Security Audit Report

Celestia: Q4 2023 QGB

Authors: Andrija Mitrovic, Mirel Dalcekovic

Last revised 26 October, 2023

Table of Contents

Audit overview

The Project

In October 2023, Informal Systems has conducted a security audit for Celestia of the Quantum Gravity Bridge, which has since been renamed to Blobstream.

The main focus of the audit was the correctness and functionality of the protocol components.

The audited consisted of inspecting the following repositories:

- celestia-app/x/qgb: commit hash is 6515446b
- QuantumGravityBridge contracts: commit hash is cf301adf

The audit took place from October 9, 2023 through October 27, 2023 by the following personnel:

- Andrija Mitrovic
- Mirel Dalcekovic

Conclusions

In general, we found the codebase to be of very high quality: the code is well structured and easy to follow, and all functions contain good unit-tests. The documentation was good and code was very well commented and easy to follow. In the audit, we identified 7 issues, the majority of which were informational, with one categorized as low severity. These findings were collectively acknowledged by the team and subsequently reported on GitHub.

Further Increasing Confidence

To enhance the level of confidence in the QGB audit, it is essential to conduct an audit of both the relayer and orchestrator components. This audit would aim to thoroughly examine any potential issues inherited from the design of the gravity bridge, given that the QGB flow heavily relies on the functionality of these two off-chain components.

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 bug free 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.

Audit Dashboard

Target Summary

- **Type**: Specification and Implementation
- **Platform**: Go, Solidity
- **Artifacts:**
    - x/qgb module over tag: v1.0.0
    - QuantumGravityBridge.sol over commit hash: cf301ad
    - Inclusion Verifier DAVerifier.sol over commit hash: cf301ad
    - Libraries implementation over commit hash: cf301ad:
        - Binary
        - nmt

Engagement Summary

- **Dates**: 09.10.2023 until 27.10.2023.
- **Method**: Manual code review & protocol analysis
- **Employees Engaged**: 2

Severity Summary

Finding Severity	**#**
Critical	0
High	0
Medium	0
Low	1
Informational	6
Total	7

Resolution Status Summary

Resolution Status	**#**
Resolved	0
Acknowledged/Reported	6
Functioning as Designed	1
Total	7

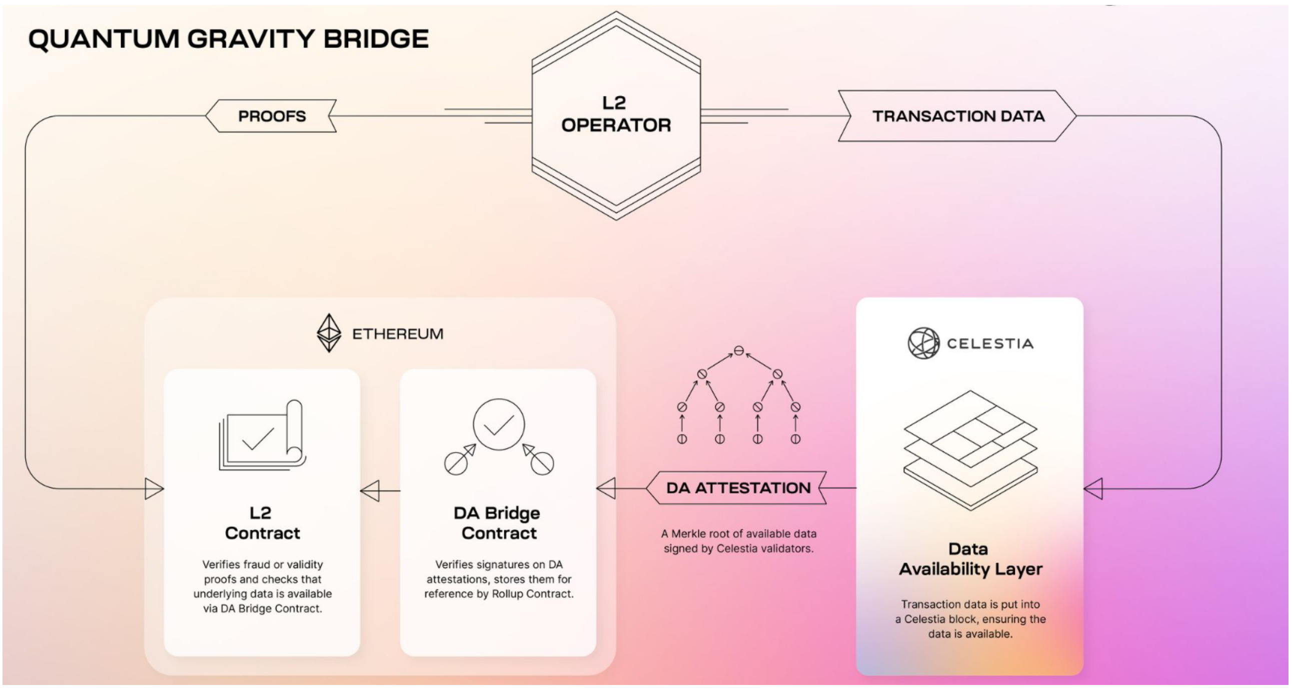
System overview

The persistent rise in gas costs on the Ethereum network has spurred the development of Layer 2 (L2) solutions aimed at reducing Ethereum usage expenses. While these L2 solutions employ various approaches to cost reduction, they don't entirely circumvent the need to pay gas per byte of data on Ethereum. Enter Celestia, serving as the Data Availability (DA) layer for Ethereum rollups, resulting in the creation of "Celestiums."

In this setup, Ethereum L2 operators can transmit their transaction data to the Celestia network. Celestia's Proof of Stake (PoS) validators then organize this data into blocks. A data availability attestation, represented by a Merkle root of L2 data and verified by Celestia validators, confirms the data's presence on Celestia. This attestation is facilitated by the Quantum Gravity Bridge (QGB), a one-way bridge from Celestia to Ethereum.

