Security Audit Report

Celestia Rollkit Phase 1

Authors: Manuel Bravo, Andrija Mitrovic

Last revised 12 January, 2024

Table of Contents

Audit Overview

Scope

In December 2023 and January 2024, Informal Systems conducted a security audit for Rollkit framework. The audit aimed at inspecting the correctness and security properties of the framework, with a focus on the following components:

- `block`
- `executor`
- `store`
- `mempool`
- `da`
- This PR that adds logic to halt a node if applying a block fails.

This audit complements a previous audit conducted in November 2023 by Informal Systems that focused in the `block`, `state`, `store` and `mempool` modules.

The audit was performed from December 11, 2023 to January 10, 2024 by the following personnel:

- Manuel Bravo
- Andrija Mitrovic

Relevant Code Commits

The audited code was from one repository, rollkit/rollkit at the following commit:

- `rollkit/rollkit`: hash `eccdd0f1793a5ac532011ef4d896de9e0d8bcb9d`

The PR was not included in the commit hash. Thus, it is been audited separately.

Conclusion

We performed a thorough review of the project. We found some subtle problems - more on them in the section Findings. Those problems, if left unattended, would violate both Rollkit's safety and liveness properties.

We are glad to report that Rollkit's dev team has acknowledged our findings and is working towards fixing them.

Audit Dashboard

Target Summary

- **Type**: Protocol and Implementation
- **Platform**: CometBFT, Go
- **Artifacts**: rollkit/rollkit, PR

Engagement Summary

- **Dates**: 11.12.2023 -- 10.01.2024
- **Method**: Manual code review, protocol analysis

Severity Summary

Finding Severity	**#**
Critical	4
High	1
Medium	1
Low	0
Informational	4
Total	**10**

System Overview

Rollkit is a framework for building rollups on top of DA layers. The framework is built as an open-source project, making it easier for developers to reuse its four main components and customize their rollups. It is designed to offer developers flexibility and the ability to customize rollups as they see fit.

Transaction Flow

In this section, we describe the process through which a rollkit transaction passes from the time a user submits it until it is settled in the state of rollkit nodes. As the audited implementation, the transaction flow description assumes that there is a single trusted sequencer, which is key to guarantee that valid rollkit transactions are eventually committed to the state of rollkit nodes.

There are four main actors in a given rollup service:

- Users (aka light nodes) of the service.
- Rollup nodes (aka full nodes, or simply nodes): the actor that maintains a local copy of the rollup state and that users use to query it. The union of all rollup nodes forms the rollup network.
- The sequencer: a special node responsible for creating rollup blocks, publishing them in the DA layer and in the p2p layer.
- The DA layer: the data availability layer where the rollup blocks are published. It could be any DA layer in principle, but for simplicity, we assume that it is Celestia's DA layer.

The transaction flow can be divided in five high-level steps:

1. A user submits a transaction to any node. When a node receives a transaction, it adds to its mempool. Periodically, the node gossips transactions in its mempool to other nodes. Eventually, the sequencer receives the transaction.
2. When the sequencer receives a transaction (either sent by a user directly or through gossip), it adds it to its mempool. Periodically, the sequencer reaps its mempool without censoring transactions and creates rollup blocks. Thus, eventually, the sequencer includes the user's transaction in a rollup block.
3. The sequencer assigns consecutive timestamps (aka height) to rollup blocks and publishes them in the DA layer by submitting a transaction to the rollup namespace. The sequencer also gossips rollup blocks through the rollup network.
4. A node can receive rollup blocks from two sources: via the gossip layer or the DA layer (nodes periodically pull blocks from it). When a node receives a rollup block, it stores it locally. Nodes execute rollup blocks in block height order (after checking that the block is signed by the trusted sequencer). Within each block, transactions are ordered and executed in that order. Each node keeps track of the latest block height applied.
5. Validated blocks received from the DA layer are marked as hard confirmed. The current implementation only marks them. The idea is to use this to certify that the node's state (typically produced by applying blocks coming through the gossip layer) is consistent with the canonical rollup state (the one written in the DA layer).

Code Overview

As described in the Audit Overview section, the main focus of this audit is the rollkit's node implementation.

