Security Audit Report

Celestia: Q3 2023

Authors: Andrija Mitrovic, Ivan Gavran

Last revised 18 August, 2023

Table of Contents

Audit overview

The Project

In July 2023, Informal Systems has conducted a security audit for Celestia of the blob module within celastia-app. The focus of the audit was on creation and processing of blobs and PFB (Pay For Blobs) transactions.

The audited commit hash is 341006e5.

The audit took place from July 3, 2023 through July 28, 2023 by the following personnel:

- Andrija Mitrovic
- Ivan Gavran

Conclusions

We performed a thorough code review of the `celestia-app` codebase and found it to be implemented very carefully, equipped with many defensive checks, detailed documentation, and tests for each functionality. In our audit, we report the total of 9 findings, one of them of `medium` severity and the rest `informational`.

Disclaimer

Audit dashboard

Target Summary

- **Type**: Specification and Implementation
- **Platform**: Golang
- **Artifacts**: celestia-app

Engagement Summary

- **Dates**: 03.07.2023 to 28.07.2023
- **Method**: Manual code review, protocol analysis
- **Employees Engaged**: 2

Severity Summary

Finding Severity	#
Critical	0
High	0
Medium	1
Low	0
Informational	8
Total	**9**

System overview

Celestia (`celestia-app`) is a minimal blockchain that only accepts messages posting blobs onto the data availability network and paying for that service. The key property is that the posted data will always be paid for and that it will always be possible to verify that the data blob was indeed included. This property needs to be maintained while treating light clients as first-class citizens and never requiring space or computational resources that exceed light-clients' capabilities.

A typical use-case for the Celestia network is to serve as a layer 1 for different rollups, which post their transactions to Celestia. Celestia will make sure that the received transactions are published, but will not try to interpret them.

Transaction Lifecycle

Each Celestia transaction consists of a `MsgPayForBlobs` message and the blobs the message is paying for. Each blob is associated with a namespace ID. Different IDs signify different sources of transactions (for instance, different rollups) and can be used to get proofs of existence or non-existence of namespaced data in a block.

The `MsgPayForBlobs` contains a commitment to each of the blobs it is posting. Each transaction is validated for basic blob validity rules (including the correctness of the commitments) before being added to the mempool. This commitment allows for checking that the blob that was paid for was actually included in the block.

The proposer of the block splits transactions (ordered by the fee paid) in shares, which it then arranges into a datasquare. The datasquare is then extended with parity data, from which row and column headers hashes, and the final block hash, are calculated. (Without entering into details of the extended datasquare construction, let us just note that this is needed for data availability sampling.) Validators of the block go independently though the datasquare construction process and accept the proposal only when convinced that the proposed hash matches the one they generated.

Finally, once the consensus on the block is reached, the blob data is considered published and Celestia light nodes can verify that it was indeed published by sampling storage nodes.

Identified Threats and Audit Plan

While Celestia is a very simple chain in terms of the transactions it accepts and how its internal state is evolving, it relies on many subtle calculations to function correctly. Thus, in this audit, we focus on inspecting the implementation of those key parts:

- `PrepareProposal` and `ProcessProposal` functions
- `CreateCommitment` function
- Construction of the datasquare
- Validity rules for `BlobTx`
- Splitting blobs into shares and creation of the commitments
- Inspection of the existing tests and their functionality

Studying the documentation and establishing the correspondence between the documentation and the implementation, we validated the following desired properties:

- Whenever a PFB message is included, all the blobs it contains are included in the same block, with correct commitments.
- Every PFB passes basic blob validation.
- Data square is created using the prescribed set of rules, which enable reconstruction of the whole square from partial data.

Findings

Title	Severity	Status
Iteration over all transactions when building the datasquare	2 MEDIUM	ACKNOWLEDGED
Blob transactions missing from the square size test	0 INFORMATIONAL	ACKNOWLEDGED
The test fuzz_abci_test.go passes trivially	0 INFORMATIONAL	RESOLVED
Unnecessarily verbose implementation of roundUpByMultipleOf	0 INFORMATIONAL	RESOLVED
Namespace calculation inside a loop	0 INFORMATIONAL	RESOLVED
When building padding shares, a ShareVersionZero is used instead of the share version of the blob.	0 INFORMATIONAL	RESOLVED
Different, non-equal implementations of ValidateBlobNamespace, with a non-future-proof check	0 INFORMATIONAL	RESOLVED
Strict inequality missing in the documentation on the namespace validity rules	0 INFORMATIONAL	RESOLVED
Slightly confusing documentation on shares creation	0 INFORMATIONAL	RESOLVED

Iteration over all transactions when building the datasquare

Title	Iteration over all transactions when building the datasquare
Project	Celestia: Q3 2023
Type	IMPLEMENTATION
Severity	2 MEDIUM
Impact	2 MEDIUM
Exploitability	2 MEDIUM
Issue	https://github.com/celestiaorg/celestia-app/issues/2144

Involved artifacts

- celestia-app/pkg/square/square.go

Description

Inside the Build function, there is an iteration over all transactions. (The `Build` function is called inside `PrepareProposal` after filtering out the transactions that don't pass antehandlers.)
It will always iterate over all transactions `txs`, even if there is no chance any of them fits into the square.

