



# **Security Audit Report**

Celestia 2024 Q2

Authors: Andrija Mitrovic, Ivan Golubovic, Manuel Bravo

Last revised 1 July, 2024

# Table of Contents

**Audit Overview ..... 1**

Scope 1

Conclusion 1

**Audit Dashboard ..... 3**

xTarget Summary 3

Engagement Summary 3

Severity Summary 3

**Threat analysis ..... 4**

Upgrade system 4

minfee 6

prepare/process proposal 6

**Findings ..... 9**

Unfair voting result due to non unique version identifier 10

A malicious user could exploit upgrade mechanism to crash nodes 11

Signal is used for quorum counting and readiness which can lead to validators using vulnerable code 12

Delayed signal leads to incorrect quorum counting 14

Proposed block could exceed size limit 16

**Appendix: Vulnerability Classification ..... 18**

Impact Score 18

Exploitability Score 18

Severity Score 19

**Disclaimer ..... 21**

# Audit Overview

## Scope

In May 2024, [Informal Systems](#) has conducted a security audit for Celestia's custom upgrade mechanism (signal module), minimum fee check and diff to Prepare and Process Proposals as well as integration of ICA and PFM modules, and removal of blobstream module . The audit aimed at inspecting the correctness and security properties of the before mentioned components.

The audit was performed from May 20th, 2024 to June 14th, 2024 by the following personnel:

- Andrija Mitrovic
- Ivan Golubovic
- Manuel Bravo

## Relevant Code Commits

The audited code was from:

- celestia-app repository at commit hash c5b0b9421e8bbe04d26104716bb86750a6891ab0

## Conclusion

Overall assessment of audited codebase indicates high level of quality. The Celestia team made great effort in handpicking the right parts of their implementation to be audited together. This lead to code being well contained, without too much overhead in auditing process. The team also made sure we had all our questions clarified in the meetings and our communication over slack.

After addressing the identified issues, some changes were made to the design, specifically addressing two highly ranked findings that were resolved by the end of the audit. The biggest changes include:

- The validators can now signal for version that is more than 1 greater from the current one,
- Version voted for called quorumVersion is now persisted in the multistore
- There is a delay added before changing the version

Despite these updates, the audit report still only references the initial commit hash audited, without incorporating the subsequent design modifications and bug fixes.

## Modules integration

Our analysis focused on the integration of two modules : Packet Forward Middleware and Interchain Accounts. We leveraged the existing integration of these modules in the Cosmos Hub (Gaia) as a benchmark for comparison. Our investigation revealed a high degree of similarity between the implementations. Both projects achieve the same functionality, with the primary difference lying in code organization. In Celestia, the integration is centralized within the `app/app.go` file, while Gaia distributes the code across various files like `keepers.go` and `modules.go` .

These separate files are then called upon within `app.go` to achieve the desired functionality. This difference is primarily an organizational one and does not impact the core functionality of the modules.

The removal of the blobstream module leverages the new module versioning that ensures that the blobstream module will be used in the first version but not after it. In addition the hooks

`AfterValidatorBeginUnbonding` and `AfterValidatorCreated` are no-ops if the app version is greater than 1 because blobstream is disabled from v2.

## Upgrade system, minimum fee check and prepare/process proposal diff

This part of audit was executed in structured and already established manner in our auditing process where we first created a [threat model](#), inspected threat by threat and concluded with the findings. The final number of findings is 2 of high severity, 2 low and 1 informational severity. The findings were thoroughly discussed with Celestia team and are all related to new upgrade process and its design mostly. This upgrade process was also in some way compared to the already existing upgrade process in Cosmos, but with Celestia's goal to improve it. We found the code to be easily readable, well maintained and well structured. The design decisions and possible improvements are described in detail in our [Findings](#) section.

