a time.

However, if the `submitNextBatch` function in line `46` errors, the mutex is not released, preventing the next batch from being processed and resulting in a deadlock.

Recommendation

We recommend releasing the mutex in case of an error.

Status: Resolved

6. Invalid blocks can be received via p2p gossip, potentially preventing block syncing

Severity: Major

In `block/manager.go:256-278`, the `applyBlockCallback` callback function is called whenever a new block is received via libp2p gossip. If the received block height is equal to the next expected height, the block is applied via the `applyBlock` function in line `269`. Otherwise, if the block height is for a future height, the block and the corresponding commit are stored in `prevBlock` and `prevCommit`, respectively.

However, at this point, the block is not yet validated by verifying the signature to ensure the sequencer has produced it. This stateful check is only performed later in the `executeBlock` function in `block/block.go:180-182` via the `Validate` function.

As a result, invalid blocks can be received and processed by nodes, leading to potential issues such as:

1. If the block height is equal to the next expected height, it is attempted to be applied. Before any stateful validations, the block is saved via the `SaveBlock` function in `block/block.go:38`. While the block is ultimately dismissed and not applied due to not passing the signature verification check, it remains stored, wasting disk space of the node. An attacker could exploit this behavior by spamming invalid blocks to nodes.
2. If the received block is for a future height, the block and commit are stored in `prevBlock` and `prevCommit`, potentially overwriting a valid block and commit. As a result, nodes can be prevented from syncing via this mechanism. Please note that nodes also synchronize via the settlement layer (Dymension Hub), which mitigates this issue to some extent.

Additionally, it should be noted that the sequencer does not receive blocks via libp2p gossip, only regular nodes do.

Recommendation

We recommend executing the stateful block validation in the `applyBlockCallback` function before proceeding.

Status: Resolved

7. Non-atomic batch submission to data availability and settlement layers causes repayment for the same data and potential indefinite failure

Severity: Major

In `block/submit.go:89-103`, in the `submitNextBatch` function, a batch is submitted to the data availability (DA) layer by calling `SubmitBatch`, followed by an attempt to submit the same batch to the settlement layer (SL) using `m.settlementclient.submitbatch`.

If the SL submission fails due to reasons such as insufficient gas or censorship at the settlement layer, the transaction remains in the mempool awaiting inclusion. The system then retries to submit the same batch to both layers after the waiting period ends. Since `syncTarget` remains unchanged, the DA layer accepts the redundant batch, leading to unnecessary payments for already existing data. If the original transaction eventually gets included in the SL before the new transaction, the SL rejects the resubmitted transaction due to a height check requirement.

Consequently, this cycle of failing settlement attempts and redundant payments perpetuates indefinitely and incurs unnecessary costs for the system.

Recommendation

We recommend checking whether data is already submitted to the DA layer before resubmitting it.

Status: Resolved

8. Applying blocks concurrently can lead to unexpected errors

Severity: Major

The `applyBlock` function executes the current block by calling the `executeBlock` function in `block/block.go:44`. As part of the execution, the application's `BeginBlock`, `EndBlock`, and `DeliverTx` are called for every included transaction.

However, applying the block to the store and the ABCI application is not atomic, shared state is not locked between processes, such as the goroutine that processes libp2p gossiped blocks and the sync goroutine that syncs the node with blocks from the settlement layer.

In the rare case that a block for the same height is simultaneously processed by both goroutines, both processes will attempt to execute the block, leading to unexpected errors such as a panic in the ABCI application when [checking the header's validity](#).

Recommendation

We recommend adding a synchronization lock to the manager state to prevent concurrent block processing.

Status: Resolved

9. Stopped block production is immediately resumed after receiving a health status event from the data availability or settlement layer

Severity: Minor

In `block/produce.go:17-73`, the `ProduceBlockLoop` function produces new blocks unless the `shouldProduceBlocksCh` channel receives a `false` value. In this case, block production is stopped, and the channel waits for a `true` value again.

There are three causes that may stop the block production:

1. the data availability (DA) layer is unhealthy,
2. the settlement layer (SL) is unhealthy, or
3. block production encounters an error.

The DA and SL's current health status is continuously reported to the `healthStatusEventCallback` health listener function, defined in `block/manager.go:250-254`. This function sends the received health boolean value into the `shouldProduceBlocksCh` channel.

However, if block production was previously stopped due to erroneous block production, the health listener will enable block production again as soon as the DA or SL health status is reported as `true`, even if the error that caused the block production to stop has not been resolved.

Recommendation

We recommend not enabling block production when receiving a health status event if the block production was previously stopped by a block production error.

Status: Resolved

10. Pending blocks are repeatedly gossiped

Severity: Minor

In `block/produce.go:148`, the `produceBlock` function gossips a newly produced block to its peers. However, if the sequencer picks up a pending block that has already been gossiped, it will be gossiped again, which is unnecessary.

We classify this issue as minor, as repeatedly gossiped blocks do not harm the network besides causing networking inefficiencies.

Recommendation

We recommend only gossiping new blocks.

Status: Acknowledged

11. Validating gossiped transactions may block the libp2p validator queue

Severity: Minor

In `p2p/validator.go:43-69`, the `TxValidator` libp2p validator function calls the mempool's `CheckTx` function to validate the gossiped transaction prior to adding it to the mempool.

However, if the `CheckTx` function returns an error that does not match any of the `switch` cases, i.e., the `default` case, the function will attempt to read from the `checkTxResCh` channel to receive the ABCI response. As `CheckTx` may have errored before initiating an ABCI `CheckTx` request, the channel will not receive any response, causing the goroutine to block and clog the libp2p validator queue.