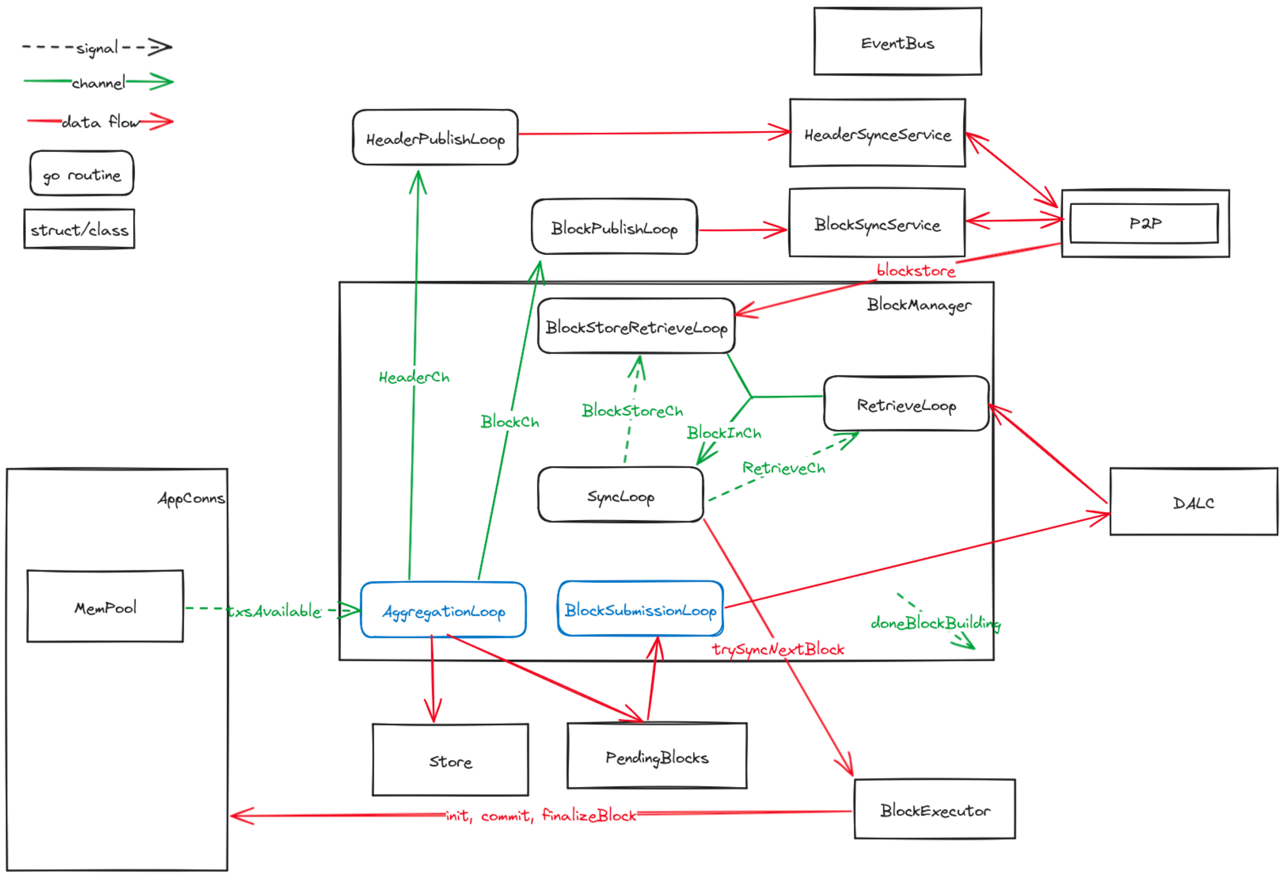
Node Initialization

On start, a non-sequencer node spawns three goroutines: `RetrieveLoop`, `BlockStoreRetrieveLoop`, and `SyncLoop`. Both the `RetrieveLoop` and `BlockStoreRetrieveLoop` are in charge of sending rollkit blocks to the `blockInCh` channel: the `RetrieveLoop` retrieves them from the DA layer and the `BlockStoreRetrieveLoop` receives them from the p2p layer. The `SyncLoop` is then responsible for pulling blocks from the `blockInCh` channel, filtering out non-valid blocks, and executing them in rollkit height order. The `SyncLoop` is also used to signal both the `RetrieveLoop` and `BlockStoreRetrieveLoop` by using the `retrieveCh` and `blockStoreCh` channels respectively.

The sequencer spawns four goroutines:

- The `AggregationLoop` responsible for creating rollkit blocks.
- The `BlockSubmissionLoop` responsible for publishing blocks in the DA layer.
- The `headerPublishLoop` responsible for publishing headers in the p2p layer.
- The `blockPublishLoop` responsible for publishing blocks in the p2p layer.

The figure below shows how the different goroutines coordinate through channels, the main components and data structures involved, and how the data flows.

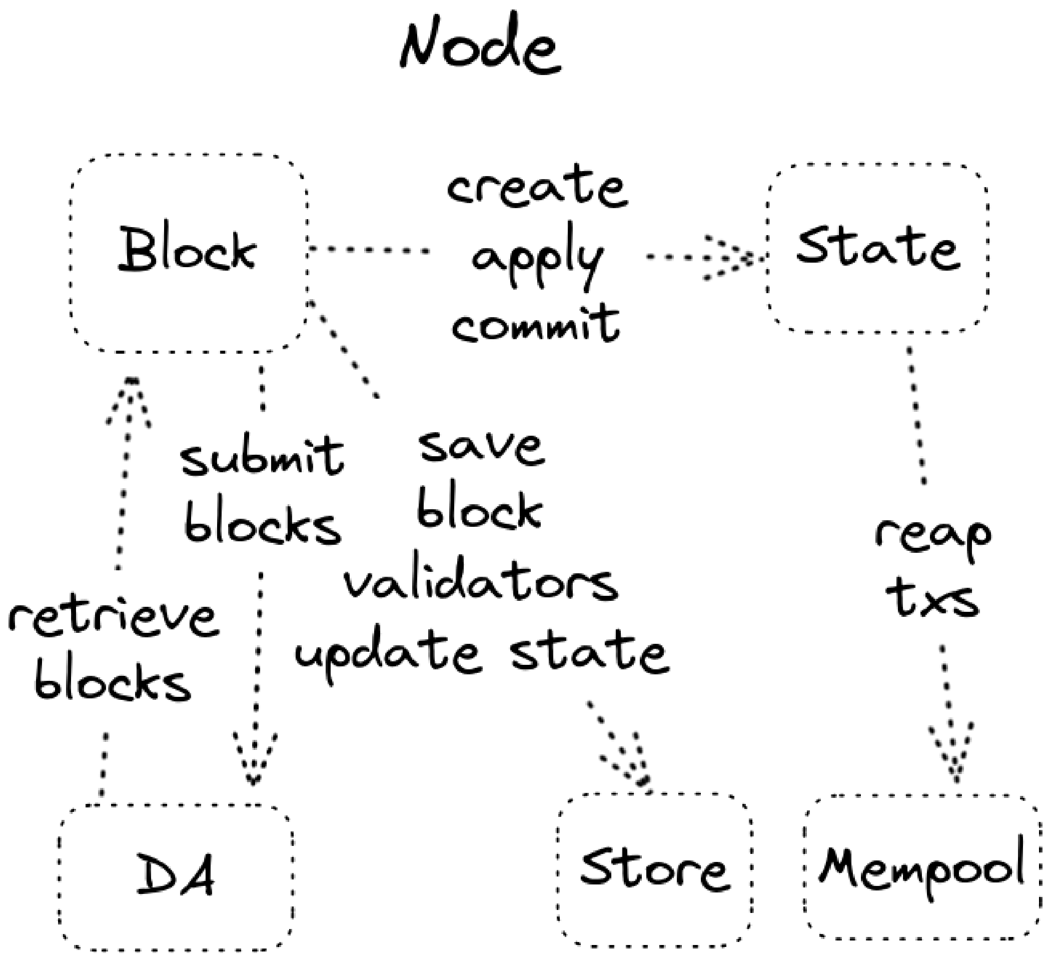


Modules Overview

In the audit, we focused on the following modules:

- `Block`, which defines how block production and syncing for the rollup is handled.
- `State`, which defines how a node executes rollup blocks.
- `Store`, which defines how the rollup state and blocks are stored.
- `Mempool`, which defines the mempool for the rollup node.
- `Da`, which defines how the rollup interacts with the DA layer.