# Audit Dashboard

## xTarget Summary

- **Type:** Protocol and Implementation
- **Platform:** Go
- **Artifacts:**
  - New upgrade system: [x/signal](#), [celstia-app/app](#)
  - Minimum fee checker: [fee\\_checker.go](#), [ante.go](#)
  - Prepare and Process Proposal diff: [prepare\\_proposal](#), [process\\_proposal](#)

## Engagement Summary

- **Dates:** 20.05.2024. - 14.06.2024.
- **Method:** Manual code review, protocol analysis

## Severity Summary

Finding Severity	#
Critical	0
High	2
Medium	0
Low	2
Informational	1
Total	5

## Threat analysis

The threat analysis is done by defining threats. Threats are divided in the three groups: upgrade system threats, min fee threats and prepare/process proposal threats. Each threat is accompanied with a consequence if the threat is feasible and a conclusion if the implementation mitigates or not the mentioned threat.

We have inspected the listed threats, resulting in the findings presented in the [Findings](#) section.

### Upgrade system

#### 1. The app version is updated to a version that has not formed a valid quorum

**Consequences:** Validators upgrade to a non-valid version.

**Conclusion:** The threat does not hold. The app versions are represented by integer numbers and during an update only one version number is reserved for the new binary. This approach can not differentiate versions ids of two different binaries that are suggested in one update, for example the suggested binary is changed by developers during the update because some critical fix.

**Linked Findings:**

- [Unfair voting result due to non unique version identifier](#)
- [Signal is used for quorum counting and readiness which can lead to validators using vulnerable code](#)

#### 2. Given an upgrade, two validators apply state or store migrations at different heights

**Consequences:** Due to the state migrations, each validator may have a different state at a given height.

**Conclusion:** The threat does not hold. The upgrade happens when a `TryUpdate` message is processed. This message is processed only if a majority of validators voted on a new version. If a quorum on a new version is reached the app version is changed to the new one in the end block of a block where the `TryUpdate` message is processed. On that block commit the state and store migration are done. Thus migrations can not happen in different blocks for different validators.

#### 3. The app version can decrease

**Consequences:** This is unexpected; binaries won't be prepared for it causing nodes to panic.

**Conclusion:** The threat does not hold. The app version can not decrease because validators through `SignalVersion` can only vote for the current or a version that is by one larger than the current. This is enforced [here](#).

#### 4. The app version can increase by more than one at once: a node can skip a version

**Consequences:** This is unexpected; binaries won't be prepared for it causing nodes to panic.

**Conclusion:** The threat does not hold. A validator can not even suggest a version that is greater more more than one than the current one. This check of `SignalVersion` message is done [here](#).

5. The signal module does not count votes correctly: given a height for which `5/6` of the voting power has signaled an upgrade, the signal module does not form a quorum

**Consequences:** Liveness is compromised - if `5/6` of the voting power at a given height has signaled an upgrade, then the signal module should form a quorum if a `TryUpgrade` message is called at that height and update the `quorumVersion` accordingly.

**Conclusion:** The signal module tracks validator support for version upgrades. It uses a map called `versionToPower` to accumulate signaling power for each version. If more than 5/6 of validators signal for version upgrade, the version is marked as `quorumVersion`. This is later used in `EndBlock` to conclude if it is time for an upgrade.

A low severity issue in quorum counting was discovered and described in the following finding:  
[Delayed signal leads to incorrect quorum counting](#)

6. The app version is not updated at the end of a block whose execution results in updating `quorumVersion`.

**Consequences:** Liveness is compromised - if `quorumVersion` is updated during the execution of a block, then the app version must be updated at the end of the block.

**Conclusion:** The threat does not hold. The main focus of the threat was to inspect if it's possible to have a quorum that is not recorded properly. Upon calling `tryUpgrade` the signals are counted and the result is recorded right away into `quorumVersion` field of the keeper. There has not been found a way to miss or incorrectly record this.

A slightly related finding to this threat has been discovered:  
[A malicious user could exploit upgrade mechanism to crash nodes](#)

7. An upgrade is not executed or fails during the commit of a block whose execution resulted in updating the app version

**Consequences:** Liveness is compromised - if the app version is updated at the end of a block, then the upgrade must be completed successfully when the block is committed.

**Conclusion:** The threat does not hold. The upgrade at the end of block depends on `ShouldUpgrade` function directly checking the quorum version. If the `quorumVersion` is saved properly, the code running the upgrade will be triggered.

