



Security Audit Report

Celestia 2024 Q1: Blobstream

Authors: Josef Widder, Andrija Mitrovic

Last revised 15 February, 2024

Table of Contents

Audit overview.....	1
The Project	1
Conclusions	1
Audit dashboard.....	2
Target Summary	2
Engagement Summary	2
Severity Summary	2
Overview.....	3
Verification logic comparison between Tendermint light client and TendermintX.	4
VerifyNonAdjacent and skip comparison	4
VerifyAdjacent and step comparison	8
Findings.....	10
Threshold check uses greater or equal instead of only greater	11
Missing checks in commit verification	13
Skip function does not check if the trusted and untrusted header are non adjacent	15
Function parameter naming is confusing	17
Minor code readability issue	18
Disclaimer.....	20
Appendix: Vulnerability Classification.....	21
Impact Score	21
Exploitability Score	21
Severity Score	22

Audit overview

The Project

In January 2023, [Informal Systems](#) has conducted a security audit for Celestia of the light client verification functions in TendermintX.

The main focus of the audit was the confirmation that all the necessary verification steps are done to include the incoming untrusted header as a trusted one. This was done through comparison of the audited functions with the standard Golang implementation of the Tendermint Light Client verification.

The audited commit hash is [477c704](#).

The audit took place from January 18, 2024 through February 2, 2024 by the following personnel:

- Josef Widder
- Andrija Mitrovic

Conclusions

In general, we found the codebase to be of very high quality: the code is well structured and easy to follow. In the audit, we found 5 issues: 2 critical, 1 low and 2 informational in severity. The issues have been resolved. Besides this, the Tendermint Light Client verification has some sanity checks that are not present in the TendermintX implementation. These are pointed out in the report as a to do.

Audit dashboard

Target Summary

- **Type:** Implementation
- **Platform:** Rust and Solidity
- **Artifacts:**
 - [tendermintx](#)

Engagement Summary

- **Dates:** 18.02.2024 to 02.02.2024
- **Method:** Manual code review and comparison against standard Golang implementation
- **Employees Engaged:** 2

Severity Summary

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

Overview

The code under review implements the verification of Tendermint headers based on the Tendermint light client (skipping verification) and sequential verification (step). More concretely, only the latest header is used to validate a requested header of greater height. If all the verification steps pass, the newly requested header replaces the previously header stored in the contract.

In comparison to full Tendermint Light Clients (as in IBC), at the moment **TendermintX has limited security-related functionality:**

- only the latest header is currently used to verify new headers,
- the contract currently does not have logic to freeze the light client in case of a light client attack, i.e., two different headers for the same height that both pass verification (but such a logic can be added to the contract in the future),
- there is no check for the trusting period that is linked to security provided by proof-of-stake and the unbonding period. There is a check that makes sure that the new headers height is not much bigger than the previous one. However, this does not prevent that the smart contract is not called for, say three weeks, and therefore the old header is not trustworthy according to the Tendermint security model.

As a result, less security measures are in place compared to standard light clients, e.g., in the standard IBC implementation. **This audit's results must be understood with respect to the limited functionality.**

We don't see a fundamental problem that this cannot be addressed in the future, in particular, as the verification task that actually is implemented is the most complex logic, while the mentioned points above are just bookkeeping logic, which can be easily added in the future.

The flow of adding a new header is as follows:

- Initially one sets up a *TendermintX circuit* for a specific blockchain, and an initial header that is trusted (subjective initialization).
- A user provides a Tendermint RPC endpoint matching the blockchain specified in the circuit.
- A user (or another contract) that wants to add a new header requests that a header of a specific height is added
- Via RPC calls, the witness fetching logic obtains the header data from the specified blockchain and then supplies the witness data that is verified in the circuit.
- **Only the verification logic implemented in this circuit is the scope of this audit.**

Our approach to audit the verification logic was to compare it to the industry quasi-standard golang implementation in CometBFT, and check whether all checks are in place and implemented correctly.

Verification logic comparison between Tendermint light client and TendermintX

In the course of this audit, we utilized Tendermint's light client checks as a reference for comparison with TendermintX checks. Both implementations feature two distinct functions dedicated to these checks:

- The first one for checking incoming block headers that are non adjacent (that is, if the height of the locally stored trusted header is h , the height of the to-be-verified header is greater than $h+1$):
 - In Tendermint, the equivalent function is named `VerifyNonAdjacent`.
 - In TendermintX, the equivalent function is named `skip`.
- The second one for checking incoming block headers that are adjacent (that is, if the height of the locally stored trusted header is h , the height of the to-be-verified header is equal to $h+1$):
 - In Tendermint, the equivalent function is named `VerifyAdjacent`.
 - In TendermintX, the equivalent function is named `step`.

