Celestia

Audit

Presented by:

OtterSec contact@osec.io

James Wang james.wang@osec.io

Robert Chen r@osec.io

Contents

01 | Executive Summary

Overview

Celestia engaged OtterSec to perform an assessment of the `celestia-app` and `blobstream-contracts` programs. This assessment was conducted between October 17th and November 16th, 2023. For more information on our auditing methodology, see Appendix B.

Key Findings

Over the course of this audit engagement, we produced 8 findings in total.

In particular, we discovered several vulnerabilities, including an invariant violation in which the precondition for the function responsible for retrieving the path length from the key is not fulfilled in a specific case, resulting in a revert (OS-CLST-ADV-00). Another issue is related to incorrect validation during Merkle proof verification (OS-CLST-ADV-01). We further highlighted a stack overflow scenario due to utilizing recursive calls (OS-CLST-SUG-00).

We also recommended removing unused functions (OS-CLST-SUG-04) and modifying the code to adhere to coding best practices (OS-CLST-SUG-05).

02 | Scope

The source code was delivered to us in a git repository at github.com/celestiaorg/celestia-app/tree/main. This audit was performed against v3.1.0 and v1.3.0.

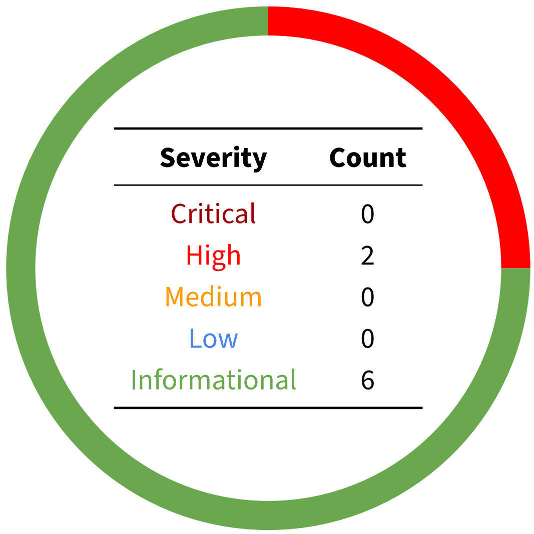
A brief description of the programs is as follows:

Name	Description
celestia-app	A blockchain application built utilizing parts of the Cosmos stack that implement the blobstream state machine, which creates attestations for EVM chains. The attestations are signed by orchestrators and submitted by relayers.
blobstream-contracts	Enables the relay of Celestia block header data roots to an EVM chain in one direction. It does not directly bridge assets such as fungible or non-fungible tokens, and it is unable to send messages from the EVM chain back to Celestia.

03 | Findings

Overall, we reported 8 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.



Severity	Count
Critical	0
High	2
Medium	0
Low	0
Informational	6

04 | Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-CLST-ADV-00	High	Resolved	The `numLeaves > 1` pre-condition for `pathLengthFromKey` is violated when `numLeaves` is not a power of two.
OS-CLST-ADV-01	High	Resolved	`verifyInner` performs incorrect validations during Merkle proof verification.

OS-CLST-ADV-00 [high] | Invariant Violation

Description

`pathLengthFromKey` calculates the path length from the root to a specific leaf in a Merkle tree. It considers the leaf's position in the left or right subtree, utilizing the starting bit, and recursively calculates the path length. However, there is a precondition violation in the function, where `numLeaves` should always be greater than one.

Utils.sol SOLIDITY

```solidity
/// @notice Calculate the length of the path to a leaf
/// @param key: The key of the leaf
/// @param numLeaves: The total number of leaves in the tree
/// @return pathLength : The length of the path to the leaf
/// @dev A precondition to this function is that `numLeaves > 1`, so that
    ↪  `(pathLength - 1)` does not cause an underflow when pathLength = 0.
function pathLengthFromKey(uint256 key, uint256 numLeaves) pure returns (uint256
    ↪  pathLength) {
    // Get the height of the left subtree. This is equal to the offset of the
        ↪  starting bit of the path
    pathLength = Constants.MAX_HEIGHT - getStartingBit(numLeaves);
    // Determine the number of leaves in the left subtree
    uint256 numLeavesLeftSubTree = (1 << (pathLength - 1));
    [...]
    else {
        return 1 + pathLengthFromKey(key - numLeavesLeftSubTree, numLeaves -
            ↪  numLeavesLeftSubTree);
    }
}
```