However, it's crucial to note that the QGB is solely designed to relay Celestia block header data roots to Ethereum and doesn't support direct bridging of assets like tokens or facilitate messages from Ethereum back to Celestia.

A diagram that represents the usage of Celestia as a DA for Ethereum rollups is given:

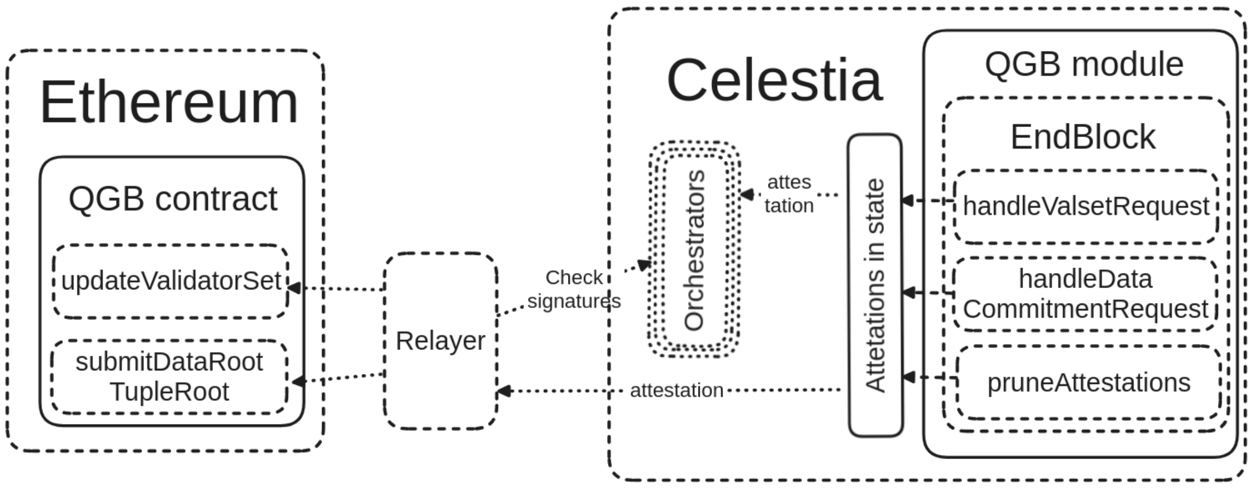

QUANTUM GRAVITY BRIDGE
PROOFS
L2 OPERATOR
TRANSACTION DATA
ETHEREUM
L2 Contract
Verifies fraud or validity proofs and checks that underlying data is available via DA Bridge Contract.
DA Bridge Contract
Verifies signatures on DA attestations, stores them for reference by Rollup Contract.
DA ATTESTATION
A Merkle root of available data signed by Celestia validators.
CELESTIA
Data Availability Layer
Transaction data is put into a Celestia block, ensuring the data is available.

As it can be seen from the previous diagram Celestia's Quantum Gravity Bridge contract is integrated into Ethereum. Ethereum Layer 2 operators send their transaction data to Celestia, where it is structured into blocks by Celestia's PoS validators. This data is then relayed as a data availability attestation from Celestia to Ethereum, with the Quantum Gravity Bridge verifying these attestations. Ethereum's Layer 2 contract can verify updates by checking that the required data is available on Celestia via the bridge contract, ensuring data integrity.

The QGB implementation consists of the following three components:

1. The state machine (on Celestia),
2. The orchestrator-relayer,
3. The QGB smart contract (on Ethereum).

These components are presented in the following diagram:



A detailed explanation of these components will follow.

State machine

State machine is a part of the Celaestia application and is implemented as a module named `qgb` within it. Its purpose is to create attestations within the state that should be signed by orchestrators (processes run by Celestia validators) and then queried and relayed by relayers to the QGB smart contract.

There are two types of attestations:

- **Valsets -** attestation type representing snapshots of validator sets with their staking power, used by the QGB smart contract to monitor Celestia's validator set status. Generated within the state machine, orchestrators sign and submit them to the QGB P2P network. Once over 2/3 of Celestia validators endorse a valset, relayers transmit it, along with signatures, to the QGB contract. This ensures the contract's ability to verify the correct Celestia validator set and majority support for attestations that follow. Valsets facilitate changes in the validator set to determine which orchestrator should sign attestations.
- **Data commitments -** attestation type representing requests to commit to a specific range of blocks, allowing orchestrators to sign and share them in the QGB P2P network. This range is determined by the `DataCommitmentWindow` parameter. Orchestrators sign and submit data commitments when they are part of the validator set for the previous valset. These commitments are then relayed to the QGB smart contract, where they are verified and saved, enabling the verification of Merkle-based inclusion proofs for any data posted in the committed blocks.

The State machine logic is executed in the EndBlock, and it consists of three functions:

1. handleValsetRequest - generating new valsets in case that there is no valset in store, validator starts unbonding or when there is a significant power change.
2. handleDataCommitmentRequest - generates a new data commitment when a sufficient number of blocks have passed since the previous one.
3. pruneAttestations - prunes all expired attestations from state after running basic checks on saved attestations. If the all the attestations in store are expired, which is an edge case that should never occur, the QGB state machine doesn't prune the latest attestation.

QGB Orchestrator and Relayer **(Not part of the audit)**

Orchestrator

Orchestrator is a process run by a Celestia validator and which purpose is to sign attestations using its corresponding validator EVM private key. These orchestrators are connected to a separate P2P network on which they sign attestations.

The orchestrator does the following:

1. Connect to a Celestia-app full node or validator node via RPC and gRPC and wait for new attestations
2. Once an attestation is created inside the QGB state machine, the orchestrator queries it.
3. After getting the attestation, the orchestrator signs it using the provided EVM private key. The private key should correspond to the EVM address provided when creating the validator.
4. Then, the orchestrator pushes its signature to the P2P network it is connected to, via adding it as a DHT value.
5. Listen for new attestations and go back to step 2.