VerifyNonAdjacent and skip comparison

Both implementations can divide the verification steps into following groups:

1. Sanity checks on untrusted header data structures and validity conditions.
2. Verification of the connection between the trusted and the incoming untrusted header - height comparison and cross validator signature checks

First group - Sanity checks on header data structures and validity conditions

Upon scrutinizing the implementation of these checks in TendermintX, we observed a notable distinction in the approach to the first group of checks compared to the implementation in Tendermint. In the Tendermint light client implementation, check validity is primarily ensured through `ValidateBasic` functions and sanity checks. However, in the TendermintX implementation, these checks are predominantly conducted using Merkle tree inclusion proofs for specific data within the header. Leveraging these proofs and the provided data, the Merkle root is reconstructed and subsequently compared to the expected/given Merkle root.

	VerifyNonAdjacent (Golang reference implementation)	Skip (audited function)	Comment
Chain id length	Check chain id length.	Verify the chain ID against the header.	OK
Is header missing.	Missing header check.	-	OK, because most of the checks with merkle inclusion proofs are compared against the header, if it is missing these would fail.

Is commit missing.	Missing commit check.	-	This is implicitly checked with check if the >2/3 validators signed a commit message. If this check goes through it implies a non-empty commit.
Commit can not be for a missing block.	Commit cannot be for nil block.	-	This is implicitly checked with check if the >2/3 validators signed a commit message. If this check goes through it implies a block is present.
Validation of validators hash	Validate Validators Hash	Verify the validators hash against the computed hash using merkle tree.	OK
Signature and commit checks	Signatures presence and commit check.	Verify the signatures of the validators that signed the header.	OK
Check if the chain id is the expected one.	Header chain id check.	Verify the chain ID against the header.	OK
New header validators match the given validators	New header validators match the given validators.	<ul style="list-style-type: none"> • Computed validator hash matches the expected validator hash. • Check the validators hash came from this header. 	OK
Height of the untrusted commit is non negative	Negative height check.	The following two checks from solidity part: <code>check1</code> and <code>check2</code> , in combination with: merkle tree inclusion check	OK

Height of the untrusted header is non negative	Negative height check.	The following two checks in combination: check1 and check2 , in combination with: merkle tree inclusion check	OK
Round is non negative	Negative round check.	-	Issue: Missing checks in commit verification
Header and commit height must match	Header and commit height mismatch.	-	
Commit and header hash check	Check commit signs block and that header is the same block.	-	

The verification check, which entails confirming whether the sum of the voting power of validators that voted for the untrusted header is greater than 2/3 of the total voting power of the complete validator set, is consistently executed in both the Golang reference implementation and the audited function. This is achieved by summing the power of the validators that voted and subsequently comparing it to the specified threshold. An issue in the used operator has been discovered and it is given in the following table:

	VerifyNonAdjacent (Golang reference implementation)	Skip (audited function)	Comment
Ensure that +2/3 of new validators signed correctly	2/3 signed correctly.	2/3 signed correctly.	Issue: Threshold check uses greater or equal instead of only greater

Second group - Light client skipping verification - >1/3 signatures and trusted and untrusted header relations

In the context of the second group of checks, specifically pertaining to the verification of light client skipping (>1/3 signatures) and the relationship between trusted and untrusted headers, the objective is to confirm the connection between these headers. This verification process involves ensuring that validators from the untrusted header are not only present on the trusted header but also constitute a minimum of 1/3 of the total voting power on the untrusted block.

	VerifyNonAdjacent	Skip	Comment
Ensure that +1/3 of trusted validators signed untrusted header	Verify light trusting.	Verify the threshold.	Issue: Threshold check uses greater or equal instead of only greater

Check if headers are non adjacent	Non adjacent in height.	Range check.	Issue: Skip function does not check if the trusted and untrusted header are non adjacent
Untrusted header height must be greater then the trusted height	Untrusted header height must be greater than the old header height.	Range check and merkle tree inclusion check	OK.

Time-related checks

As mentioned in the [Overview](#) time related checks are missing. These are listed in the following table.