This situation arises when `numLeaves` is not a power of two, resulting in an integer underflow in the recursive calculation of the path length. In `pathLengthFromKey`, the call to `getStartingBit` dynamically determines the starting bit of the path based on the total number of leaves in the Merkle tree. Consequently, if `numLeaves` is not a power of two, it eventually becomes one, violating the precondition. Subsequently, in the recursive call, `numLeaves` changes to zero due to the subtraction of `numLeaves` from `numLeavesLeftSubTree`.

Utils.sol SOLIDITY

```solidity
function getStartingBit(uint256 numLeaves) pure returns (uint256 startingBit) {
    startingBit = 0;
    while ((1 << startingBit) < numLeaves) {
        startingBit += 1;
    }
    return Constants.MAX_HEIGHT - startingBit;
}
```


The value of `numLeaves` is utilized in determining the `StartingBit` within `getStartingBit`, where the initial value of `StartingBit` is set to zero. Since both `numLeaves` and `StartingBit` are zero, the loop condition in `getStartingBit` is never met, and `StartingBit` remains unaltered. Consequently, `getStartingBit` returns zero, resulting in `pathLength` being zero. Therefore, when calculating `numLeavesLeftSubTree`, `pathLengthFromKey` subtracts one from `pathLength` (which is zero), resulting in an integer underflow.

Proof of Concept

1. `pathLengthFromKey` is called with `numLeaves` = 3 and `key` = 3.
2. The while loop in `getStartingBit` iterates two times, incrementing `StartingBit` to two, before the loop condition fails.
3. Thus, `pathLength` becomes two and consequently `numLeavesLeftSubTree` becomes two.
4. The program executes the first recursive call where `numLeaves` becomes one (due to the subtraction with `numLeavesLeftSubTree`).
5. This time, the while loop in `getStartingBit` does not iterate even once as the loop condition fails on the first iteration itself and returns zero.
6. Now, `pathLength` becomes zero, and while deriving `numLeavesLeftSubTree`, `pathLength` is subtracted by one. Since `pathLength` is zero, underflow occurs, and execution stops.

Remediation

Ensure that `numLeaves` is always a power of two when calling this function.

Patch

Resolved in fff73c2.

OS-CLST-ADV-01 [high] | Incorrect Validations

Description

`verifyInner` verifies the inclusion of an inner node (non-leaf) in a Merkle tree. However, there is an issue related to comparing the key's position within the subtree. The current condition incorrectly compares `(proof.key - subTreeStartIndex)` to `1 << (height - heightOffset - 1)`. The intent is to check whether `proof.key` is in the subtree's first or second half. Thus, instead of comparing against the midpoint of the subtree, it compares against the difference between the height and height offset, compromising the integrity of the Merkle proof verification process.

NamespaceMerkleTree.sol SOLIDITY

```solidity
function verifyInner(
[...]
) internal pure returns (bool) {
    [...]
    while (true) {
        [...]
        // Determine if the key is in the first or the second half of
        // the subtree.
        if (proof.key - subTreeStartIndex < (1 << (height - heightOffset - 1))) {
            node = nodeDigest(node, proof.sideNodes[height - heightOffset - 1]);
        }
        [...]
    }
    [...]
}
```

Moreover, an additional erroneous condition assesses whether sufficient side nodes exist in the proof for verification. This condition pertains to comparing the length of `proof.sideNodes`, where `proof.sideNodes` is compared to `height - 1` instead of `height - heightOffset - 1`, accurately representing the minimum required number of side nodes for the current height.

NamespaceMerkleTree.sol SOLIDITY

```solidity
function verifyInner(
[...]
) internal pure returns (bool) {
    [...]
    if (stableEnd != proof.numLeaves - 1) {
        if (proof.sideNodes.length <= height - 1) {
            return false;
        }
    }
    [...]
}
```

All production code at the time of this audit calls `verifyInner` with `startingHeight = 1`, so there are no immediate impacts. However, we still recommend fixing it to prevent future issues.