The code of interest is also in `EndBlocker` where `ShouldUpgrade` is used to control if it's time for an upgrade. The upgrade is executed in the following lines:

```
app.SetAppVersion(ctx, newVersion)
app.SignalKeeper.ResetTally(ctx)
```

Later, in Celestia's cosmos-sdk fork a call to `migrateCommitStore` and `migrateModules` is executed. `MigrateCommitStore` is migrating stores based on the store keys of old and new version.

`MigrateModules` calls `RunMigrations` which is utilized to handle modules migration from old to new version.

## 8. Users can execute messages of a given app version that are not running

**Consequences:** The application has not been upgraded yet. Thus, state migration has not happened. Executing such a message may crash the app.

**Conclusion:** The threat does not hold. The code regarding filtering allowed messages is in [msg\\_gatekeeper.go](#). The messages allowed for a certain version are stored in a [map](#), matching the version and allowed messages. This should prevent any message not supported in current version to be executed.

## 9. The ante handler prevents the execution of some messages of the current app version

**Consequences:** Liveness is compromised - the app must be able to execute all messages for a given app version.

**Conclusion:** The threat does not hold. The same code from the previous threat should apply for this threat as well. The mapping of messages and versions should also prevent that some message is prevented in its execution even though it is allowed. This map is populated directly from the configurator on initializing the app.

## minfee

### 1. The fee checker ante handler permits transactions that do not adhere to the minimum fee to be executed

**Consequences:** Users can execute transactions without reaching the fee minimum threshold.

**Conclusion:** The threat does not hold. This is prohibited through [DeductFeeDecorator](#) [AnteHandler](#) .

[AnteHandler](#) rejects transactions which fees does not meet a minimum threshold for the node as well as a global minimum threshold. The global minimum fee threshold check is only done if the app version is greater than one.

### 2. The fee checker ante handler prevents transactions that adhere to the minimum fee from being executed

**Consequences:** Liveness is compromised - Users cannot execute transactions that adhere to the minimum fee threshold.

**Inspection:** The threat does not hold. Only transactions that do not adhere to the minimum fee are prevented. This is done by returning error in these situations. No errors are returned if transactions adhere to the minimum.

### 3. The minimum fee threshold cannot be updated via governance proposal

**Consequences:** Chain developers cannot adjust the minimum fee threshold to control load.

**Conclusion:** The minimum fee threshold is a part of parameters. It can be updated through submit a param change proposal using the sdk [NewProposal](#).

## prepare/process proposal

### 1. The set of transactions [prepareProposal](#) response exceeds the maximum amount of bytes

**Consequences:**



- It violates Requirement 2 [ `PrepareProposal` , tx-size] of the ABCI specification: when p's Application calls `ResponsePrepareProposal` , the total size in bytes of the transactions returned does not exceed `RequestPrepareProposal.max_tx_bytes` .
- It compromises liveness as the consensus engine will reject proposals from honest validators that exceed the maximum allowed.

**Conclusion:** The threat does not hold. The transactions are fetched from the mempool in consensus already taking into account the size limit, so the `PrepareProposal` is operating only with these transactions and could not exceed the limit.

During inspection, an informational bug was discovered and described in the following finding:  
[Proposed block could exceed size limit](#)

## 2. Honest validators reject proposals from honest proposers

### Consequences:

- It violates Requirement 3 [ `PrepareProposal` , `ProcessProposal` , coherence] of the ABCI specification: for any two correct processes p and q, if q's CometBFT calls `RequestProcessProposal` on up, q's Application returns `Accept` in `ResponseProcessProposal` .
- It compromises liveness as validators will reject proposals coming from honest proposers.

**Conclusion:** The threat does not hold. The proposer filters the transactions to be included in the block and is building the data square in `prepareProposal` . In `processProposal` proposed transactions are divided into blob and non-blob ones with stateless checks around them. Then, those transactions are used to form the data square and calculate the data availability header which is checked against the proposed data availability header. No scenario in which honest validators reject honest proposal has been discovered.