	VerifyNonAdjacent	Skip	Comment
Trusted header expiration	Trusting period check.	-	No time related checks in tendermintX.
Untrusted header time must be greater then the trusted time	Untrusted header time to be after old header time.	-	No time related checks in tendermintX.
Untrusted header time must not be in the future	Untrusted header time not in the future.	-	No time related checks in tendermintX.

Missing checks (do not introduce security issues at the moment)

The following table outlines checks that are currently identified as missing but do not pose immediate security issues. Nevertheless, we recommend explicitly incorporating these checks into the code as a defensive programming measure. The majority of these checks involve hash validations for hashes within the untrusted header:

	VerifyNonAdjacent	Skip	Comment
BlockId hash validation	Validate block id hash of the untrusted header.	-	We would recommend to include this check as a defensive programming.
LastCommit hash hash validation	Validate last commit hash of the untrusted header.	-	We would recommend to include this check as a defensive programming.
Data hash validation	Validate data hash of the untrusted header.	-	We would recommend to include this check as a defensive programming.

Evidence hash validation	Validate evidence hash of the untrusted header.	-	We would recommend to include this check as a defensive programming.
Next Validators hash validation	Validate next validators hash of the untrusted header.	-	We would recommend to include this check as a defensive programming.
Consensus hash validation	Validate consensus hash of the untrusted header.	-	We would recommend to include this check as a defensive programming.
Last Result hash validation	Validate last result hash of the untrusted header.	-	We would recommend to include this check as a defensive programming.
PartSetHeader basic validation	ValidateBasic of PartSetHeader.	-	We would recommend to include this check as a defensive programming.
Block protocol check	block protocol	-	This is introduced into the documentation through https://github.com/succinctlabs/tendermintx/pull/54/commits/8aea1852add5995c466be537d45cb4f8988799e7 .
Proposer Address Length	Check Address Length	-	We would recommend to include this check as a defensive programming.

VerifyAdjacent and step comparison

As with the previous two functions, the validations can be categorized into the same two groups. While most of the checks are identical, this section will spotlight the checks that differ from those mentioned in the preceding section.

Second group - Light client step verification - > the trusted next set of validators is the untrusted header validator set and trusted and untrusted header relations

The only checks that differ from those for a non-adjacent header are the ones from the second group. This is reasonable because headers are one after another, allowing for a more precise connection verification through the checks listed in the following table:

	VerifyAdjacent	Step	Comment
--	-----------------------	-------------	----------------

Check the trusted next set of validators is the untrusted header validator set	Check the validator hashes are the same.	Check the trusted next validator set against the trusted header and the new validators hash matches the next validators' hash of the trusted header.	OK.
Check if headers are adjacent	Adjacent in height.	Adjacent in height.	OK.
Verify trusted header in the untrusted header	https://github.com/cometbft/cometbft/issues/2252	Verify the previous header hash in the new header matches the previous header.	For this check there is an issue reported on the Cometbft. This is not a security issue but good to have. We recommend to follow the resolution of the issue and incorporate some parts if necessary.

Findings

Title	Type	Severity	Impact	Exploitability	Status	Issue
Threshold check uses greater or equal instead of only greater	IMPLEMENTATION	4 CRITICAL	3 HIGH	3 HIGH	RESOLVED	
Missing checks in commit verification	IMPLEMENTATION	4 CRITICAL	3 HIGH	3 HIGH	RESOLVED	
Skip function does not check if the trusted and untrusted header are non adjacent	IMPLEMENTATION	1 LOW	2 MEDIUM	1 LOW	RESOLVED	
Function parameter naming is confusing	PRACTICE	0 INFORMATIONAL	0 NONE	0 NONE	RESOLVED	
Minor code readability issue	PRACTICE	0 INFORMATIONAL	0 NONE	0 NONE	RESOLVED	

Threshold check uses greater or equal instead of only greater

Title	Threshold check uses greater or equal instead of only greater
Project	Celestia 2024 Q1: Blobstream Home
Type	IMPLEMENTATION
Severity	4 CRITICAL
Impact	3 HIGH
Exploitability	3 HIGH
Status	RESOLVED
Issue	

Involved artifacts

- [cometbft/types/validation.go](https://github.com/cometbft/types/validation.go)
- [cometbft/light/verifier.go](https://github.com/cometbft/light/verifier.go)
- [tendermintx/circuits/builder/verify.rs](https://github.com/tendermintx/circuits/builder/verify.rs)
- [tendermintx/circuits/builder/voting.rs](https://github.com/tendermintx/circuits/builder/voting.rs)

Description

Tendermint Light Client and TendermintX incoming header verification have two important threshold checks:

1. Check that +2/3 of new validators signed correctly the untrusted header ([tendermint light client](#) and [tendermintx](#))
2. Check that +1/3 of trusted validators signed untrusted header ([tendermint light client](#) and [tendermintx](#))

Tendermint Light Client uses a function named `verifyCommitSingle` to do these checks, while TendermintX implementation uses the function named `is_voting_power_greater_than_threshold`. While `verifyCommitSingle` uses 'greater than' (>) when checking the threshold, the `is_voting_power_greater_than_threshold` uses 'greater or equal' (>=).