Remediation

Adjust `1 << (height - heightOffset - 1)` to `1 << (height - 1)` in the initial condition, and for the subsequent condition, modify the comparison to check against `height - heightOffset - 1` instead of `height - 1`.

Patch

Resolved in 86cbb51.

05 | 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-CLST-SUG-00	`_computeRoot` may surpass the Ethereum virtual machine's stack limit due to the recursive calls.
OS-CLST-SUG-01	Pruning of old `DataCommitments` due to an extended block may recreate data commitments for past windows.
OS-CLST-SUG-02	`verifyMultiHashes` may encounter an overflow issue when estimating the leaf size of the subtree containing the proof range.
OS-CLST-SUG-03	`GetDataCommitmentForHeight` utilizes an incorrect operator.
OS-CLST-SUG-04	Removal of unutilized code to improve code readability.
OS-CLST-SUG-05	Suggestions regarding best coding practices.

OS-CLST-SUG-00 | Overflow Of EVM Stack

Description

The recursive structure of `_computeRoot` presents a potential risk of exhausting the stack on the Ethereum virtual machine. A limited stack size constrains the Ethereum virtual machine, and each recursive invocation consumes a specific portion of the stack space. In scenarios where the recursion depth is substantial, mainly when the recursive function utilizes a significant number of local variables, as in `_computeRoot`, this may surpass the Ethereum virtual machine's stack limit, resulting in transaction failure.

NamespaceMerkleTree.sol SOLIDITY

```solidity
function _computeRoot(
    NamespaceMerkleMultiproof memory proof,
    NamespaceNode[] memory leafNodes,
    uint256 begin,
    uint256 end,
    uint256 headProof,
    uint256 headLeaves
) private pure returns (NamespaceNode memory, uint256, uint256, bool) {
    [...]
    // Recursively get left and right subtree
    uint256 k = _getSplitPoint(end - begin);
    (NamespaceNode memory left, uint256 newHeadProofLeft, uint256
        ↪ newHeadLeavesLeft,) =
        _computeRoot(proof, leafNodes, begin, begin + k, headProof, headLeaves);
    (NamespaceNode memory right, uint256 newHeadProof, uint256 newHeadLeaves, bool
        ↪ rightIsNil) =
        _computeRoot(proof, leafNodes, begin + k, end, newHeadProofLeft,
            ↪ newHeadLeavesLeft);
   [...]
}
```

It is important to note that this limit may differ among various Ethereum virtual machine implementations or network setups. Consequently, opting for loop-based structures over deep recursion offers a better solution to reduce stack usage. Furthermore, it is worth acknowledging that a comparable stack exhaustion scenario may manifest in `pathLengthFromKey` due to its recursive nature. However, given its utilization of significantly fewer local variables, such a scenario is practically non-existent.

Remediation

Re-implement the logic with a loop instead of recursive calls to reduce stack utilization.

Patch

Celestia's team decided to address this issue later due to the low probability of it occurring.

OS-CLST-SUG-01 | Recreation Of Pruned Data Commitments

Description

The issue concerns a potential edge case where the program prunes all past `DataCommitments` due to the block being halted or stalled for an extended period. In such a circumstance, if the `else` branch in `NextDataCommitment` is activated, it may regenerate data commitments for previous windows. While this occurrence is deemed improbable, it may manifest if generating fewer than `DataCommitmentWindow` commitments within `AttestationExpiryTime`.

keeper/keeper_data_commitment.go GO

```go
// NextDataCommitment returns the next data commitment that can be written to
// state.
func (k Keeper) NextDataCommitment(ctx sdk.Context) (types.DataCommitment, error) {
    [...]
        else {
                // only for the first data commitment range, which is: [1, data
                    ↪ commitment window + 1)
                beginBlock = 1
                endBlock = dcWindow + 1
        }

        dataCommitment := types.NewDataCommitment(nonce, beginBlock, endBlock,
            ↪ ctx.BlockTime())
        return *dataCommitment, nil
}
```

Remediation

Verify that the recreated commitment does not overlap with existing commitments or that the program created it within a reasonable time frame from the current block time.

Patch

Celestia's team decided to address this issue later due to the low probability of it occurring.