## 3. `processProposal` is non-deterministic, i.e., its result does not depend exclusively on the proposal and the committed blockchain state

### Consequences:

- It violates Requirement 4 [ `ProcessProposal` , determinism-1]: of the ABCI specification: `ProcessProposal` is a (deterministic) function of the current state and the block that is about to be applied. In other words, for any correct process p, and any arbitrary block u, if p's CometBFT calls `RequestProcessProposal` on u at height h, then p's Application's acceptance or rejection exclusively depends on u and  $sp, h-1$ .
- It compromises liveness as validators will reject proposals coming from honest proposers.

**Conclusion:** The threat does not hold. The changes in the scope of the audit did not affect deterministic set of rules used to build data square by the proposer in `prepareProposal` and the validators in `processProposal` .

## 4. `prepareProposal` and/or `processProposal` modify the blockchain state

### Consequences:

- It violates Requirement 9 [all, no-side-effects]: of the ABCI specification: p's calls to `RequestPrepareProposal` , `RequestProcessProposal` at height h do not modify  $sp, h-1$ .

- It compromises safety, as at `prepareProposal` and `processProposal`, the proposal has not yet been decided, so it may need to be rolled back.

**Conclusion:** The system threat does not hold. There has not been found that any part of the implementation in `prepareProposal` or `processProposal` would modify the blockchain state. The changes introduced between the last and this audit do not affect the corresponding part of code.

## Findings

Title	Type	Severity	Status
Unfair voting result due to non unique version identifier	IMPLEMENTATION	HIGH	RESOLVED
A malicious user could exploit upgrade mechanism to crash nodes	IMPLEMENTATION	HIGH	RESOLVED
Signal is used for quorum counting and readiness which can lead to validators using vulnerable code	IMPLEMENTATION	LOW	ACKNOWLEDGED
Delayed signal leads to incorrect quorum counting	IMPLEMENTATION	LOW	ACKNOWLEDGED
Proposed block could exceed size limit	IMPLEMENTATION	INFORMATIONAL	ACKNOWLEDGED

## Unfair voting result due to non unique version identifier

Project	Celestia 2024 Q2
Type	IMPLEMENTATION
Severity	3 HIGH
Impact	3 HIGH
Exploitability	2 MEDIUM
Status	RESOLVED
Issue	<a href="https://github.com/celestiaorg/celestia-app/issues/3550">https://github.com/celestiaorg/celestia-app/issues/3550</a>

### Involved artifacts

- [x/signal/keeper.go](https://x.com/signal/keeper.go)

### Description

The app version is presented with an integer number. The upgrade process relies only on an version (integer) number (e.g., version 1, version 2) to identify the new version being proposed to validators. Such version identifier lacks uniqueness in representing two different app binaries suggested as updates for the next version.

### Problem Scenarios

1. A new binary is shared with the validators for voting on a new upgrade (version 2).
2. Validators initiate voting based on version 2.
3. Developers discover a critical issue in the proposed version 2 code.
4. Due to the limited version identification system, developers are forced to issue a new binary (fixed version 2) but maintain the same version number.
5. The new version 2 already has votes in for it (or votes against it) based on the previous voting procedure since there is no unique way to distinguish the new version 2 code for an upgrade. This breaks the voting process and could lead to a version being rejected or accepted despite the majority of votes.

### Recommendation

Implement a unique version identifier that incorporates not only a version number but also a unique identifier for each specific binary version. This would allow developers to address critical bugs in proposed versions without causing confusion among validators who have already participated in the initial voting process and prevent unfair voting result.

Another possible solution, which was adopted by Celestia team, is to remove the constraint that allows validators to signal only for the version that is equal to `currentVersion+1`. This would allow skipping the versions that are discovered to be unsafe or broken in any way.

## A malicious user could exploit upgrade mechanism to crash nodes

Project	Celestia 2024 Q2
Type	IMPLEMENTATION
Severity	3 HIGH
Impact	3 HIGH
Exploitability	2 MEDIUM
Status	RESOLVED
Issue	<a href="https://github.com/celestiaorg/celestia-app/issues/3552">https://github.com/celestiaorg/celestia-app/issues/3552</a>

### Involved artifacts

- [x/signal/keeper.go](https://x.com/signal/keeper.go)

### Description

The current upgrade process lacks a predefined activation height for new versions. Instead, the system relies on achieving a specific voting threshold (5/6) and subsequent manual execution of the `tryUpgrade` message to initiate the upgrade. This approach introduces a vulnerability to potential attacks.