Problem Scenarios

Tendermint Consensus works under the assumption that at most 1/3 of the voting power belongs to faulty (byzantine) validators.

- The +1/3 check means that at least one correct validator signed a header. If the check is just for 1/3 all present signatures could be from faulty validators. Thus the data they signed cannot be trusted.
- The +2/3 check is related to finalizing (deciding) a block in Tendermint consensus. A block is finalized if there are +2/3 precommits for the same block, height, round. If there would be only 2/3 there are scenarios

where validators changed their minds during a consensus instance, and later they decided differently based on +2/3 precommit messages.

Recommendation

Change the `is_voting_power_greater_than_threshold` function so that it uses “greater” when checking the threshold.

Status

Resolved through PR: <https://github.com/succinctlabs/tendermintx/pull/54/commits/380fda08a41a7ed8604669893f13681b03d6ec70>.

Missing checks in commit verification

Title	Missing checks in commit verification
Project	Celestia 2024 Q1: Blobstream Home
Type	IMPLEMENTATION
Severity	4 CRITICAL
Impact	3 HIGH
Exploitability	3 HIGH
Status	RESOLVED
Issue	

Involved artifacts

- [cometbft/types/validation.go](https://github.com/cometbft/types/validation.go)
- [cometbft/types/vote.go](https://github.com/cometbft/types/vote.go)
- [cometbft/types/canonical.go](https://github.com/cometbft/types/canonical.go)

Description

A central check check in Tendermint lightclient verification is that a signed header can be trusted. For that, one needs to ensure that the signed header actually is valid, which involves whether the commit is valid, that is, that it actually corresponds to a finalized block from the chain that is observed. Some checks are missing, in particular, that the validators all signed the same consensus messages.

Problem Scenarios

The Tendermint consensus algorithm, in principle (in each height) can go through multiple rounds, and different validators can send (and sign) precommit messages for different blockIDs in different rounds. A block becomes decided (finalized), once there are more than 2/3 precommits for the same BlockID, height, and round. In principle, there could be more that 2/3 precommits from different rounds for BlockID A, but in the end BlockID B ends up being decided. In such a scenario, an attacker may compose a commit using correctly signed precommit messages for different rounds for Block A. So there needs to be a check in place where this is ruled out, that is, that >2/3 signed a precommit message for the same height, round, and blockID.

Recommendation

On the Golang implementation side, it looks like this: `verifyCommit` calls `verifyCommitBatch` which uses `VoteSignBytes` to compute what vote message actually needs to be signed over, which uses `CanonicalizeVote` that contains among others the round number. The Rust implementation should have equivalent checks in place.

Status

Verify height in step resolved through PRs:<https://github.com/succinctlabs/tendermintx/pull/54/commits/7bfdeaf810abf04e7c6d36c566e0e6f2b366ea3b> and <https://github.com/succinctlabs/tendermintx/commit/779649e1e3e566c5ffdac39bb54d483cd0c6947c>.

Skip function does not check if the trusted and untrusted header are non adjacent

Title	Skip function does not check if the trusted and untrusted header are non adjacent
Project	Celestia 2024 Q1: Blobstream Home
Type	IMPLEMENTATION
Severity	1 LOW
Impact	2 MEDIUM
Exploitability	1 LOW
Status	RESOLVED
Issue	

Involved artifacts

- cometbft/light/verifier.go
- tendermintx/contracts/src/TendermintX.sol

Description

Tendermint light client verification of the non adjacent headers checks if the incoming untrusted header height is not adjacent to the trusted header. This is done in the `VerifyNonAdjacent` function [here](#).

TendermintX verification of the non adjacent headers (`skip` function) does not check this. [The check regarding the range](#) of the incoming untrusted block allows the incoming block to be adjacent to the trusted one.