The following diagram presents how these modules interact. We included it for completeness as it is the module in charge of interacting with the DA layer.



1 Rollkit components interaction diagram
We now describe the main functionality of each of the modules. A possibly more detailed description can be found in the project specification.

Block Module

The main component of the Block module is the block manager. The manager is responsible for block creation, block publication in the DA layer, gossiping blocks throughout the rollkit network, and orchestrating block execution.

Block Production. Only the sequencer can create blocks. Thus, if the node is the sequencer, it runs a separate go routine for aggregating blocks (`AggregationLoop` function). AggregationLoop function is responsible for aggregating transactions into rollup blocks. This function has two modes based on the value of `nodeConfig.LazyAggregator` attribute:

- normal mode - the block manager runs a timer, which is set to the `BlockTime` configuration parameter, and continuously produces blocks at `BlockTime` intervals (regulated by the getRemainingSleep function). This mode may generate empty blocks if the sequencer does not receive transactions for a while.
- lazy mode - the block manager begins constructing a block as soon as any transaction is detected in the mempool. It waits for a 1-second timer to maximize the inclusion of transactions, aligning with the default 1-second block time.

In either mode, the block creation is delegated to the publishBlock function. This function takes the following main steps to produce a block. We refer in multiple places to the executor. This component is described later as part of the State module.

1. The function first checks if the sequencer has already created a rollup block for the target height. If so, it retrieves it from the store and uses it. Otherwise, it creates a new block using the createBlock function of the executor.
2. The sequencer then saves the block to its store and applies it to its state using the executor's `ApplyBlock` function. At this point, the sequencer has not yet committed the resulting state.
3. It signs the block using the signing key to generate commitment.
4. It adds the block to a queue called `pendingBlocks` that the sequencer later uses to orchestrate the publication of blocks in the DA layer.
5. The sequencer then commits the new state using the executor's `Commit` function.
6. Finally, the sequencer sends the header and block to the `HeaderCh` and `BlockCh` channels. Both channels are used for gossiping headers and blocks throughout the rollkit network.

Block Publication. The sequencer is responsible for publishing the rollkit blocks it creates in the DA layer. This happens in a continuous loop that is started as a separate go routine (`BlockSubmissionLoop`) when the node starts.

The function `BlockSubmissionLoop` calls the `submitBlocksToDA` function every `DABlockTime` . The submitBlocksToDA function reads the `pendingBlocks` map and tries to publish all retrieved rollkit blocks in the DA layer. The function tries a maximum of `maxSubmitAttempts` (30 by default). The function sleeps in between failing attempts. The sleeping time is initially 100ms (`initialBackoff`) and increases every unsuccessful attempt exponentially (it doubles every time until it reaches a max of `DABlockTime`). When blocks are successfully submitted, the submitted blocks are removed from the `pendingBlocks` map.

Gossiping Headers and Blocks. The sequencer is responsible for gossiping the headers and blocks it creates. When the sequencer creates a block it sends its header and the block itself to the `HeaderCh` and `BlockCh` channels.

The sequencer runs two separate go routines (this for headers and this for blocks) that actively read these channels and call WriteToHeaderStoreAndBroadcast and WriteToBlockStoreAndBroadcast respectively to gossip them.

Note that in the current implementation, the sequencer sends headers but no node consumes them.

Block Retrieval from the DA Layer. Any node periodically retrieves blocks from the DA layer. For this, nodes execute a continuous loop (RetrieveLoop) in a separate routine.

RetrieveLoop periodically (every `DABlockTime` time) retrieves blocks from the DA layer in DA layer height order (`daHeight`). The block manager actively maintains and increments the daHeight counter after each DA block successful retrieval.

After a successful retrieval, retrieved blocks are first validated by checking whether they have been created by the trusted sequencer or not. Every block that passes the validation is marked as DA-included and sent to the `BlockInCh` channel if they have not been seen before.

Block Retrieval from the p2p Layer. When a node receives a block from the p2p layer, it stores it in the block store. Any node periodically retrieves blocks from the block stores. For this, nodes execute a continuous loop (BlockStoreRetrieveLoop) in a separate routine.

BlockStoreRetrieveLoop periodically retrieves blocks from the block store in block store height order (starting from zero). The retrieval interval is determined by the manager signaling the `blockStoreCh` channel at `blockTime` intervals. Each retrieved block is sent to the `blockInCh` channel.

Block Sync service is created during the node initialization by spawning the `SyncLoop` function in a separate go routine. SyncLoop is a loop that reads blocks from the `blockInCh` channel. When a block is retrieved, the function takes the following steps:

1. It first checks if it has seen the block already or if the node has executed a block for the same height. If any of the conditions is met, then the function skips the block.
2. If the node has never seen the block, it caches it.
3. It then calls the `trySyncNextBlock` function.
4. The `trySyncNextBlock` tries to execute as many blocks as possible as follows:
    a. It retrieves the height of the latest executed block `height`.
    b. It then tries to retrieve a block from the cache with height `height+1`.
    c. If it does not exist, then the function returns.
    d. If there exists a block for height `height+1` in the cache it first validates it.
    e. If the validation passes, it then applies it, saves it in the store, increases the store's height to indicate that the execution has already executed up to height `height+1`, and deletes the block from the cache.
    f. The function then goes to step a and tries to execute the next block.
5. If the `trySyncNextBlock` does not return an error, we mark the block as seen in the cache.

State Module

The main component of the State module is the block executor. It is responsible for the following actions:

- Creation of blocks (CreateBlock) - creates blocks by reaping (`ReapMaxBytesMaxGas` function from Mempool module) transactions from mempool, passing the set of prepared transactions to the app via ABCI, and building a block with the ABCI response.
- Execution of block (ApplyBlock) - validates and executes the given block.
- Committing the block (Commit) - commits and publishes events.

Mempool Module

Every node instantiates a local mempool. Rollkit uses a version of mempool called CListMempool. This is an ordered in-memory pool of transactions received either directly from users or via the p2p layer. Transaction validity is checked using the `CheckTx` ABCI message before the transaction is added to the mempool. The mempool uses a concurrent list structure for storing transactions that can be efficiently accessed by multiple concurrent readers. It is initialized on full node initialization here (NewCListMempool).