Relayer

Relayer is a separate process that can be run by anyone and its role is to gather attestations' signatures from the orchestrators, and submit them to a target EVM chain (more specifically to the QGB smart contract).

The relayer works as follows:

1. Connect to a Celestia-app full node or validator node via RPC and gRPC and wait for attestations.
2. Once an attestation is created inside the QGB state machine, the relayer queries it.
3. After getting the attestation, the relayer checks if the target QGB smart contract's nonce is lower than the attestation.
4. If so, the relayer queries the P2P network for signatures from the orchestrators.
5. Once the relayer finds more than 2/3s signatures, it submits them to the target QGB smart contract where they get validated.
6. Listen for new attestations and go back to step 2.

QGB Smart Contract

QGB works by relying on a set of signers to attest to some event on Celestia: the Celestia validator set. The QGB contract keeps track of the Celestia validator set by updating its view of the validator set with `updateValidatorSet()`. More than 2/3 of the voting power of the current view of the validator set must sign off on new relayed events, submitted with `submitDataRootTupleRoot()`. Each event is a batch of `DataRootTuple`s, with each tuple representing a single data root (i.e. block header). Relayed tuples are in the same order as Celestia block headers.

There are two types of events and messages that can be relayed to the smart contract: validator sets and batches.

Validator sets

The relayer informs the QGB contract who are the current validators and their power. This results in an execution of the updateValidatorSet function which allows the contract to stay up to date with the Celestia validator set changes. Update happens if the current set of validators have signed of the new set. Anyone can call this function, but they must supply valid signatures of the current validator set over the new validator set.

This function performs the following checks:

- Check that the new nonce is one more than the current one.
- Check that current validators and signatures are well-formed.
- Check that the supplied current validator set matches the saved checkpoint.
- Check that enough current validators have signed off on the new validator set (checkValidatorSignatures).

Valset digest is generated within the domainSeparateValidatorSetHash method. This digest is computed as the keccak256 hash of a combination of the following components: a predefined constant called `VALIDATOR_SET_HASH_DOMAIN_SEPARATOR`, the universal nonce indicating a validator set change, the power threshold representing two-thirds of the validator set, and the hash of the validator set itself, determined using the computeValidatorSetHash function.

Batches

The relayer informs the QGB contract of new data root tuple roots. This results in an execution of the submitDataRootTupleRoot function. Anyone can call this function, but they must supply valid signatures of the current validator set over the data root tuple root.

This function performs the following checks:

- Check that the new nonce is one more than the current one.
- Check that current validators and signatures are well-formed.
- Check that the supplied current validator set matches the saved checkpoint.
- Check that enough current validators have signed off on the data root tuple root and nonce.

The data (batch) commitment digest is generated using the domainSeparateDataRootTupleRoot method by taking the keccak256 hash of a combination of constant elements, including `DATA_ROOT_TUPLE_ROOT_DOMAIN_SEPARATOR`, a universal nonce denoting the data commitment, and the data root tuple root representing a commitment over a specific set of blocks.

Threat inspection

During the initial stage of the audit the auditing team spent time onboarding to the project's scope and identifying potential threats. Analysis of the possibility of realization of those threats was done during code inspection. Our primary objective was to validate the design's correctness and identify any issues in the code base that could result in the realization of these threats. To achieve this, we held weekly sync meetings with the Celestia QGB team to discuss possible flaws in the system and gain insights into the current behavior of the system. Through these efforts, we were able to validate that the identified threats could not/could be realized.

The following analysis contains our conclusions and the explanation of the audited product design and implementation on how those threats are mitigated. In addition to our threat inspection analysis, we have documented our results as findings.

It should be noted that, while we do our best in the analysis below, a security audit can by no means guarantee the absence of threats: the real guarantees can be provided only via systematic quality assurance, including manual and automated testing, and, ultimately, via the usage of formal methods, such as model-based testing or formal verification.
It's essential to emphasize the limited scope of this audit project. Notably, the QGB design includes two off-chain components, namely the **Relayer** and **Orchestrator**, which play crucial roles within the system. These components, however, were not subject to the audit's assessment. The inclusion of off-chain components raises the potential for additional threats, some of which are highlighted below:

- **Congestion Due to High Activity:** In the event of a substantial surge in attestations and submissions, if not effectively managed, the activities of these off-chain components could result in network congestion.
- **Malicious Behavior:** The deliberate malicious actions of any of these off-chain components have the potential to disrupt the network, causing congestion or even a complete halt.
- **Impact on Ethereum Rollup:** Failure in QGB communication, such as the absence of attestation processing, could potentially impact the Ethereum rollup.
- **Assumptions and Their Implications:** Assumptions concerning the expected behavior of off-chain components may lead to severe issues if these assumptions are not met. For example, the assumption of having at least one honest relayer within the system is critical for the proper functioning of the QGB and its support for Ethereum rollup.

The design of the Gravity Bridge served as the foundation for the Quantum Gravity Bridge. An audit team found that the Quantum Gravity Bridge design had addressed several design flaws that were present in the Gravity Bridge, including:

- Unclear Rules to Issue Validator Set Updates

However, it appears that some similarities in protocol and architecture still persist:

- No Independent Validation in Ethereum
- No Trusting Period Check in Ethereum

Threat: Valset power change influence on consensus over the attestation

One of the three conditions for a validator set change on the QGB smart contract is when there's a significant shift in the power allocation within the validator set, exceeding the specified threshold (5%). This condition can have implications for the decision-making process on the QGB, particularly in terms of achieving a 66% agreement with the attestation.

Attack vectors:

- **Unbonding Validators Leaving the Active Valset:** When validators exit the current validator set due to unbonding or a power change, they must still sign attestations during the unbonding period. A malicious validator might exploit this transition phase by intentionally omitting the attestation signature.
- **Significant Power Changes in Current Valset:** In instances where multiple validators adjust their power levels, leading to an update or reorganization of the validator set that involves different power threshold values, the network's dynamics can shift, potentially impacting attestation consensus.

