Celestia Blobstream X

Security Assessment

March 9th, 2024 — Prepared by OtterSec

James Wang james.wang@osec.io

Robert Chen r@osec.io

Table of Contents

Executive Summary **2**

Overview 2

Key Findings 2

Scope 2

Findings **3**

General Findings **4**

OS-BLB-SUG-00 | Explicit Check For Zero Hash Results 5

OS-BLB-SUG-01 | Check Against Non Existing Nonce 6

Appendices

Vulnerability Rating Scale **7**

Procedure **8**

© 2024 Otter Audits LLC. All Rights Reserved.

01 — Executive Summary

Overview

Celestia engaged OtterSec to assess the `FunctionRegistry`, `SuccinctGateway`, `TimelockUpgradeable`, and `BlobstreamX` programs. This assessment was conducted between February 20th and February 29th, 2024. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 2 findings throughout this audit engagement.

We recommend addressing a potential edge case by asserting that the nonce used to index state data commitments is greater than zero. This precautionary measure aims to prevent the utilization of an uninitialized entry as the proof root (OS-BLB-SUG-01). Additionally, we highlighted the absence of a check to disallow the usage of the function ID when it equals the default value (OS-BLB-SUG-00).

Scope

The source code was delivered to us in a Git repository at https://github.com/succinctlabs/blobstreamx. This audit was performed against commit 2afc6ba.

A brief description of the programs is as follows:

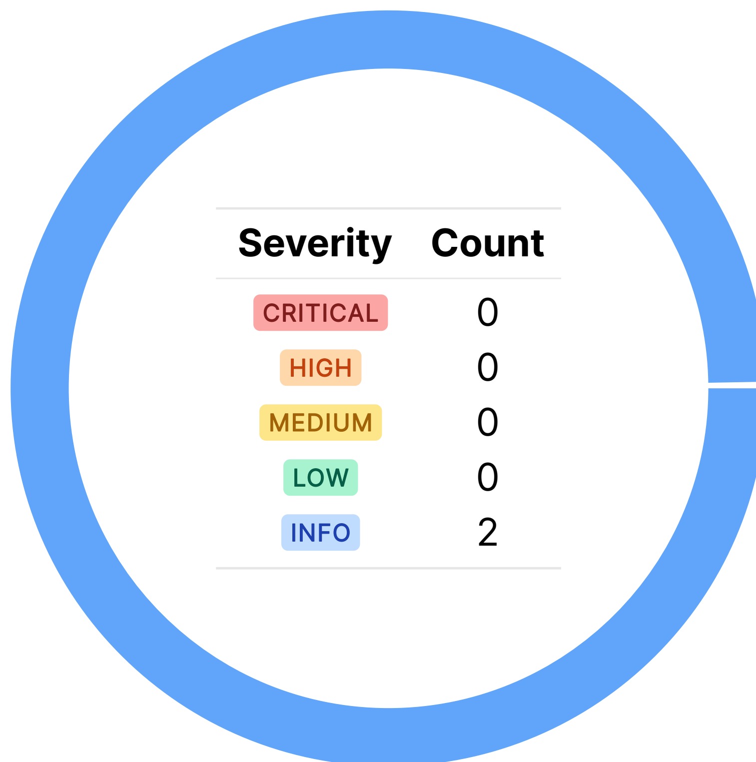
Name	Description
FunctionRegistry	Manages zero knowledge (zk) verifier contract registration to allow users to verify the proofs on-chain safely.
SuccinctGateway	A gateway contract that allows users to request for output or zkproof generation.
TimelockUpgrade-able	Contains the timelocked upgrade contract.
BlobstreamX	Celestia's new blobstream contract that uses Succinctx's zkproof service to validate data commitments before including them in the Layer 1 state.

© 2024 Otter Audits LLC. All Rights Reserved.

02 — Findings

Overall, we reported 2 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



© 2024 Otter Audits LLC. All Rights Reserved.

03 — 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-BLB-SUG-00	The code base lacks a check to disallow the usage of `_functionId` when it is equal to `bytes32(0)`.
OS-BLB-SUG-01	There is a potential edge case where the `_proofNonce` used to index `state_dataCommitments` should be asserted to be greater than zero to prevent using an uninitialized entry as the proof root.

© 2024 Otter Audits LLC. All Rights Reserved.

Explicit Check For Zero Hash Results

OS-BLB-SUG-00

Description

The codebase utilizes `_functionId` in several places, including in guards against reentrancy in the `nonReentrant` modifier and indexing into `allowedProvers` within the `onlyProver` modifier. The program implicitly assumes that `_functionId` is never equal to `bytes32(0)` throughout the entire codebase, based on the program calculating it by taking the hash of `msg.sender` and a salt value in `getFunctionId`.

>_ *FunctionRegistry.sol* solidity

```solidity
function getFunctionId(address _owner, bytes32 _salt)
    public
    pure
    override
    returns (bytes32 functionId)
{
    functionId = keccak256(abi.encode(_owner, _salt));
}
```

While this assumption may seem reasonable given the hashing process, it is advisable to add an explicit check to prevent the possibility of collisions where `_functionId` may be equal to `bytes32(0)`.

Remediation

Add an explicit check to ensure that `_functionId` is not equal to `bytes32(0)` before using it in the relevant parts of the code.

Patch

After careful consideration, the Succinct team concluded that the possibility of constructing a hash where `_functionId = bytes32(0)` is exceedingly low. A fix is deemed unnecessary and the risk is accepted.

© 2024 Otter Audits LLC. All Rights Reserved.

Check Against Non Existing Nonce OS-BLB-SUG-01

Description

In `BlobstreamX`, `state_proofNonce` is initialized to one within `initialize`. However, since the `_proofNonce` indexes `state_dataCommitments` in `verifyAttestation`, having `_proofNonce` as zero may result in using the uninitialized default value (`bytes32(0)`) as the Merkle tree root. In practice, this may not be exploitable due to the unlikeliness of constructing a valid Merkle tree with a root equal to `bytes32(0)` that includes the desired data.

>_ BlobstreamX.sol solidity

```solidity
function verifyAttestation(
    uint256 _proofNonce,
    DataRootTuple memory _tuple,
    BinaryMerkleProof memory _proof
) external view returns (bool) {
    if (frozen) {
        revert ContractFrozen();
    }
    if (_proofNonce >= state_proofNonce) {
        return false;
    }
    [...]
}
```

Remediation

Check `_proofNonce` is greater than zero to assert that `_proofNonce` is a valid index.

Patch

Resolved in c36881d.

© 2024 Otter Audits LLC. All Rights Reserved.

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.

© 2024 Otter Audits LLC. All Rights Reserved.

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 the other 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.

© 2024 Otter Audits LLC. All Rights Reserved.