Problem Scenarios

Depending on the number of transactions, this can create delays, possibly creating further rounds in the consensus (but not jeopardizing liveness, since the timeout value is dynamically increased).

This scenario is only a problem if there is a large number of transactions present in the mempool. An adversarial DOS attack is possible, but at a significant cost for the attacker (in that scenario the attacker would have to create and pay for a large number of transactions).

Recommendation

A further investigation is needed. The straight-forward solution, waiting for the square to be completely full, is maybe not the best thing (as that will happen only very rarely).

The tradeoff that needs to be explored is between:

a) leaving everything as-is, iterating over all transactions;

b) doing some pre-processing to have better information of when no new transactions could fit in without iterating through the whole thing; and

c) using some heuristic to finalize the square even before it is completely full.

Alternatively, one could limit the number of transactions passed to `PrepareProposal`.

Status

Acknowledged.

Blob transactions missing from the square size test

Title	Blob transactions missing from the square size test
Project	Celestia: Q3 2023
Type	**IMPLEMENTATION**
Severity	**0 INFORMATIONAL**
Impact	**1 LOW**
Exploitability	**0 NONE**
Issue	https://github.com/celestiaorg/celestia-app/issues/2034

Involved artifacts

- celestia-app/app/test/square_size_test.go

Description

In the `square_size_test`, none of the `blobTx` transactions ever reaches `PrepareProposal`. The root of the problem seems to be in this check inside `UnmarshalBlobTx`.

Problem Scenarios

This filtering out renders the test useless, creating false sense of confidence.

Recommendation

Make sure that the valid transactions indeed make it into a block.

Status

Acknowledged.

The test fuzz_abci_test.go passes trivially

Involved artifacts

Title	The test fuzz_abci_test.go passes trivially
Project	Celestia: Q3 2023
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	1 LOW
Exploitability	0 NONE
Issue	https://github.com/celestiaorg/celestia-app/issues/2027

Involved artifacts

- celestia-app/app/test/fuzz_abci_test.go

Description

The test `celestia-app/app/test/fuzz_abci_test.go` passes in checking that every proposed block is accepted, but trivially: all the transactions get filtered out. The failure happens on the antehandler VerifySignature.

Problem Scenarios

This filtering out renders the test useless, creating false sense of confidence.

Recommendation

Make sure that the valid transactions indeed make it into a block.

Status

Resolved.

Unnecessarily verbose implementation of roundUpByMultipleOf

Title	Unnecessarily verbose implementation of roundUpByMultipleOf
Project	Celestia: Q3 2023
Type	PRACTICE
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Issue	https://github.com/celestiaorg/celestia-app/issues/2175

Involved artifacts

- celestia-app/pkg/shares/blob_share_commitment_rules.go

Description

This check within `roundUpByMultipleOf` function, which treats `cursor == 0` and `cursor%v == 0` as two distinct cases can be summarized into a single `cursor % v == 0` case (because `0 % i` is always 0, for any `i`).

Recommendation

We suggest simplifying the code by considering only two cases (`cursor` divisible by `v` and every other case).

Status

Resolved.

Namespace calculation inside a loop

Title	Namespace calculation inside a loop
Project	Celestia: Q3 2023
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Issue	https://github.com/celestiaorg/celestia-app/issues/2110

Involved artifacts

- celestia-app/x/blob/types/payforblob.go

Description

In the `CreateCommitment` function, there is a loop over all `leafSets`, corresponding to mountains in the Merkle mountain range. A new namespace is created within that loop (line 210), which re-creates the same namespace over and over again.

Problem Scenarios

This is a wasteful computation (even if there are no severe consequences stemming from it, since the number of mountains is limited).

Recommendation

Move the creation of namespace before the loop body.

Status

Resolved.

When building padding shares, a ShareVersionZero is used instead of the share version of the blob

Title	When building padding shares, a ShareVersionZero is used instead of the share version of the blob
Project	Celestia: Q3 2023
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Issue	https://github.com/celestiaorg/celestia-app/issues/2142

Involved artifacts

- celestia-app/x/blob/types/payforblob.go

Description

Builder for building padding shares is created using the constant `ShareVersionZero` share version. This happens within function `NamespacePaddingShare` that is called from `Export` when building padding shares (WriteNamespacePaddingShares->NamespacePaddingShares->NamespacePaddingShare->NewBuilder).

Conversely, the non-padding shares within `Export` function are created ((Write->NewBuilder(blobNamespace, blob.ShareVersion, true).Init())) with shares Builder that takes `blob.ShareVersion`.