If more than 33% of the validators power threshold is changed with the reasons listed above the impact on the QGB SC attestation acceptance is under risk.

Threat inspection

- **Unbonding Validators threat inspection**: The QGB design includes the implementation of the **unbonding hook** within the QGB module, signifying a substantial enhancement in contrast to the Gravity Bridge design. Upon a validator initiating the unbonding period, the hook's code is executed within the QGB module, subsequently updating the `LatestUnBondingBlockHeight` to the current block height.
  This, in turn, serves as a signal during the end block process - here, indicating that the current validator set should be replaced with a new validator set.
  This implementation addresses the potential issue of the QGB smart contract expecting validators undergoing unbonding to sign attestations.
- **Signficant Power Changes threat inspection:** The QGB design includes two important aspects:
  - validators are always ordered by their current power in the valset
  - when sending the attestation containing the new valset, QGB module is also providing the newThreshholdPower within the attestation data.

  With the two, after the validator set is updated both with validators ordered by power and new powerThreshold, all the attestations should be signed accorging to new bridging consensus rules prior to being accepted on the QGB smart contract.

Findings:

- No findings reported during this analysis.

Threat: Disrupting next nonce processing on QGB Smart contract due to attestation pruning

The current design of the QGB module may have a flaw when it comes to attestation pruning, as it exhibits the potential issues:

1. The QGB module lacks awareness of whether attestations have been successfully accepted on the QGB Smart Contract side.
2. Similarly, the QGB Smart Contract is unaware of the process of attestation pruning.

These shortcomings in the design could potentially introduce vulnerabilities and inconsistencies in the system, impacting its overall performance and integrity.

The combination of both issues — unnoticed attestation pruning and the smart contract's unawareness — could result in a critical impact on the QGB Smart Contract. The QGB Smart Contract may not receive the nonce it expects as the next one (nonce value in state + 1), which can disrupt its operations and the overall system's security.

Threat inspection:

Pruning takes place within the QGB EndBlocker logic as the concluding action for each block. It involves the removal of all attestations up to a designated AttestationExpiryTime, which is currently configured to align with the consensus unbonding period of 3 weeks.

```
    // AttestationExpiryTime is the expiration time of an attestation. When this
    // much time has passed after an attestation has been published, it will be
    // pruned from state.
    AttestationExpiryTime = 3 * oneWeek // 3 weeks
```

If all the existing attestations are expired, the latest one will not be pruned, so pruning is done until `latestAttestationNonce`:

```
for newEarliestAvailableNonce = earliestAttestation.GetNonce();
```

```
    newEarliestAvailableNonce < latestAttestationNonce;
    newEarliestAvailableNonce++ { ... }
```

Possible reasons for reaching this situation include:

1. The absence of an honest or active relayer capable of relaying signed attestations during the `attestationExpiryTime`.
2. A shortage of honest or functional orchestrators to facilitate validator signing, or a deficiency of validators with the requisite power threshold for the attestation to be accepted on the QGB smart contract.
3. Inability of relayer to transfer the attestation to QGB smart contract on Ethereum (due to high transaction fees, lost connections…)

Once relaying is reestablished, the QGB contract will anticipate the oldest pruned attestation nonce as the next one to be processed, rather than the youngest attestation that was not deleted in the QGB module (which is updated as the `latestAttestationNonce`).

Findings:

- No trusting period or attestation expiry time check in Ethereum

Threat: System congestion or halt due to invalid or non sequential nonce

Could a gap between attestation nonces occur so that this check fails leading to stop of data processing on QGB contract?

Attack vectors:

1. **Disorderly Attestation Submissions:** The ordering of attestation submissions may become disorganized, potentially leading to adverse effects on QGB bridge processing.
2. **Malicious Relayer Exploitation:** A malicious relayer could circumvent the requirement for data commitment attestation submission to be verified by the current validator set. Instead, they might send the valset update attestation, posing a security risk.

Threat Inspection

The occurrence of potential situations is rendered impossible by the robust design of the QGB. This design incorporates

- **Synchronous Bridge Approach:** The utilization of a synchronous bridge approach ensures that the various components of the QGB bridge remain synchronized, reducing the risk of vulnerabilities.
- **Universal Nonces Approach:** Both `ValsetConfirm` and `DataCommitmentConfirm` employ a universal nonce mechanism, wherein they increment the same nonce. This enhances consistency and reduces the likelihood of inconsistencies or vulnerabilities.
- **QGB Smart Contract Safeguard**: The QGB Smart Contract includes a stringent check that verifies the nonce of the incoming attestation against the universal nonce. If the nonce of the attestation is not incremented by 1, it is promptly rejected, ensuring the integrity and security of the system.

```
// Check that the new nonce is one more than the current one.
if (_newNonce != currentNonce + 1)
{
  revert InvalidValidatorSetNonce();
}
```

Findings:

- No findings reported during this analysis.

Threat: Invalid new validator set update on QGB contract halts QGB bridge

Updating the validator set is initiated by an off-chain component - relayer.
There is a risk of an incorrect `newValidatorSetHash` being saved on the QGB contract. Such an error has the potential to render bridging impossible at the point where it becomes necessary to verify incoming attestations against the updated validator set signatures.

This would result in the obstruction of Celestia attestation's bridging and the updating of the validator set, thus failing to address the issue. To guarantee the success of both operations, it is imperative to verify that the attestation, which may contain new valset, has been signed by a sufficient threshold of the current validators. The validation of a legitimate validator set is achieved through the comparison of the `checkpoint` here:

```
  // Check that the supplied current validator set matches the saved checkpoint.
   bytes32 currentValidatorSetHash = computeValidatorSetHash(_currentValidatorSet);
     if (
         domainSeparateValidatorSetHash(_oldNonce, currentPowerThreshold,
 currentValidatorSetHash)
             != lastValidatorSetCheckpoint
      ) {
          revert SuppliedValidatorSetInvalid();
     }
```

Additionally, the transmission of invalid data within signatures has the potential to bring the QGB bridge to a halt here.

