



Celestia Shwap

Security Assessment

April 24th, 2025 — Prepared by OtterSec

James Wang

james.wang@osec.io

Table of Contents

Executive Summary	2
Overview	2
Key Findings	2
Scope	2
General Findings	3
OS-CSP-SUG-00 Code Maturity	4
Appendices	
Vulnerability Rating Scale	5
Procedure	6

01 — Executive Summary

Overview

Celestia Labs engaged OtterSec to assess the `shwap` program. This assessment was conducted between March 25th and April 10th, 2025. For more information on our auditing methodology, refer to [Appendix B](#).

Key Findings

We produced 1 finding throughout this audit engagement.

We made recommendations to emphasizing the need to adhere to coding best practices ([OS-CSP-SUG-00](#)).

Scope

The source code was delivered to us in a Git repository at <https://github.com/celestiaorg/celestia-node>. This audit was performed against [568744f](#).

A brief description of the programs is as follows:

Name	Description
shwap	It handles serialization and organization of erasure-coded data shares in a namespace-aware format.

02 — General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Description
OS-CSP-SUG-00	Suggestions regarding ensuring adherence to coding best practices.

Code Maturity

OS-CSP-SUG-00

Description

1. The `SetTree` method is marked as test-only in `celestia-app`, but it is utilized in production within `RowNamespaceDataFromShares` to inject a custom-configured NMT. This contradicts its intended utilization and introduces a future maintenance risk if it is altered or removed under the assumption that it is only for testing. To avoid accidental breakage and clarify its role, remove the test-only comment.

```
>_ pkg/wrapper/nmt_wrapper.go
```

GO

```
// SetTree sets the underlying tree to the provided tree. This is used for  
// testing purposes only.  
func (w *ErasedNamespacedMerkleTree) SetTree(tree Tree) {  
    w.tree = tree  
}
```

2. Ensure that `NamespaceDataID` is properly documented, as it is currently missing from `CIP-19`.

Remediation

Implement the above-mentioned suggestions.

A — Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the [General Findings](#).

CRITICAL

Vulnerabilities that immediately result in a loss of user funds with minimal preconditions.

Examples:

- Misconfigured authority or access control validation.
 - Improperly designed economic incentives leading to loss of funds.
-

HIGH

Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit.

Examples:

- Loss of funds requiring specific victim interactions.
 - Exploitation involving high capital requirement with respect to payout.
-

MEDIUM

Vulnerabilities that may result in denial of service scenarios or degraded usability.

Examples:

- Computational limit exhaustion through malicious input.
 - Forced exceptions in the normal user flow.
-

LOW

Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.
-

INFO

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants.
 - Improved input validation.
-

B — Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that others may have missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.