The Mempool module provides the following functionalities:

- ReapMaxBytesMaxGas reaps transactions from the mempool up to `maxBytes` bytes total with the condition that the total `gasWanted` must be less than `maxGas`. If both maxes are negative, there is no cap on the size of all returned transactions (~ all available transactions).
- EnableTxsAvailable initializes the `TxsAvailable` channel, ensuring it will trigger once every height when transactions are available.
- Lock and Unlock functions are used for safe concurrent use by multiple goroutines.
- FlushAppConn lushes the mempool connection to ensure async callback calls are done.
- Flush removes all transactions from the mempool and caches.
- CheckTx executes a new transaction against the application to determine its validity and whether it should be added to the mempool.

Store Module

The Store interface defines methods for storing and retrieving blocks, commits, and the state of the blockchain. The main methods are:

- Height: Returns the height of the highest block in the store.
- SetHeight: Sets given height in the store if it's higher than the existing height in the store.
- SaveBlock: Adds a block to the store along with it seen commit.
- GetBlock: Returns a block at a given height.
- UpdateState: Updates the state saved in the Store. Only one State is stored.

Da Module

The Da interface defines methods for submitting and retrieving blocks from the DA layer. It includes two methods:

- SubmitBlocks. The function receives a list of blocks ordered by height and attempts to submit as many as possible without exceeding the maximum blob size. If there is no error and at least one block is published in the DA layer, the function returns a counter indicating how many of the blocks have been successfully published. The function returns an error otherwise.
- RetriveBlocks. The function attempts to retrieve a set of blocks written in the DA layer at the rollup namespace at a given DA height. If it successfully retrieves a set of blocks, it filters out those that cannot be unmarshalled and returns the set of well-formed blocks to the caller function. It returns an error otherwise.

Threat Analysis

In our threat analysis, we start by defining a set of properties that are required for the correctness of rollkit. We separate them into invariants and progress properties. We have then analyzed each of them individually to see if they can be violated. A violation of a property is a potential threat, as detailed for each of the properties.

Additionally, we have inspected a set of threats related to memory usage, DoS attacks, and rollup initialization. We have listed them at the bottom of this section.

We have inspected the listed threats, resulting in the findings presented in the Findings section.

Invariants

1. The mempool filters out duplicates when transactions are received either from the p2p layer or directly from users.

- Verify that when a node receives a transaction, it only adds the transaction to its mempool if it is not already there.
- Verify that the node keeps track of already committed transactions to filter out duplicates when these are received at a later point in time.

Threat: User transactions are included in the rollup multiple times. Such behavior harms the users of the rollup.

Conclusion: The invariant is not theoretically guaranteed because the mempool cache has a maximum capacity - this property is inherited from CometBFT. However, for practical purposes, the cache capacity is large enough to prevent duplicate transactions.

2. Once a transaction is in the mempool, the sequencer includes it in at most one block, and only once.

- Verify that once a transaction is included in a committed rollup block, it is always removed from the mempool

Threat: User transactions are included in the rollup multiple times. Such behavior harms the users of the rollup.

Conclusion: The implementation ensures the invariant.

3. Rollup blocks only include well-formed transactions.

- Verify that the sequencer checks the validity of a transaction before adding it to its mempool.
- Verify that the sequencer only includes transactions from its mempool when it creates a block.

Threat: Non-well-formed transactions are included in a rollup block. This is a problem because the application may fail in execution, halting the rollup.

Conclusion: Rollkit guarantees that any transaction included in a rollup block has passed the `CheckTx` app call. It is then up to the application to guarantee the invariant. This finding is related to this invariant.

4. The sequencer does not equivocate. Assume that a rollup node executes block `b1` and that a second node executes block `b2`. If both blocks are created by the sequencer and `b1.height=b2.height`, then `b1=b2`. Note that the node could be the same in both cases and that the sequencer is considered a node.

- Verify that the sequencer only creates a new block for a given height when there is no block saved in its store for that height.
- Verify that once a block for a given height is saved in the store, this cannot be overwritten with a different block.
- Verify that before the sequencer executes a block, this is saved in its store.
- Verify that before a block is published in the DA layer: added to the `pendingBlocks` queue, the sequencer saves the block in its store.
- Verify that before a block is gossiped in the p2p layer, the sequencer saves the block in its store.

Threat: Nodes execute different blocks for the same height, breaking consensus.

Conclusion: The implementation ensures the invariant.

5. A node only executes rollup blocks created by the trusted sequencer.

- Verify that the blocks created by the sequencer are properly signed.
- Verify that nodes check the validity of a block before executing it.
- Verify that the validity check done by nodes before executing checks that the block was signed by the trusted sequencer.

Threat: Malicious nodes can create rollup blocks that other nodes would accept as valid and execute. This has all types of implications, e.g., nodes may execute different blocks at the same height, breaking consensus.

Conclusion: The implementation **does not** ensure the invariant. This describes the issue. This other finding is also related.

6. A node executes rollup blocks in increasing height order without skipping any height.

- Verify that a node only executes a block with height `h` is it has executed `h-1` already.
- Verify that this is also the case for the sequencer that executes blocks on creation as well.

Threat: Nodes that have executed the same prefix of blocks do not agree on the resulting state, even if all executed blocks were created by the trusted sequencer and there exists a single block per height.

Conclusion: The implementation ensures the invariant.

7. Rollup nodes pull DA blocks in strict height order: no skipping.

- Verify that nodes pull blocks in DA height order when no pull attempts fail.
- Verify that nodes pull blocks in DA height order even when some pull attempts fail.

Threat: Nodes may get stuck if they skip a DA block containing some rollup blocks because they never observe the block for a given height.

Conclusion: The implementation ensures the invariant.

8. The sequencer creates rollup blocks in increasing height order without skipping.

- Verify that if the sequencer creates a block for height `h`, it has previously created a block for height `h-1`.

Threat: Since nodes execute blocks in strict height order, nodes may get stuck if the sequencer skips heights on creation.

Conclusion: The implementation ensures the invariant.

9. If a node has received rollup blocks created by the sequencer up to height `h`, then it executes them.

- Verify that if a node receives blocks in order, it executes them as it receives them.
- Verify that if a node receives blocks out of order, it executes them as it completes prefixes.

Threat: Nodes may get stuck forever.