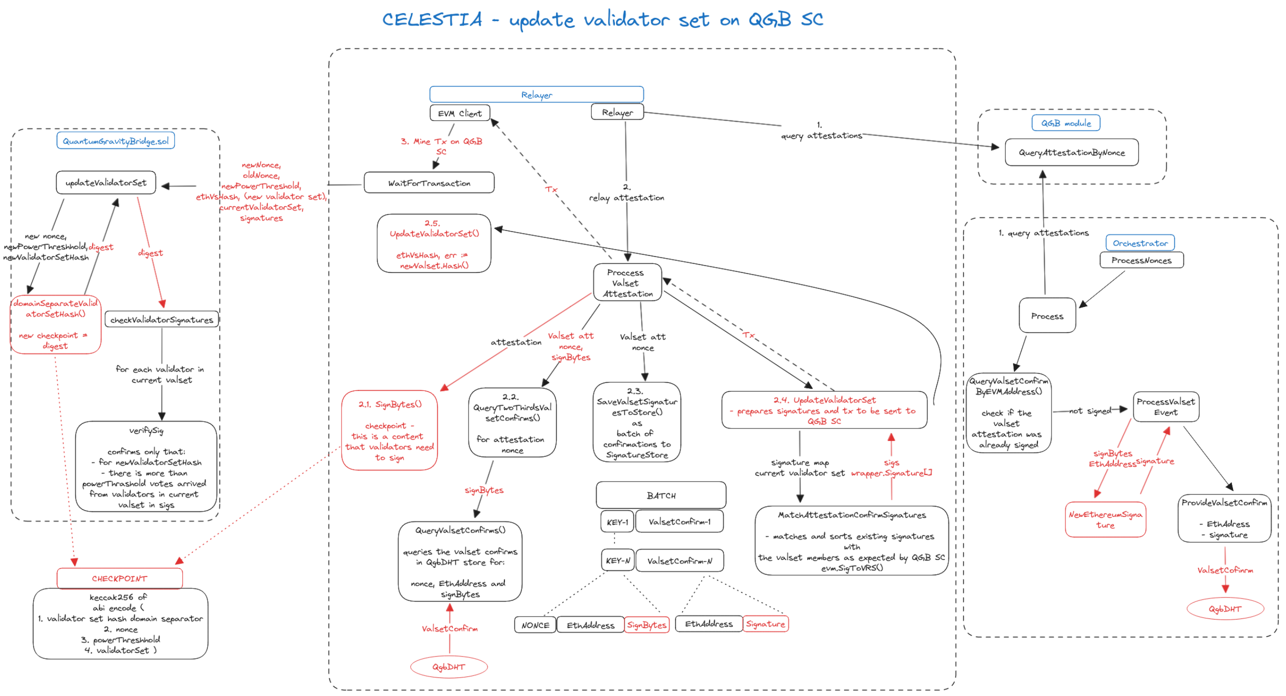
The analysis must encompass considerations regarding potential threat realization in the presence of a malicious relayer or if there are security vulnerabilities related to the verification of signatures.

Attack vectors:

- **Invalid new validator set hash:** Invalid hash is sent by the relayer, together with signatures and the current validator set. If there is no check to verify that the exact content of the hash was signed by the current set of validators, a malicious relayer could change the hash and cause the QGB contract to update the validator set to an invalid checkpoint.
- **Invalid power threshold**: A malicious relayer could set a new power threshold to a value higher or lower than the valid, signed threshold, and produce attestations that may be wrongly rejected or accepted.
- **Invalid number of signatures** sent to the QGB contract in `sigs`. The number of signatures must align with the number of validators in the current validator set. Any attempt to send an invalid number of signatures should be detected and prevented.

Threat Inspection

During our threat inspection, we examined the validator set update process to confirm the potential existence of this threat. To accomplish this, we conducted an in-depth review of the orchestrator and relayer code, paying particular attention to specific, critical, details. This level of scrutiny was necessary as the accuracy of this process relies on the performance of off-chain components.


CELESTIA - update validator set on QGB SC
QuantumGravityBridge.sol
updateValidatorSet
new nonce, newPowerThreshhold, newValidatorSetHash
digest
domainSeparateValidatorSetHash()
new checkpoint = digest
checkValidatorSignatures
for each validator in current valset
verifySig
confirms only that: - for newValidatorSetHash - there is more than powerThrashold votes arrived from validators in current valset in sigs
CHECKPOINT
keccak256 of abi encode (1. validator set hash domain separator 2. nonce 3. powerThreshhold 4. validatorSet)
newNonce, oldNonce, newPowerThreshold, ethVsHash, (new validator set), currentValidatorSet, signatures
Relayer
EVM Client
Relayer
3. Mine Tx on QGB SC
WaitForTransaction
Tx
1. query attestations
2. relay attestation
2.5. UpdateValidatorSet() ethVsHash, err := newValset.Hash()
Process Valset Attestation
attestation
Valset att nonce, signBytes
Valset att nonce
Tx
2.1. SignBytes() checkpoint - this is a content that validators need to sign
2.2. QueryTwoThirdsValsetConfirms() for attestation nonce
2.3. SaveValsetSignaturesToStore() as batch of confirmations to SignatureStore
2.4. UpdateValidatorSet - prepares signatures and tx to be sent to QGB SC
signature map current validator set
sigs wrapper.Signature[]
signBytes
QueryValsetConfirms() queries the valset confirms in QgbDHT store for: nonce, EthAddress and signBytes
ValsetConfirm
QgbDHT
BATCH
KEY-1
ValsetConfirm-1
KEY-N
ValsetConfirm-N
NONCE
EthAddress
SignBytes
EthAddress
Signature
MatchAttestationConfirmSignatures - matches and sorts existing signatures with the valset members as expected by QGB SC evm.SigToVRS()
QGB module
QueryAttestationByNonce
1. query attestations
Orchestrator
ProcessNonces
Process
QueryValsetConfirmByEVMAddress() check if the valset attestation was already signed
not signed
ProcessValset Event
signBytes EthAddress
Signature
NewEthereumSignature
ProvideValsetConfirm - EthAdress - signature
ValsetCofinrm
QgbDHT