### Problem Scenarios

1. A malicious user monitors the upgrade voting process.
2. Once the voting signals reach the required threshold (e.g., or even slightly earlier - close to 2/3 acceptance), the attacker submits a `tryUpgrade` message before validators have sufficient time to prepare for the switch.
3. The network, upon receiving the `tryUpgrade` message and verifying sufficient voting acceptance, activates the new version.
4. Due to the lack of predefined activation height, validators may not be fully prepared for the upgrade. This can lead to crashes or unexpected behavior on unprepared validator nodes.

### Recommendation

To mitigate this attack vector, the upgrade process should incorporate a delay before the proposed version is enforced across all the validators. This mechanism ensures validators have a time window to prepare for the switch before the upgrade is triggered.

Signal is used for quorum counting and readiness which can lead to validators using vulnerable code

Project	Celestia 2024 Q2
Type	IMPLEMENTATION
Severity	1 LOW
Impact	1 LOW
Exploitability	1 LOW
Status	REPORTED
Issue	

## Involved artifacts

- [x/signal/keeper.go](https://x/signal/keeper.go)

## Description

The current upgrade process relies on a single-shot "signal" mechanism that serves two distinct purposes - indicating a vote for the upgrade and signifying the validator's readiness to adopt the new version.

## Problem Scenarios

Imagine a scenario where developers propose an upgrade with a new binary. Due to the combined voting and readiness nature of the signal, validators may switch to the new binary before the upgrade proposal's result is final. This premature switch exposes them to potential risks:

- **Unnecessary binary swapping:** If the upgrade proposal is rejected, validators who already switched binaries may have to revert to the previous binary. This switching can be disruptive and resource-intensive.
- **Exposure to vulnerable code:** Under the assumption that some validators check the binary of a given upgrade before voting in favor of it, it could be argued that the more votes a given upgrade gets, the less likely is for the binary to contain critical security issues. Thus, the fact that validators switch to the new binary before a quorum is reached, it may increase the likelihood of running code containing major security vulnerabilities.

## Impact and Exploitability

Our impact and exploitability analysis only considers the first scenario. We leave the second scenario out as currently, we do not expect validators to do a thorough vulnerability check before voting for an upgrade, which would invalidate the assumption upon which the problem scenario builds. Note that if the assumption holds in the future, the second scenario would be of high relevance and be a major security vulnerability with a high impact.

The exploitability of the aforementioned scenario is considered low because we expect upgrade rejections to be rare. We consider the impact to be low as well. Despite being unoptimal, doing some extra binary swaps is nothing critical.

## Recommendation

To mitigate this risk, the upgrade process could be redesigned to separate the voting and readiness signals. Validators should first cast their vote (signal) on the proposed upgrade. After reaching the quorum for the proposed version, validators would express their readiness with another round of voting. Note that this two-round voting mechanism does not eliminate the risk completely: there is no guarantee that once a quorum for acceptance is reached, the readiness quorum will be reached as well, e.g., if some validators change their mind. Nevertheless, one could consider these cases rare and that in the normal case, if a quorum for acceptance is formed, it is very likely that it will be for readiness, which would mitigate the risk.

## Delayed signal leads to incorrect quorum counting

Project	Celestia 2024 Q2
Type	IMPLEMENTATION
Severity	1 LOW
Impact	2 MEDIUM
Exploitability	1 LOW
Status	ACKNOWLEDGED
Issue	

### Involved artifacts

- [x/signal/keeper.go](https://x.com/signal/keeper.go)

### Description

The upgrading process requires a quorum of validators to support an upgrade before upgrading the chain. This is managed via a signaling system. A validator can support the proposed upgrade by signaling the proposed chain version (+1 of the current one). When the validator does not support it, it can signal the current version. If a quorum is reached for the proposed version, a user (typically chain developers) will submit a `TryUpgrade` message, which will trigger the upgrade.