OS-CLST-SUG-02 | Integer Overflow

Description

There is a potential for overflow when calculating `proofRangeSubtreeEstimate` in `verifyMultiHashes`. `verifyMultiHashes` calls `_getSplitPoint` internally to calculate the split point.

NamespaceMerkleTree.so SOLIDITY

```solidity
function verifyMultiHashes(
    NamespaceNode memory root,
    NamespaceMerkleMultiproof memory proof,
    NamespaceNode[] memory leafNodes
) internal pure returns (bool) {
    [...]
    // estimate the leaf size of the subtree containing the proof range
    uint256 proofRangeSubtreeEstimate = _getSplitPoint(proof.endKey) * 2;
    if (proofRangeSubtreeEstimate < 1) {
        proofRangeSubtreeEstimate = 1;
    }
    [...]
}
```

If `proof.endKey` is near the maximum value of a `uint256` ($2^{256} - 1$), calculating `_getSplitPoint(proof.endKey) * 2` may result in an overflow. This occurs as the product of `_getSplitPoint(proof.endKey)`, and two may exceed the maximum value of a `uint256`. However, this scenario is highly impractical since getting as many nodes in a tree to trigger this overflow is practically impossible.

Remediation

Ensure that the developers are aware of such a possibility.

Patch

Celestia's team decided to address this issue later due to the low probability of it occurring.

OS-CLST-SUG-03 | Faulty Range Check

Description

In `GetDataCommitmentForHeight`, the condition: `if latestDC.EndBlock < height` compares the end block of the latest data commitment (`latestDC.EndBlock`) with the provided height. The intent is to check if the provided height falls within the range of the latest data commitment, where the range is `[BeginBlock, EndBlock)`.

keeper/keeper_data_commitment.go GO

```go
// GetDataCommitmentForHeight returns the attestation containing the provided
    ↪  height.
func (k Keeper) GetDataCommitmentForHeight(ctx sdk.Context, height uint64)
    ↪  (types.DataCommitment, error) {
    [...]
        if latestDC.EndBlock < height {
                return types.DataCommitment{}, errors.Wrap(
                        types.ErrDataCommitmentNotGenerated,
                        fmt.Sprintf(
                                "Latest height %d < %d",
                                latestDC.EndBlock,
                                height,
                        ),
                )
        }
    [...]
}
```

Thus, a problem occurs when the provided height equals the latest data commitment's `EndBlock`. In this instance, the height would be considered part of the range, which is incorrect given the range definition.

Remediation

Utilize the <= operator in the condition, ensuring it is still part of the range when the provided height equals `EndBlock`.

Patch

Celestia's team decided to address this issue later due to the low impact.

OS-CLST-SUG-04 | Unused Functions

Description

The following functions are unused and removing them improves readability:

1. `NamespaceMerkleTree._nextSubtreeSize`.
2. `NamespaceMerkleTree._bitsTrailingZeroes`.
3. `keys.ConvertByteArrToString`.

Remediation

Remove the unutilized functions.

OS-CLST-SUG-05 | Code Maturity

Description

1. The comment in `_getSplitPoint` mentions that ”x is always an unsigned int * 2,” but this statement is incorrect, as the program may it with `x` equal to `1`, as observed in `verifyMultiHashes`.

   NamespaceMerkleTree.sol SOLIDITY

   ```solidity
   function _getSplitPoint(uint256 x) private pure returns (uint256) {
       // Note: since `x` is always an unsigned int * 2, the only way for this
       // to be violated is if the input == 0. Since the input is the end
       // index exclusive, an input of 0 is guaranteed to be invalid (it would
       // be a proof of inclusion of nothing, which is vacuous).
       [...]
   }
   ```

2. The utilization of `params` is deprecated, as indicated here. Store `params` directly in the module store instead.
3. The comment in `GetCurrentValset` indicates the scenario where a validator does not have an associated EVM address should never occur and is considered an indication of a potential vulnerability. However, there is an attempt to recover from this situation by deriving a default Ethereum address for the validator. Hence, it is advisable to check `IsEVMAddressUnique` before `SetEVMAddress` to ensure that the Ethereum address is globally unique.

Remediation

Implement the suggestions mentioned above.

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 section.

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.
Informational	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 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.