We have identified the following steps and existing checks:

1. When attestation processing for an updated validator set is triggered, the first step is to retrieve the `signBytes` of the attestation. Source is QGB state machine, where the `signBytes` are created and represent the keccak256 hash of abi encoded validator set hash domain separator, nonce, new power threshold and new validator set. These bytes should be signed by the validators.
2. Confirm that more that two thirds of current validator set has signed the attestation's `signBytes` - the check is performed over `QgbDHT` store. `EthAddress` validator signature for `nonce` and `signBytes` should be present in the store, upon which the `currThreshold` is calculated. If the majority of needed votes are present, the validity of the attestation is confirmed.
3. Signatures are saved in relayer's `Signature Store` in a batch, containing `key` (`nonce` , `EthAddress` , `signBytes`) and the `value` - `ValsetConfirm` for each validator that signed this attestation.
4. This means that `UpdateValidatorSet` `tx` should be relayed to QGB smart contract. To accomplish this, the transaction should be created using the EVM client's UpdateValidatorSet function. In this process, an `ethVsHash` is generated for the `newValset` , which will be sent in wrapped tx with `newThreshhold` value to the QGB Smart Contract on execution.
   The creation of this transaction by relayer is a potential threat, as a malicious relayer could potentially transmit incorrect `hash` values, `newThreshold` values, and invalid signature (`sigs`) data. The absence of security signature checks within the QGB Smart contract could mean that there is no mechanism to validate whether the provided hash matches the one endorsed by the current validator set.
5. On the QGB Smart contract side, it is crucial to ensure that any attempt to halt the QGB bridge is effectively mitigated to maintain uninterrupted operation.
   a. In the event of a malicious relayer sending invalid `hash` , `nonce` , or an incorrect number of signatures in `sigs` , the presence of at least one honest relayer ensures that the bridge cannot be halted.
   b. Orchestrators, operated by validators off-chain, are responsible for signing hashed details of the new validator set as mentioned in point 1. They create `ValsetConfirm` during this process and store

it in the `QgbDHT`. To verify that the same data was signed by the current validator set, the signatures (`sigs`) received on the QGB Smart contract side are used to recover the hash received.

Findings:

- During this analysis there was issue reported: Redundant EthAdress in ValsetConfirm. The problem doesn't confirm the actualization of the threat but rather hints at potential enhancements in the current design.

Threat: Misalignment between verification and data submitting to QGB

The whole process of proving attestations sent from Celestia to QGB Smart Contract is dependent on alignment between the attestation creation and its verification on the Ethereum side. This means that the creation of attestation over a range of blocks must be done adequately in accordance with the proof verification on the contract side. If these protocols are in any way misaligned a possibility may arise that a invalid attestation could be verified as valid or vice versa.

Potential misalignment:

1. **Namespace Merkle tree functionalities misalignment -** QGB Verifyer implements functions for proving given proofs for leaves from within nmt. It is of utter importance that this protocol is aligned with the Namespace Merkle Tree library used in Celestia for generating row and column roots for the Extended Data Square in order to be able to do proper validation of proofs.
2. **Binary tree misalignment -** QGB Verifyer implements functions for proving given proofs for leaves from within binary tree. These should be aligned with the binary tree usage on Celestia for generating data root from `rowRoots` and `columnRoots`.
3. **Verification process misalignment -** QGB Verifyer needs to properly do a proving process from shares to the `dataRootTupleRoot`. Verification process should be aligned with the process of generation of `dataRootTupleRoot` from single shares on Celestia. The whole process must incorporate usage of nmt proof verification, binary tree proof verification and attestation verification.

Threat Inspection

1. Namespace Merkle Tree library implementation in solidity smart contract has no major differences with the nmt library in go. The only noticeable difference is the non existence of the parameter `IgnoreMaxNamespace` and the lib in solidity acts as this parameter is true on Celestia (which it is by default). Even though no issue is caused by this it can lead to maybe future problems.
2. Binary tree has no differences with the implementation used in Celestia-app for calculating data root of the block.
3. This verification process consists of three parts:
   a. verifying the attestation: verification is done in verifyAttestation that does the checking that the data root was committed to by the QGB smart contract. This is done by verifying the given proof against committed `dataRootTupleRoot.` This root represents the binary tree merkle root where the leafs of the tree are `dataRoots` of each block within the data commitment window. As it can be seen in the verifyAttestation the proof verification is done using the Binary Merkle tree.
   b. verifying the shares against corresponding row roots: Verify if shares exists in Merkle tree, given leaves, mutliproof and root. This is in accordance with the proof creation on Celestia, no differences found in the nmt libs.
   c. verifying row roots in the corresponding data root: Verify if the given `rowRoot` exists in Merkle tree, given data, proof, and `dataRoot`. This also corresponds to the process of creating the `dataRoot` from row and column roots using the Binary Merkle tree.

Findings:

- During this analysis there was issue reported:Difference between two nmt libraries .