Recommendation

We recommend handling all other errors and returning `false` in this case. Additionally, consider setting a timeout for the transaction validator.

Status: Resolved

12. Mempool is initialized at genesis without the `preCheck` and `postCheck` functions, leading to not validating the transaction's size and wanted gas until the first block is committed

Severity: Minor

In `state/executor.go:301`, the `commit` function calls the mempool's `Update` function to update the `preCheck` and `postCheck` functions with the new `maxBytes` and `maxGas` consensus parameters received from the underlying application. Those functions are used to additionally validate that the transaction's size (in bytes) is less than the maximum and that the transaction's wanted gas does not exceed the block gas limit. This validation occurs before and after a transaction is added to the mempool.

However, at genesis, the mempool is not initialized with those `preCheck` and `postCheck` functions, even though both consensus parameters, `maxBytes`, and `maxGas`, are made available after executing `InitChain` in `block/initchain.go:45`.

As a result, transactions will not be validated until the first block is committed.

Recommendation

We recommend initializing the mempool after executing `InitChain`.

Status: Resolved

13. Maximum number of peer IDs can potentially be reached, preventing gossiped transactions from being added to the mempool

Severity: Minor

In `node/mempool/mempool.go:35-48`, the `nextPeerID` function returns the next unused peer ID to use and panic if the number of active IDs exceeds the maximum threshold, `maxActiveIDs = math.MaxUint16`.

However, over time, as more and more peer IDs are utilized, and unused IDs are not reclaimed via the `Reclaim` function, the maximum threshold can potentially be reached at some point. As a result, the libp2p transaction validator function, `TxValidator`, will be unable to claim an ID for a new peer and panics, preventing incoming transactions from being added to the mempool.

We classify this issue as minor as it requires a total number of 65,535 unique peers to have been connected to the node at some point.

Recommendation

We recommend using the `Reclaim` function to reclaim unused peer IDs.

Status: Acknowledged

14. Incomplete validation of configuration parameters

Severity: Minor

In `config/config.go:137-144`, the `Validate` function is executed by the `GetViperConfig` function execution to validate the `NodeConfig`.

However, the validation process is incomplete, as only `BlockManagerConfig` and `SettlementLayer` are currently checked.

Consequently, the node may use an invalid configuration, possibly leading to unintended behaviors and node halt.

Recommendation

We recommend updating the `Validate` function to include validation for `Instrumentation`, `DALayer`, `DAConfig`, and `DAGrpc` alongside `BlockManagerConfig` and `SettlementLayer`.

Status: Resolved

15. Node can be started without providing the genesis

Severity: Minor

In `cmd/dymint/commands/start.go:128-135`, the `checkGenesisHash` function is executed during the node initialization to verify that the provided `genesisHash` and the SHA-256 hash of the genesis file are equal.

However, if the `Genesis` file or the `genesisHash` is not provided, it does not generate an error, thus permitting the node to start even without providing the genesis.

Recommendation

We recommend triggering an error in case the genesis is not provided and halting the node startup process.

Status: Resolved

16. Suboptimal data retrieval due to static configurations

Severity: Minor

The `fetchBatch` function in `block/retriever.go:106` retrieves data batches using a retriever instance (`m.retriever`) initialized at node startup. However, this approach leads to suboptimal data retrieval due to static configurations. For instance, if sequencer 1 uses Celestia as the DA layer and sequencer 2 uses Avail, parsing data would fail for one another. This static configuration presents a significant issue, as nodes submitting to Celestia or Avail may face difficulties syncing data. Since `m.retriever` is statically configured based on the `daregsitry` settings, the syncing node might encounter situations where it's unable to parse both sets of data.

When data is submitted to the settlement layer, it includes a `DASubmitMetaData.client` field to validate the data availability from the client. Retrievers are configured to use only one DA client. If the settlement has data from mixed clients, the retriever can only parse data for its configured client. If the data is only on another client, this static configuration would prevent processing the data.

We classify this issue as minor because, at the time of the audit, only one sequencer node is used. Once further sequencer nodes are introduced, it becomes a serious issue.

Recommendation

We recommend parsing the data on a per-blob basis.

Status: Acknowledged

17. Occurrence of "Tendermint" instead of "Dymint"

Severity: Informational

In `rpc/client/client.go:168`, the `BroadcastTxCommit` function uses the error reason "Tendermint exited". This error message might be misleading as the client is called "Dymint".

Recommendation

We recommend amending the error message to use "Dymint" instead of "Tendermint".

Status: Acknowledged

18. Non-sequencer nodes unnecessarily register health status event listener

Severity: Informational

In `block/manager.go:243`, the `EventListener` function subscribes to the `EventQueryHealthStatus` event and registers the corresponding `healthStatusEventCallback` callback function. Whenever this event is emitted, the event's `Healthy` value is sent to the `shouldProduceBlocksCh` channel, potentially enabling or disabling block production.

However, this mechanism is only relevant for the sequencer node, as it is currently the only node that is producing blocks. For other regular nodes, the `shouldProduceBlocksCh` channel is not relevant, rendering the event listener and callback function unnecessary.

Recommendation

We recommend only registering the `healthStatusEventCallback` callback function for the sequencer, i.e., if `isAggregator` is `true`.

Status: Resolved

19. Use of magic numbers decreases maintainability

Severity: Informational

In `rpc/json/service.go:107-108` and `rpc/client/client.go:315`, hard-coded number literals without context or a description are used. Using such “magic numbers” goes against best practices as they reduce code readability and maintenance as developers are unable to understand their use easily and may make inconsistent changes across the codebase.

Recommendation

We recommend defining magic numbers as constants with descriptive variable names and comments, where necessary.

Status: Resolved