Conclusion: The implementation **does not** ensure the invariant. This finding shows an execution in which due to malicious block creators, nodes may be stuck forever.

Progress properties

10. If a node executes a rollup block, the block is guaranteed to be published in the DA layer.

- Verify that the sequencer publishes all created blocks when no publication attempt fails.
- Verify that the sequencer publishes all created blocks even when publication attempts fail.
- Verify that the interplay between the block creation and the DA block submission at the sequencer does not lead to violating the property.
- Verify that the sequencer never creates blocks greater than the `maxBlockSize`.
- Verify that only blocks published in the DA are removed from the `pendingBlocks` queue.
- Verify that the sequencer cleans its `pendingBlocks` after publishing a block in the DA layer.

Threat: Since nodes execute blocks in strict height order, nodes may get stuck: it is not guaranteed that all rollup blocks are received via the p2p layer, e.g., when a node joins after the rollup starts.

Conclusion: The implementation **does not** ensure the property. This finding describes an execution in which rollup blocks are not published in the DA because the sequencer creates blocks of size greater than the DA's `maxBlockSize`.

11. The sequencer does not censor transactions: If a valid transaction is received either through gossip or directly from a client, the sequencer eventually includes it in a block.

- Verify that if the sequencer receives a valid (according to the `CheckTx` app call) transaction, it adds it to its mempool.
- Verify that the sequencer reaps transactions from the mempool in order.
- Verify that transactions are only removed from the mempool when a block is committed and not before.

Threat: User transactions are never included in a block. Such behavior harms the users of the rollup.

Conclusion: The implementation ensures the property.

Other threats

Threat: Data structures grow without bound under the assumption that all nodes behave correctly

Conclusion: The implementation does not suffer from this issue.

Threat: Data structures grow without bound under the assumption that nodes may be malicious

Conclusion: This threat is significant, as it is possible due to no-sequencer nodes creating blocks. See this finding.

Threat: The implementation assumes specific initial configurations

Conclusion: The implementation assumes that the initial height is 1. See this finding.

Threat: Goroutine loops block each other permanently

Conclusion: The implementation correctly implements the synchronization between the loops.

Threat: The sequencer can be flooded with messages that impose a high computational load (DoS)

Conclusion: Transactions are pre-checked with respect to size, available pool size, and by the application before adding them to the mempool.

Findings

Title	Project	Type	Severity	Impact	Exploitability	Status	Issue
Blocks arriving out of order could halt the chain	Celestia Rollkit Phase 1	IMPLEMENTATION	4 CRITICAL	3 HIGH	3 HIGH	ACKNOWLEDGED	https://github.com/rollkit/rollkit/issues/1417
The authenticity of blocks received through the p2p layer is not checked	Celestia Rollkit Phase 1	IMPLEMENTATION	4 CRITICAL	3 HIGH	3 HIGH	ACKNOWLEDGED	https://github.com/rollkit/rollkit/issues/1417
checkCentralizedSequencer does not check that the block was created by the trusted sequencer	Celestia Rollkit Phase 1	IMPLEMENTATION	4 CRITICAL	3 HIGH	3 HIGH	ACKNOWLEDGED	https://github.com/rollkit/rollkit/issues/1417
Nodes aren't initialized properly, especially when InitialHeight isn't 1 in the genesis file	Celestia Rollkit Phase 1	IMPLEMENTATION	4 CRITICAL	3 HIGH	0 NONE	ACKNOWLEDGED	https://github.com/rollkit/rollkit/issues/1360
BlockCache can be filled with blocks from malicious nodes	Celestia Rollkit Phase 1	IMPLEMENTATION	3 HIGH	3 HIGH	2 MEDIUM	ACKNOWLEDGED	https://github.com/rollkit/rollkit/issues/1417

Title	Project	Type	Severity	Impact	Exploitability	Status	Issue
Large rollup blocks may compromise liveness	Celestia Rollkit Phase 1	IMPLEMENTATION	2 MEDIUM	3 HIGH	1 LOW	ACKNOWLEDGED	https://github.com/rollkit/rollkit/issues/1428
Miscellaneous code improvements	Celestia Rollkit Phase 1	IMPLEMENTATION PRACTICE	0 INFORMATIONAL	1 LOW	04 UNKNOWN	ACKNOWLEDGED	
The semantics of submitBlocksToDA is unclear	Celestia Rollkit Phase 1	IMPLEMENTATION	0 INFORMATIONAL	1 LOW	04 UNKNOWN	ACKNOWLEDGED	https://github.com/rollkit/rollkit/issues/1429
The current implementation assumes that a sequencer operates a single rollup	Celestia Rollkit Phase 1	IMPLEMENTATION	0 INFORMATIONAL	04 UNKNOWN	04 UNKNOWN	ACKNOWLEDGED	
Halt a node if it errors when applying a block from the sequencer	Celestia Rollkit Phase 1	IMPLEMENTATION	0 INFORMATIONAL	04 UNKNOWN	04 UNKNOWN	ACKNOWLEDGED	https://github.com/rollkit/rollkit/issues/1437

Blocks arriving out of order could halt the chain

Title	Blocks arriving out of order could halt the chain
Project	Celestia Rollkit Phase 1
Type	IMPLEMENTATION
Severity	4 CRITICAL
Impact	3 HIGH
Exploitability	3 HIGH
Status	ACKNOWLEDGED
Issue	https://github.com/rollkit/rollkit/issues/1417

Involved artifacts

- block/manager.go

Description

The chain may halt due to blocks arriving out of order while non-canonical blocks are created by malicious rollup nodes. The main cause is that validation is done too late and a node may overwrite a canonical block with a non-valid block. By *canonical* we mean that the trusted sequencer has created it.

Problem Scenarios

Assume that any user can create rollup blocks and gossip them. The following scenario could halt the chain:

1. Assume that full node `n` is at height `h`: it has executed all rollup blocks up to `h`. This means that `m.store.Height()=h`.
2. A non-canonical block for height `h+1` that the node has never seen comes through `m.blockInCh`. Since `m.store.Height()=h`, it will be set in `m.blockCache` for the height `h+1` here.
3. No block is executed because the block for height `h+1` is not valid.
4. Assume that we now received the canonical block for height `h+2` and save it in `m.blockCache` for the height `h+2` here.
5. Since we still have a non-valid block at height `h+1` we do not execute any block.
6. We now receive a non-canonical block for height `h+2` and overwrite the canonical one that we received in step 4. Again, since we still have a non-valid block at height `h+1` we do not execute any block.