Findings

Title	Type	Severity	Impact	Exploitability	Issue
Redundant EthAdress in ValsetConfirm	IMPLEMENTATION	1 LOW	1 LOW	1 LOW	https://github.com/celestiaorg/orchestrator-relayer/issues/559
Minor code improvements in x/qgb module	IMPLEMENTATION	0 INFORMATIONAL	0 NONE	0 NONE	https://github.com/celestiaorg/celestia-app/issues/2766
No trusting period or attestation expiry time check in Ethereum	PROTOCOL IMPLEMENTATION	0 INFORMATIONAL	3 HIGH	04 UNKNOWN	https://github.com/celestiaorg/blobstream-contracts/issues/243
Minor code improvements in qgb contracts	IMPLEMENTATION	0 INFORMATIONAL	0 NONE	0 NONE	https://github.com/celestiaorg/blobstream-contracts/issues/244
Unnecessary code	IMPLEMENTATION	0 INFORMATIONAL	0 NONE	0 NONE	https://github.com/celestiaorg/celestia-app/issues/2765
Code simplification	IMPLEMENTATION	0 INFORMATIONAL	0 NONE	0 NONE	https://github.com/celestiaorg/blobstream-contracts/issues/242
Difference between two nmt libraries	IMPLEMENTATION	0 INFORMATIONAL	0 NONE	0 NONE	https://github.com/celestiaorg/blobstream-contracts/issues/241

Minor code improvements in x/qgb module

Title	Minor code improvements in x/qgb module
Project	Celestia: Q4 2023 QGB
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Issue	https://github.com/celestiaorg/celestia-app/issues/2766
Status	

Involved artifacts

- x/qgb/abci.go
- x/qgb/types/genesis.go

Description

The minor code improvements listed within this finding do not pose a security threat nor do they introduce an issue. The suggestions are shared to improve the code readability, keep consistency, optimize, and improve logging:

- The following state reads are performed in EndBlocker, so no gas is used, but double reading is not needed: `dataCommitmentWindow` is read here & here, while it can be retrieved from the store once and propagated further.
- Add *maximum* in error log here, to make the message consistent to the log here.

Problem Scenarios

Findings listed above could not introduce any issues, they are suggestions for code improvements.

Recommendation

As described above.

Remediation Plan

Acknowledged/Reported: It was agreed with the team to report this finding as an issue on the Celestia's GitHub repository.

No trusting period or attestation expiry time check in Ethereum

Title	No trusting period or attestation expiry time check in Ethereum
Project	Celestia: Q4 2023 QGB
Type	PROTOCOL IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	3 HIGH
Exploitability	04 UNKNOWN
Issue	https://github.com/celestiaorg/blobstream-contracts/issues/243
Status	

Involved artifacts

- x/qgb/abci.go
- QuantumGravityBridge.sol

Description

The EndBlocker in the QGB module has a function to remove old attestations. Attestations are considered expired if they are older than the AttestationExpiryTime, which is currently set to the consensus unbonding period (3 weeks) - trusted period.
However, the Solidity contract doesn't verify the attestation's expiry time, nor does the GQB module keep track of when the attestation was last processed. As a result, the contract can't determine if the attestation has expired since the last processing, and it won't be aware of any pruning that occurred in the QGB module.

The Tendermint security model establishes a trusting period to ensure that validators with over 2/3 of the voting power, who signed the block, are behaving correctly during this period. In simple terms, once the trusting period is over, we must assume that validators could act arbitrarily.

The Solidity contract doesn't check this trusting period, and it doesn't record the last time the checkpoint was updated. Consequently, the contract can't confirm if the trusting period has passed since the last update. Even though the QGB design updates the validator set as soon as a validator starts unbonding, issues might arise if the attestation containing this update isn't accepted on the QGB Smart Contract.

Problem Scenarios

If no attestation processing (including valset updates and data commitments) occurs for a period longer than the attestation expiry time, it can be attributed to the following reasons:

1. Validators may not be signing.
2. It's possible that a relayer is unable to transfer the ValSetUpdate on Ethereum. This could be due to factors such as high transaction fees, lost connections, and so on.

3. There might not be active or honest relayers or orchestrators/validators available to reach the necessary voting power threshold.

As a result, the current `checkpoint` in the Solidity contract may become disconnected from the current status of bonded and unbonding validators on the Cosmos side.
It's essential to understand that the Solidity smart contract is:

- under the complete control of the validators within the checkpoint and
- expecting the nonce, next to be processed and is unaware of the pruning happened on the QGB module, which deleted nonces.

Recommendation

Address the issue with implementing a check for the age of the current checkpoint and time passed from the latest attestation processed. Alternatively, the design choice should be made explicit in the documentation.

Remediation Plan

Functioning as Designed: The team has communicated that this discovery holds true, but in the present phase, the Quantum Gravity Bridge does not emphasis liveness.
The intention is to shift the focus towards resolving liveness issues in the next iteration of QGB development, at which point this particular finding will be addressed on a protocol level.

Redundant EthAdress in ValsetConfirm

Title	Redundant EthAdress in ValsetConfirm
Project	Celestia: Q4 2023 QGB
Type	IMPLEMENTATION
Severity	1 LOW
Impact	1 LOW
Exploitability	1 LOW
Issue	https://github.com/celestiaorg/orchestrator-relayer/issues/559
Status	

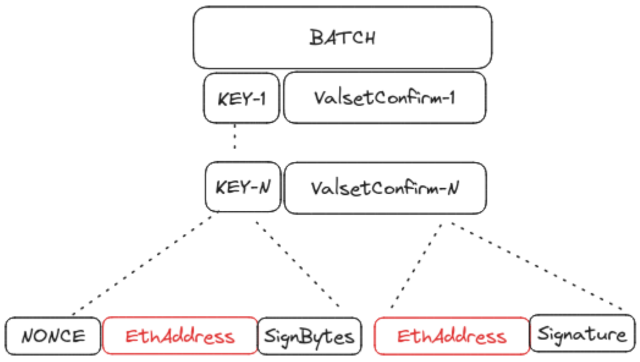
Involved artifacts

- types/valset_confirm.go
- orchestrator/broadcaster.go
- relayer/relayer.go

Description

During the analysis of update validator set flow, it was noticed that EthAddress was stored two times for one entry when:

- when orchestrator is placing the ValsetConfirm for attestation in the `qgbDHT` store
- when relayer is placing the ValsetConfirms in the `SignatureStore`



Problem Scenarios

The suggestions mentioned earlier can help save memory if the `EthAddress` field is not utilized in `ValsetConfirm`.