Problem Scenarios

The `skip` function can be called for the adjacent blocks when it is more secure to call the specialized counterpart function named `step`. If everything is within the Tendermint Security model (at most 1/3 faulty validators) both approaches are safe. Attacking the skip skip would require +1/3 validators, while attacking step required +2/3, which is more expensive in terms of staked tokens. As a result, the +2/3 check should be done whenever possible.

Recommendation

Incorporate a check that the trusted header and the untrusted one are non adjacent in the skip function.

Status

Resolved through PR:<https://github.com/succinctlabs/tendermintx/pull/54/commits/c5d8759adf0dac5db4717a580ff409b098fa6074>.

Function parameter naming is confusing

Title	Function parameter naming is confusing
Project	Celestia 2024 Q1: Blobstream Home
Type	PRACTICE
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Status	RESOLVED
Issue	

Involved artifacts

- [tendermintx/circuits/skip.rs](#)

Description

Naming of the parameters in `verify_skip` and the functions that are called within are not intuitive. For example, `verify_skip` has the parameter called `header` which is representing `target_header` (can be seen [here](#)). It would be easier to read the code if the `verify_skip` and the inner called functions keep the naming of these parameters. (For example, it is the same with the following parameters: `validators` (called with `target_block_validators`), `chain_id_proof` (`target_header_chain_id_proof`)...).

An good example of a continuation of naming the parameter the same is the `target_block` parameter.

Problem Scenarios

This bad naming of parameters makes it difficult to understand the code logic.

Recommendation

Modify the functions to have same names for the same parameters.

Status

Resolved through PR: <https://github.com/succinctlabs/tendermintx/pull/54/commits/dd6daae98e6f877bf37f6b8e36cc4cb03a61b470> .

Minor code readability issue

Title	Minor code readability issue
Project	Celestia 2024 Q1: Blobstream Home
Type	PRACTICE
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Status	RESOLVED
Issue	

Involved artifacts

- [tendermintx/circuits/builder/shared.rs](#)

Description

In `verify_block_height` function there is a `for loop` that is used for extending the `encoded_height` to 64 bytes:

```
for _i in PROTOBUF_VARINT_SIZE_BYTES + 1..64 {
    encoded_height_extended.push(self.constant::<ByteVariable>(0u8));
}
```

Being that the `encoded_height_extended` will always be the same length is not needed. The loop could be written without it:

```
for _i in encoded_height_extended.len()..64 {
    encoded_height_extended.push(self.constant::<ByteVariable>(0u8));
}
```

Problem Scenarios

Affects the readability of the code.

Recommendation

In the description.

Status

Resolved through PR: <https://github.com/succinctlabs/tendermintx/commit/eb881358417ab8abedc5ab94ec0bfcd617aacfb0>.

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.


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:

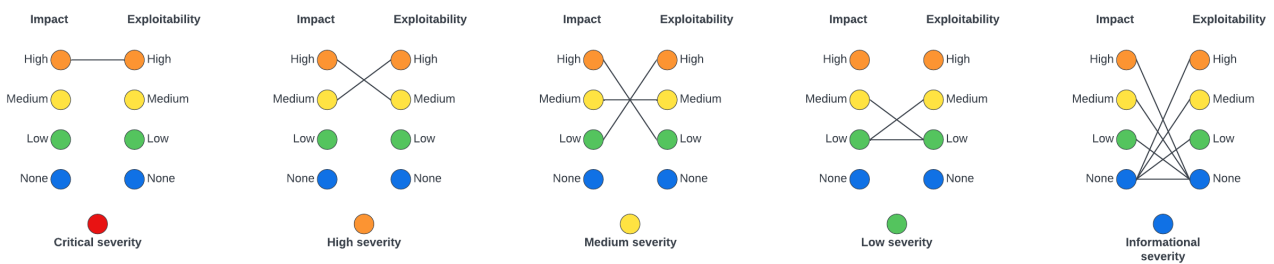
- *Actors* can be any entity that interacts with the system: other blockchains, system users, validators, relayers, but also uncontrollable phenomena (e.g. network delays or partitions).
- *Actions* can be

- *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

Severity Score	Examples
High	Permanent loss of user funds via a combination of submitting a specially crafted transaction with delaying of certain messages by a large portion of relayers
Medium	Substantial unexpected delays in processing user requests via a combination of delaying of certain messages by a large group of relayers with coordinated withdrawal of funds by a large group of users
Low	2x increase in node computational requirements via coordinated withdrawal of all user tokens
🌐 Informational	Code inefficiencies; bad code practices; lack/incompleteness of tests; lack/incompleteness of documentation; any exploit for which a coordinated illegitimate action of all actors is necessary