7. We finally receive the canonical block for height `h+1`, cache it in `m.blockCache`, and call `trySyncNextBlock`.
8. The function executes the canonical block for height `h+1` but cannot execute any other block since it has overwritten the canonical block for height `h+2` in the cache with a non-valid one.
9. If the node never receives the canonical block for height `h+2` again, which is not guaranteed, then no more blocks are ever executed.

Recommendation

Do validation early such that nodes only store blocks created by the sequencer.

The authenticity of blocks received through the p2p layer is not checked

Title	The authenticity of blocks received through the p2p layer is not checked
Project	Celestia Rollkit Phase 1
Type	IMPLEMENTATION
Severity	4 CRITICAL
Impact	3 HIGH
Exploitability	3 HIGH
Status	ACKNOWLEDGED
Issue	https://github.com/rollkit/rollkit/issues/1417

Involved artifacts

- block/manager.go

Description

When a block is received via the block gossip service several checks are performed before executing the block on the state but none checks that the block has been created by the trusted sequencer.

This issue is related to this and this other findings.

Problem Scenarios

Malicious node may make rollup nodes execute non-canonical blocks received via the p2p layer created by a malicious node.

Recommendation

There should be a check before executing a block that this was created by the trusted sequencer. We recommend doing this as early as possible to avoid filling up data structures with non-valid blocks; similar to what's done for blocks read from the DA layer: there is a check in processNextDABlock:

```
                // received block is not from the expected centralized sequencer
                if !bytes.Equal(block.SignedHeader.ProposerAddress,
m.genesis.Validators[0].Address.Bytes()) {
                    continue
                }
```

checkCentralizedSequencer does not check that the block was created by the trusted sequencer

Title	checkCentralizedSequencer does not check that the block was created by the trusted sequencer
Project	Celestia Rollkit Phase 1
Type	IMPLEMENTATION
Severity	4 CRITICAL
Impact	3 HIGH
Exploitability	3 HIGH
Status	ACKNOWLEDGED
Issue	https://github.com/rollkit/rollkit/issues/1417

Involved artifacts

- types/signed_header.go
- block/manager.go

Description

Before executing a block in `trySyncNextBlock`, the node does several checks. One of the check is done by the `checkCentralizedSequencer` function. While the function naming implies that the function checks that the block is created by the trusted sequencer, it is not the case: the function only does some checks in the metadata included in the signed header, but it does not compare it with the hardcoded sequencer address in the genesis block. A malicious proposer could accordingly fill the metadadata and pass the check.

```
func (sh *SignedHeader) checkCentralizedSequencer() error {
    validators := sh.Validators.Validators
    if len(validators) != 1 {
        return errors.New("cannot have more than 1 validator (the centralized
sequencer)")
    }
    // make sure the proposer address is the same as the first validator in the
validator set
    first := validators[0]
    if !bytes.Equal(first.Address.Bytes(), sh.ProposerAddress) {
        return errors.New("proposer address in SignedHeader does not match the
expected centralized sequencer address")
    }
    // check proposer against the first validator in the validator set
    if !validatorsEqual(sh.Validators.Proposer, first) {
        return errors.New("proposer in sh.Validators does not match the expected
centralized sequencer")
    }
    return nil
}
```

Problem Scenarios

If the function is meant to check that the block was created by the trusted sequencer as implied by the naming, this could lead to nodes to execute blocks created by malicious nodes, as the authenticity of the block is not checked. This is specially relevant for blocks received through the p2p layer, as explained in this other finding.

Recommendation

If the plan is to fix this other finding by doing the authenticity check earlier (this would be our recommendation) then it should be enough to change the name of the function to better match its purpose.

If the plan is to do the authenticity check in the `checkCentralizedSequencer` function, the function should be modified to check that the trusted sequencer is the one signing the header.

Nodes aren't initialized properly, especially when InitialHeight isn't 1 in the genesis file

Title	Nodes aren't initialized properly, especially when InitialHeight isn't 1 in the genesis file
Project	Celestia Rollkit Phase 1
Type	IMPLEMENTATION
Severity	4 CRITICAL
Impact	3 HIGH
Exploitability	0 NONE
Status	ACKNOWLEDGED
Issue	https://github.com/rollkit/rollkit/issues/1360

Involved artifacts

- rollkit/block/manager.go / functions `NewManager` and `publishBlock`
- rollkit/store/store.go

Description

We have found that in some places the implementation assumes that the rollup initial height (in the genesis) is 1. A rollup may want to start at a height greater than 1 after a hard fork. In the current implementation, if the InitialHeight is greater than 1, the rollup won't be initialized properly. For instance, as described in the problem scenarios section the node's state is not initialized and the sequencer won't create blocks.

We've ranked the finding's exploitability as none given that it is an initialization problem that cannot be exploited. Nevertheless, we consider the issue critical since without fixing it, rollups won't be able to operate after a hard fork in some cases.

Problem Scenarios

Assume that a node `n` starts after a hard fork. This means that it has no state saved in its store. Assume that the initial height is `x>1` . Some of the problematic scenarios are the following:

Nodes do not initialize its state

- At the beginning of `NewManager()` , the function `getInitialState(store, genesis)` tries to read an existing state from the store. if none exists, it creates an initial state from the genesis file with

`LastBlockHeight = 0` and `InitialHeight = genesis.InitialHeight`. This would be the case for node `n`.
- A node is initialized only if `LastBlockHeight + 1 == genesis.InitialHeight` (here). Since `n` initialized `LastBlockHeight` to 0, we have that `LastBlockHeight + 1 = 1`. Also since `genesis.InitialHeight > 1`, we have that `LastBlockHeight + 1 != genesis.InitialHeight`. Thus, the node does not get initialized.

The sequencer won't produce blocks

1. Assume that `n` is the sequencer and that starts from a fresh state: one may want to hard fork to change the sequencer.
2. The store is initialized to height 0 (here) at every node, including the sequencer.
3. Since the sequencer starts from a fresh state, the call to `GetState` (here) errors, and the sequencer node does not update the store's height (here).
4. Assume the sequencer calls `publishBlock` for the first time.
5. By steps 2 and 3, we have that `m.store.Height()=0` and `newHeight=1` when the sequencer first calls `publishBlock`. Thus, the `publishBlock` function attempts to retrieve the commit for height 0 (here).
6. There is nothing in the store's db. Thus the function would return an error, which makes `publishBlock` return an error as well.
7. As a consequence, the sequencer is unable to produce blocks.