Due to asynchrony and the fact that the upgrading system only waits for a quorum, some signals for a given upgrade may be received after the `TryUpgrade` is processed. This is a problem: the delayed signals could count as a rejection vote for the next upgrade. Although validators may be able to rewrite their signal by submitting another one, there may be a period of time in which users monitoring the signaling believe that a proposed upgrade has been rejected by the validators. How bad this is will depend on the actions taken by the users monitoring the signaling.

### Problem Scenarios

Assume that the total number of validators is 4 and that a quorum requires 3.

1. Developers propose an upgrade to version 3.
2. All validators signal the upgrade.
3. Chain developers submit `TryUpgrade` as soon as a quorum is reached: when 3 signals have been processed.
4. Validators upgrade after processing the `TryUpgrade`.
5. The signal for the last validator Bob is now processed and recorded.
6. Developers proposed an upgrade to version 4.
7. Assume that one validator Alice wants to reject the proposal and the other three want to support it (including Bob).



8. Assume that Alice signal is processed before than any other signal.
9. From this point in time and until Bob's signal is processed, any user monitoring the system would believe that the upgrade has been rejected because it should be impossible to form a quorum.

## Recommendation

We recommend counting votes properly by allowing users to explicitly vote `accept` or `reject` to a given upgrade, such that if votes are processed after an upgrade occurs, the votes are not counted for the next upgrade.

## Proposed block could exceed size limit

Project	Celestia 2024 Q2
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Status	ACKNOWLEDGED
Issue	

### Involved artifacts

- [prepare\\_proposal.go](#)

### Description

Before the `PrepareProposal` is run, a consensus parameter `maxBytes` is used to fetch as many transactions as possible from `mempool` which are to be sent in `RequestPrepareProposal`. Along with the transactions, `blockDataSize` is also included in the request, and it has the value of previously mentioned `maxBytes`. Since the `PrepareProposal` is not adding more transactions to the block, it is not expected that the block exceeds `blockDataSize`. However, `squareSize` and `hash` are added to the block and this could increase block size over limit.

The celestia-core consensus actually `panics` on oversized block data, but currently it takes into account only the transactions from the block, and not the added square size and hash fields. We are pointing out this as informational due to critical nature of a panic inside consensus and potential changes in the code that could overlook this.

### Problem Scenarios

During the retrieval of transactions from the mempool, the `maxBytes` parameter is used to ensure the total size of fetched transactions does not exceed the limit. Consider a scenario where the total size of the fetched transactions is exactly equal to `maxBytes`. Initially, this does not breach the size limit. However, after the inclusion of `squareSize` and `hash` in the `PrepareProposal` phase, the block size exceeds the proposed limit.

It is only due to the fact that consensus counts only the `size of transactions` that the panic is not triggered.

## Recommendation

The Celestia team acknowledged this as an informational issue and the resolution is going to be a test added to cover the corner case of fetched transactions from the mempool equaling maxBytes in size.





## 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
 <b>High</b>	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
 <b>Medium</b>	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)
 <b>Low</b>	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)
 <b>None</b>	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:

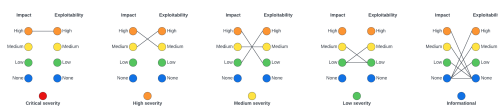
- *Actors* can be any entity that interacts with the system: other blockchains, system users, validators, relayers, but also uncontrollable phenomena (e.g. network delays or partitions).
- *Actions* can be

- *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
<span style="color: orange;">●</span> <b>High</b>	illegitimate actions taken by a small group of actors; possibly coordinated with legitimate actions taken by a medium group of actors
<span style="color: gold;">●</span> <b>Medium</b>	illegitimate actions taken by a medium group of actors; possibly coordinated with legitimate actions taken by a large group of actors
<span style="color: green;">●</span> <b>Low</b>	illegitimate actions taken by a large group of actors; possibly coordinated with legitimate actions taken by all actors
<span style="color: blue;">●</span> <b>None</b>	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
<span style="color: red;">●</span> <b>Critical</b>	Halting of chain via a submission of a specially crafted transaction
<span style="color: orange;">●</span> <b>High</b>	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
<span style="color: gold;">●</span> <b>Medium</b>	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
 <b>Low</b>	2x increase in node computational requirements via coordinated withdrawal of all user tokens
 <b>Informational</b>	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.