Recommendation

We suggest simplifying the stored values if possible and reconstructing the `ValsetConfirm`s if needed, upon reading the values from the store. In our analysis throughout this project, we did not identify any instances where the `ValsetConfirm`'s `EthAddress` field was used.
However, it's important to note that a detailed audit of the relayer and orchestrator code has not been conducted, so we cannot definitively assert this to be the case.

Remediation Plan

Acknowledge/Reported: The team agreed that this is a valid issue and that it should be reported to the Celestia GitHub repository. The team has already planned to remove `ValsetConfirm` from the orchestrator and relayer implementation entirely.

Minor code improvements in qgb contracts

Title	Minor code improvements in qgb contracts
Project	Celestia: Q4 2023 QGB
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Issue	https://github.com/celestiaorg/blobstream-contracts/issues/244
Status	

Involved artifacts

- blobstream-contracts/src/QuantumGravityBridge.sol
- blobstream-contracts/src/lib/verifier/DAVerifier.sol
- blobstream-contracts/src/lib/tree/binary/BinaryMerkleTree.sol
- blobstream-contracts/src/lib/tree/namespace/NamespaceMerkleTree.sol

Description

The minor code improvements listed within this finding do not pose a security threat nor do they introduce an issue. The suggestions are shared to improve the code readability, keep consistency, optimize, and improve logging:

1. Doubled the word "root" within the comment.
2. A comment that validators that did not signed the attestation will have nil value in `sigs` here should be added to improve readability of code, because looking at this part of the code it can be only assumed that all the validators have to sign the data. There is a similar comment in checkValidatorSignatures.
3. Wrong comment placements (should be after the following if blocks):
    a. comment 1
    b. comment 2
    c. comment 3

Problem Scenarios

Findings listed above could not introduce any issues, they are suggestions for code improvements.

Recommendation

As described above.

Remediation Plan

Acknowledged/Reported: The Celestia Team agreed that this finding is valid and they will change the comments as described above, since the team pays special attention to the documentation and code inline comments.

Unnecessary code

Title	Unnecessary code
Project	Celestia: Q4 2023 QGB
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Issue	https://github.com/celestiaorg/celestia-app/issues/2765
Status	

Involved artifacts

- /x/qgb/abci.go

Description

In `pruneAttestations` function the whole earliest attestation is taken out of the store here using `GetEarliestAvailableAttestationNonce`. This is done just to get the earliest nonce to use for newEarliestAvailableNonce in the for loop and for updating state and logging here and here. It would be sufficient to get the earliest nonce directly by using `GetEarliestAvailableAttestationNonce`.

Problem Scenarios

Unnecessary access to the storage.

Recommendation

Just use the functions `CheckEarliestAvailableAttestationNonce` and `GetEarliestAvailableAttestationNonce` for getting earliest nonce.

Remediation Plan

Acknowledged/Reported: The Celestia Team agreed that this finding is valid and they will remove unnecessary parts of the code.

Code simplification

Title	Code simplification
Project	Celestia: Q4 2023 QGB
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Issue	https://github.com/celestiaorg/blobstream-contracts/issues/242
Status	

Involved artifacts

- blobstream-contracts/src/lib/tree/namespace/TreeHasher.sol

Description

NMT assumes that the leafs are ordered by namespace. This assumption is not used here as it is used in the rest of the code.

Problem Scenarios

Findings listed above could not introduce any issues, they are suggestions for code improvements.

Recommendation

Calculated minimum namespace is always equal to the left minimum namespace so it could be calculated as follows:

```
Namespace memory min = l.min
```

This is how it is done in the nmt library that is used in Celestia:
minNs := leftMinNs

Remediation Plan

Acknowledged/Reported: The Celestia Team agreed that this finding is valid and they will simplify the code.

Difference between two nmt libraries

Title	Difference between two nmt libraries
Project	Celestia: Q4 2023 QGB
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Issue	https://github.com/celestiaorg/blobstream-contracts/issues/241
Status	

Involved artifacts

- blobstream-contracts/src/lib/tree/namespace/TreeHasher.sol

Description

This calculation of min and max namespace of the node within nmt does not correspond to the calculation in the nmt library. It actually functions the same under the assumption that `ignoreMaxNs` in the nmt library is set to true which is by default set to true in the NMT go-lang library.

Problem Scenarios

Finding listed above does not introduce any issues. It is a suggestion for preventing future issues.

Recommendation

-

Remediation Plan

Acknowledged/Reported: The Celestia Team acknowledged the differences but states that the parameter will stay true on Celestia side. They will think about potential Celestia forks and potential changes to this parameter and their effect.

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:

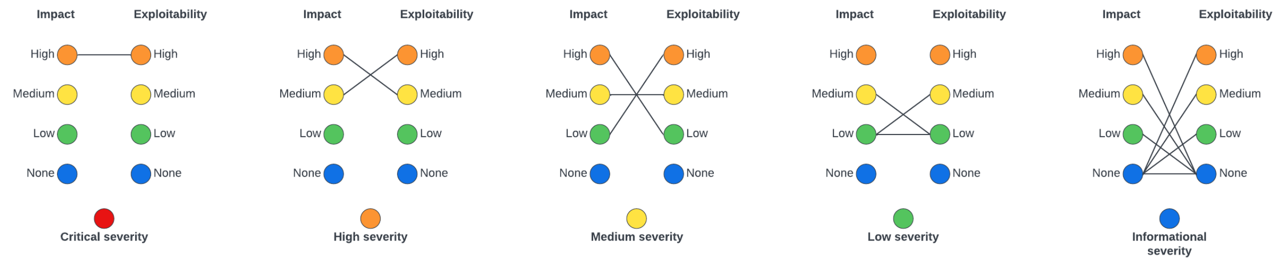
- *Actors* can be any entity that interacts with the system: other blockchains, system users, validators, relayers, but also uncontrollable phenomena (e.g. network delays or partitions).
- *Actions* can be

    - *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**
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