Recommendation

The initialization logic is scattered throughout the modules, and it is difficult to propose a list of changes that will make the overall initialization logic work properly. Instead of doing so, we recommend the team revisit the initialization logic while keeping in mind the following scenarios. We also include some suggestions that should be taken with a grain of salt.

- A node starts from a fresh state
  - In this case, independently from the initial height (whether it is 1 or greater), the node has to get initialized through the genesis file.
  - Note that if the initial height is greater than 1, e.g., after a hard fork, the `NewFromGenesisDoc` should set `LastBlockHeight` to `genesis.InitialHeight-1` instead of to `0`, as it currently does (here).
- A node starts from some state
  - If the stored `LastBlockHeight` is greater than the `genesis.InitialHeight`, it should probably just update its state with what's stored. The team should consider if some integrity checks must be performed: if the cause of the fork is to remove a sequencer that is misbehaving, the node may be in a state inconsistent with the initial state of the fork.
  - If the stored `LastBlockHeight` is less than the `genesis.InitialHeight`, then it is probably a node that has not applied all blocks before the fork. In this case, it is probably sensible to initialize the node via genesis.

BlockCache can be filled with blocks from malicious nodes

Title	BlockCache can be filled with blocks from malicious nodes
Project	Celestia Rollkit Phase 1
Type	IMPLEMENTATION
Severity	3 HIGH
Impact	3 HIGH
Exploitability	2 MEDIUM
Status	ACKNOWLEDGED
Issue	https://github.com/rollkit/rollkit/issues/1417

Involved artifacts

- block/manager.go

Description

Blocks are added to the `blockCache` once the SyncLoop processes them, and only removed from the cache if the block is successfully applied to the state. Blocks are cached (line) if they have never been seen before or the node has already executed a block for that height. This check is insufficient to guarantee that the BlockCache isn't filled with blocks from malicious nodes.

Problem Scenarios

Malicious nodes can make honest nodes' `blockCache` grow unboundedly by creating blocks for heights that the sequencer has not yet reached and disseminating them through the p2p layer. The cache may be eventually cleaned when the sequencer reaches those heights, but attackers may exhaust honest nodes memory before.

Recommendation

We recommend validating the authenticity of rollup blocks as early as possible to avoid filling up data structures with invalid blocks, similarly toThe authenticity of blocks received through the p2p layer is not checked.

Large rollup blocks may compromise liveness

Title	Large rollup blocks may compromise liveness
Project	Celestia Rollkit Phase 1
Type	IMPLEMENTATION
Severity	2 MEDIUM
Impact	3 HIGH
Exploitability	1 LOW
Status	ACKNOWLEDGED
Issue	https://github.com/rollkit/rollkit/issues/1428

Involved artifacts

- state/executor.go
- da/da.go

Description

The `SubmitBlocks` function of the da's module computes how many blocks fit within the maximum blob size (`dac.DA.MaxBlobSize`) and submits them to the DA layer. If it succeeds, the function returns the number of blocks that were submitted.

Unfortunately, there is no guarantee that the rollup blocks created by the sequencer have a size `<=dac.DA.MaxBlobSize` . Thus, if the sequencer creates a block with size `>dac.DA.MaxBlobSize` , the block (and any block of greater height) won't be published in the DA layer.

Problem Scenarios

The sequencer will stop publishing rollup blocks in the DA layer as soon as it creates a block of size `> dac.DA.MaxBlobSize` . When the sequencer tries to publish such a block by calling `SubmitBlocks` , the function will fail due to this check. Then the block will never be removed from the `pendingBlocks` . This means that any block created after that one will never be published in the DA layer either, which compromises the liveness of the rollup.

Recommendation

It should be enforced in the code that rollup blocks are `<=dac.DA.MaxBlobSize` : it should be enough to guarantee that `maxBytes<=dac.DA.MaxBlobSize` when the sequencer starts.

Miscellaneous code improvements

Title	Miscellaneous code improvements
Project	Celestia Rollkit Phase 1
Type	IMPLEMENTATION PRACTICE
Severity	0 INFORMATIONAL
Impact	1 LOW
Exploitability	04 UNKNOWN
Status	ACKNOWLEDGED
Issue	

Description

In this issue, we describe several improvements to the code. Those typically do not affect the functionality, but improve the code readability, make code more robust for future changes, or represent a good engineering practice.

1. The DA's `SubmitBlocks` function returns the height at which it publishes the set of rollup blocks in the DA layer (`DAHeight`). This is never used by the caller function (`submitBlocksToDA`). Consider removing it if it is not going to be used.
2. The logic to validate that the trusted sequencer has created a block read from the DA layer is scattered among multiple functions in different modules:
   - function `processNextDaBlock` in the manager checks that the address of the proposer in the signed header matches the sequencer address in the genesis file (here);
   - function `ValidateBasic` in signed_header.go verifies (among other things) the signed header using the public key of the first validator in the validators set of the signed header, and;
   - function `checkCentralizedSequencer` checks (among other things) that the first validator in the validators set of the signed header matches the proposer in the signed header.
   We recommend writing a wrap function that gathers all checks to improve readability. This is especially relevant given that this code may be reused to do the same validation for blocks received via the p2p layer.
3. Nodes call `ProcessProposal` in `ApplyBlock` . `ProcessProposal` is originally used in CometBFT to check the validity of a block proposal given that leaders may be malicious. Given that there is a single trusted sequencer this call may be unnecessary. Consider removing it if it is not planned to be used for any other purpose by rollup apps.

The semantics of submitBlocksToDA is unclear

Title	The semantics of submitBlocksToDA is unclear
Project	Celestia Rollkit Phase 1
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	1 LOW
Exploitability	04 UNKNOWN
Status	ACKNOWLEDGED
Issue	https://github.com/rollkit/rollkit/issues/1429

Involved artifacts

- block/manager.go

Description

The function `submitBlocksToDA` is called periodically by the `BlockSubmissionLoop`. The function attempts to publish rollup blocks in the DA layer. The function first reads from the `pendingBlock` map. It then calls the DA's module `SubmitBlocks` function and returns successfully if all of the blocks read from the `pendingBlock` map have been successfully published in the DA layer.