Problem Scenarios

At present, when there is only `ShareVersionZero`, this is not a problem. However, with the introduction of new share versions, it is possible that the padding could be created using a different share version than the preceding shares, potentially leading to inconsistencies

Recommendation

Passing `blob.ShareVersion` to `NamespacePaddingShare` and using it in the `NewBuilder` constructor should resolve the problem.

Status

Resolved.

Different, non-equal implementations of ValidateBlobNamespace, with a non-future-proof check

Title	Different, non-equal implementations of ValidateBlobNamespace, with a non-future-proof check
Project	Celestia: Q3 2023
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Issue	https://github.com/celestiaorg/celestia-app/issues/2143

Involved artifacts

- celestia-app/pkg/namespace/namespace.go
- celestia-app/x/blob/types/payforblob.go

Description

There is some confusion in the implementation of blob namespace validation:

- there are two functions called `ValidateBlobNamespace` : one (A) is a member function of the `Namespace` structure, and the other (B) takes `Namespace` as an argument. The two functions are almost identical, the only difference being that B additionally checks if the namespace version is `NamespaceVersionZero` .
- Interestingly, both functions are used in a similar context: A is called from within `ValidateBlobTx` (which is called both from `ProcessProposal` and `CheckTx`). B is called from within `ValidateBasic` of `MsgPayForBlobs` , which is also called from `ValidateBlobTx` .

Finally, a problem of future-proofing: it would make sense to replace the explicit check for version 0,

```
if ns.Version != appns.NamespaceVersionZero {
        return ErrInvalidNamespaceVersion
    }
```

with a check for whether the `ns.Version` is among the supported user-namespace versions. (Along the lines of how it was done for share versions, here.)

There is another function which validates namespace versions, ValidateVersion. In this validation, both zero-namespace and max-namespace are allowed (so, it is checking for any possible versions, and not just user-created blobs).

Problem Scenarios

Duplicated functions may pose a problem maintaining the codebase.

Recommendation

We suggest keeping only the function B and replace the explicit check for `NamespaceVersionZero` with a check of inclusion in the set of supported namespace versions.

Furthermore, it would probably make sense to change the name of the function `ValidateVersion` into something that emphasises that any supported version is allowed, e.g., `ValidateVersionSupported`.

Status

Resolved.

Strict inequality missing in the documentation on the namespace validity rules

Title	Strict inequality missing in the documentation on the namespace validity rules
Project	Celestia: Q3 2023
Type	DOCUMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Issue	https://github.com/celestiaorg/celestia-app/issues/2178

Involved artifacts

- celestia-app/x/blob/README.md

Description

The documentation about namespace validity rules states that "The namespace is not lexicographically less than the `MAX_RESERVED_NAMESPACE` range", but in reality it needs to be strictly greater than `MAX_RESERVED_NAMESPACE`.

Recommendation

We suggest rephrasing the sentence into "The namespace is lexicographically greater than the `MAX_RESERVED_NAMESPACE` range."

Status

Resolved.

Slightly confusing documentation on shares creation

Title	Slightly confusing documentation on shares creation
Project	Celestia: Q3 2023
Type	DOCUMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Issue	https://github.com/celestiaorg/celestia-app/issues/2176

Involved artifacts

- celestia-app/specs/src/specs/data_structures.md

Description

In the section *Arranging Available Data Into Shares* the process of constructing shares is described. Immediately afterwards there is a sentence: "*Note that by construction, each share only has a single namespace, and that the list of concatenated shares is lexicographically ordered by namespace ID.*" This property, however, does not follow from the described construction (but is obtained independently of it by sorting the blobs in the order of namespace).

Recommendation

Keep the emphasised property, but avoid the reference to *by construction,* as it may be confusing.

Status

Resolved.

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:

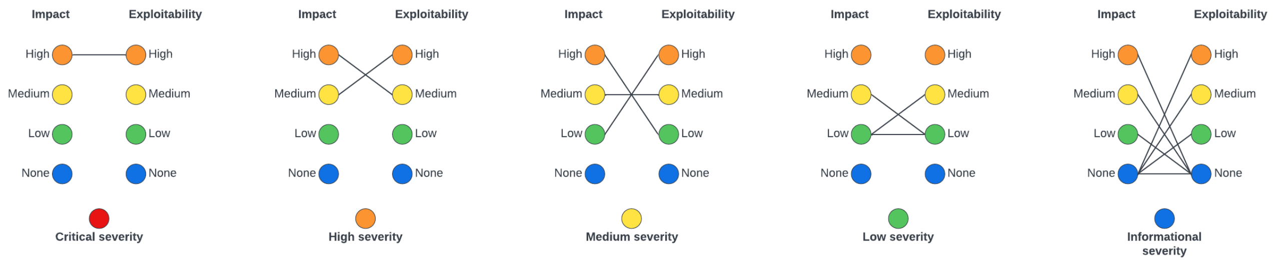
- *Actors* can be any entity that interacts with the system: other blockchains, system users, validators, relayers, but also uncontrollable phenomena (e.g. network delays or partitions).
- *Actions* can be

  - *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