If some of them were not published, e.g., due to reaching the max blob size, the `submitBlocksToDA` retries up to a maximum of `maxSubmitAttempts` attempts. At the beginning of each attempt, the function reads the `pendingBlock` map again. If no attempt succeeds (fully submits the set of blocks read at the beginning of the attempt), it returns an error that the caller function (`BlockSubmissionLoop`) logs.

The semantics of the function are not well defined: it is unclear what an error means. This is a consequence of the fact that the sequencer may add blocks to the `pendingBlock` map in between attempts. This means that for instance, the function may successfully submit to the DA layer the initial set blocks retrieved from the `pendingBlock` map (first attempt) and still return an error that it is logged in `BlockSubmissionLoop`.

Problem Scenarios

1. Assume that the sequencer calls `submitBlocksToDA` and that in the initial attempt, there are three blocks (`a, b and c)` in the `pendingBlock` map when the function reads it.
2. Assume that only `a and b` are published in the DA layer in the first attempt.

3. Assume that the sequencer now publishes two new blocks (`d and e`) in between attempts: after the `submitBlocksToDA` reads the `pendingBlocks` map in the first attempt and before it reads it again in the second attempt.
4. Assume for simplicity that `maxSubmitAttempts=2` .
5. The function reads (`c, d and e`) in the second attempt.
6. Assume that the sequencer only manages to publish `c` in the DA layer.
7. The function `submitBlocksToDA` is going to return an error even though it has published the three blocks that were originally read in the first attempt.

Recommendation

We recommend to slightly redesigning the function to clarify its semantics. We propose two alternatives:

1. Read `pendingBlock` out of the attempts loop. This way, the function simply tries to publish all the blocks that were in the `pendingBlocks` map when it was called: if more blocks are added afterwards, the function does not try to publish them. This way, if the function returns an error it is because it was not able to publish some of those blocks.
2. Only return an error if no block was published to indicate that there is a connectivity problem with the DA layer. This way the function tries to publish as many blocks as possible and only returns an error if no block was published.

The current implementation assumes that a sequencer operates a single rollup

Title	The current implementation assumes that a sequencer operates a single rollup
Project	Celestia Rollkit Phase 1
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	04 UNKNOWN
Exploitability	04 UNKNOWN
Status	ACKNOWLEDGED
Issue	

Description

The current implementation assumes that a sequencer only creates blocks for a single rollup. If a sequencer creates blocks for multiple rollups that write in the same namespace of the DA layer, nodes won't be able to discard blocks that belong to other rollups, compromising safety.

Problem Scenarios

An example scenario in the current implementation is the following. Assume that a sequencer is shared between two rollups (`a` and `b`) and that blocks of both rollups are written in the same namespace of the DA layer.

- A node retrieves blocks by calling the `RetrieveBlocks` of the da module.
- The function simply retrieves blocks from the configured namespace, without doing any other validation.
- The blocks are handled by the `processNextDABlock` function of the manager, which only checks that the blocks have been created by the sequencer by checking that the signed header proposer is the hardcoded sequencer and by calling `ValidateBasic()` on the block.
- Assume that a node of rollup `a` is at height `h` : it has executed rollup `a` blocks up to height `h` .
- Assume that the sequencer does not create more blocks for rollup `a` for a while.
- Assume that the sequencer creates now a block for rollup `b` for height `h+1` , publishes it in the DA layer, and that the node retrieves it.
- The block will pass the validation and the node will send it to the `blockInCh` channel.
- The `SyncLoop` will eventually read the block and call the `trySyncNextBlock` function. The function does similar checks like before, so the block will pass the validation and the node will execute it, which is a safety violation.

- Note that even if the node receives the canonical block for height `h+1` later, the node will filter it out in the `SyncLoop` due to this line.

Recommendation

Since shared sequencer is in the roadmap, this should be taken into consideration when the implementation is modified to support shared sequencers. One possibility would be to use the chain-id to filter out irrelevant blocks. One must be careful: **chain-id can only be used if it is guaranteed that these are unique among the rollups that share a given sequencer**.

Halt a node if it errors when applying a block from the sequencer

Title	Halt a node if it errors when applying a block from the sequencer
Project	Celestia Rollkit Phase 1
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	04 UNKNOWN
Exploitability	04 UNKNOWN
Status	ACKNOWLEDGED
Issue	https://github.com/rollkit/rollkit/issues/1437

Involved artifacts

- block/manager.go

Description

`ApplyBlock` is called by the sequencer after creating a block and by any node when it is about to execute one. This PR halts the sequencer if `ApplyBlock` returns an error. This is the correct behaviour as the block it is guaranteed to come from the trusted sequencer itself and if `ApplyBlock` returns an error may be an indicator of a bug either in the block creation logic or in the client app. We propose that the same logic should be applied when a non-sequencer node attempts to apply a block created by the trusted sequencer. This is consistent with cometBFT (here).

Problem Scenarios

If a node is unable to execute a canonical block may be an indicator of a bug either in the block creation logic or in the client app - an unrecoverable error. It can also be an indicator of non-determinism, which should be avoided.

Recommendation

We recommend halting a node by panicking when `ApplyBlock` returns an error in `trySyncNextBlock`. ***This logic should only be added after previous issues are fixed*** such that it is guaranteed that a node only attempts to apply a block when it is certain that the block has been created by the trusted sequencer. Note that this is not the case in the current implementation. If this is not guaranteed there is the risk that malicious nodes could halt correct nodes by gossiping non-canonical blocks that make `ApplyBlock` error.

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.

Impact Score	**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:

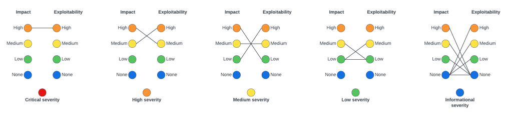
- *Actors* can be any entity that interacts with the system: other blockchains, system users, validators, relayers, but also uncontrollable phenomena (e.g. network delays or partitions).
- *Actions* can be

    - *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)

Exploitability Score	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.



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.

Severity Score	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

Severity Score	Examples
🟢 **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