certora

Security Assessment Report

Eigenlayer - Merkle audit

July 2025

Prepared for Eigenlayer

Table of contents

Project Summary

Project Scope

Project Name	Repository (link)	Audited Commits	Platform
Eigenlayer Merkle	https://github.com/Layr-Labs/eigenlayer-contracts	054b311	EVM

Project Overview

This document describes the manual code review findings for **Eigenlayer**. The work was undertaken from **July 22nd** to **July 29th 2025**.

The following contract list is included in our scope:

- `src/contracts/libraries/Merkle.sol`

The team performed a manual audit of all the Solidity contracts in scope**.** During the manual audit, the Certora team discovered bugs in the Solidity contracts code, as listed on the following page.

Protocol Overview

The scope of this audit covers the Merkle proof verification library developed by EigenLayer. This library is responsible for constructing Merkle trees, generating verification proofs, and validating leaf nodes against a specified tree root. It includes support for both the Keccak256 and SHA-256 hashing algorithms, allowing it to serve a variety of cryptographic use cases.

The audit aimed to assess the correctness and security of the library both as it is integrated within the EigenLayer contracts and as a standalone utility intended for external adoption. Given that the library is designed for use by third parties, particular attention was paid to identifying both actual and theoretical issues that could emerge from different usage patterns. This approach reflects the broader implications such a library may have when adopted across a wide range of applications.

Findings Summary

The table below summarizes the findings of the review, including type and severity details.

Severity	Discovered	Confirmed	Fixed
Critical	-	-	-
High	-	-	-
Medium	-	-	-
Low	5	5	2
Informational	15	15	13
Total	20	20	15

Severity Matrix

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
		Likelihood		

Detailed Findings

ID	Title	Severity	Status
L-01	Prevent uninitialized proofs from being used	Low	Fix Confirmed
L-02	merkleizeKeccak() will contain valid 0 leaves in most cases	Low	Acknowledged
L-03	Lack of validation on `leaves.length` is a power of two in merkleizeSha256()	Low	Fix Confirmed
L-04	Intermediate node hashes can be used as leaves in proofs	Low	Acknowledged
L-05	Keccak functions allow one-leaf trees, but SHA-256 functions don't	Low	Acknowledged

Low Severity Issues

L-01 Prevent uninitialized proofs from being used		
Severity: **Low**	Impact: **High**	Likelihood: **Low**
Files: Merkle.sol	Status: Fix Confirmed	

Description: Currently, the functions `verifyInclusionKeccak()` and `verifyInclusionSha256()` are meant to validate that a provided proof & leaf are valid against a root. However, no validation is made if the provided root is an empty bytes32 value.

Uninitialized roots have been a cause for serious exploits in the past - most notably the Nomad bridge exploit. Even though this is the responsibility of the integrating protocol, adding this additional check for empty roots would protect against cases where uninitialized roots are provided

Recommendations: Check that the provided `root` parameter is not empty in both functions

Customer's response: Fixed in 4644613

Fix Review: Fix confirmed

L-02 merkleizeKeccak() will contain valid 0 leaves in most cases		
Severity: **Low**	Impact: **Low**	Likelihood: **High**
Files: [Merkle.sol]	Status: Acknowledged	

Description: merkleizeKeccak() will initially calculate the numNodesInLayer using the following method:

```JavaScript
while (numNodesInLayer < leaves.length) {
 numNodesInLayer *= 2;
}
```

After that, it will create a layer array, assigning all leaves to it:

```JavaScript
 bytes32[] memory layer = new bytes32[](numNodesInLayer); for (uint256 i = 0; i <
leaves.length; i++) { layer[i] = leaves[i]; }
```

However, in most cases, the leaves.length < numNodesInLayer, resulting in many of the layers being initially set to 0.

As a result, the root will contain valid 0 leaves, which may be problematic for some future use cases.

Recommendations: Consider setting a default value for the unused layer slots.

Customer's response: "Acknowledged for the time being, as this fix would change how roots are calculated. May be fixed in the future after determining that this change will be compatible with our off-chain services."

L-03 Lack of validation on `leaves.length` is a power of two in merkleizeSha256()		
Severity: **Low**	Impact: **Low**	Likelihood: **Medium**
Files: Merkle.sol	Status: Fix Confirmed	

Description: The function merkleizeSha256() requires the input array leaves to have a length that is a power of two for correct Merkle root computation. However, this precondition is only documented and not enforced in the code. If leaves.length is not a power of two, the function will produce an incorrect root.

Recommendations: Add validation to check that leaves.length is a power of two before proceeding with the computation.

```
JavaScript
    function merkleizeSha256(
      bytes32[] memory leaves
    ) internal pure returns (bytes32) {
+           require(leaves.length > 0 && (leaves.length & (leaves.length - 1)) == 0, "not power of 2");
        //there are half as many nodes in the layer above the leaves
        uint256 numNodesInLayer = leaves.length / 2;
```

Customer's response: Fixed in 2f70510

Fix Review: Fix confirmed.

L-04 Intermediate node hashes can be used as leaves in proofs

Severity: **Low**	Impact: **Low**	Likelihood: **Medium**
Files: Merkle.sol	Status: Acknowledged	

Description: The structure of Merkle trees and Merkle proofs in the library allows an attacker to prove the inclusion of values that are not leaves in the tree. This is sometimes called a second preimage attack.

The following is an example of a fake leaf being proven to be included as a leaf even though it is not one in the original tree (in pseudo-Solidity for readability):

```JavaScript
bytes32[] arr = ["zero", "one", "two", "three"]
bytes memory proof = Merkle.getProofKeccak(arr, 0);
bytes32 root = Merkle.merkleizeKeccak(arr);

// The fake leaf is the hash of the first two leaves
bytes32 fakeLeaf = keccak256(abi.encodePacked(arr[0], arr[1]));
bytes memory fakeProof = proof[32:];

// Proving the inclusion of a real leaf
bool res1 = Merkle.verifyInclusionKeccak(proof, root, arr[0], 0);
// Proving the inclusion of the fake leaf for the same root
bool res2 = Merkle.verifyInclusionKeccak(fakeproof, root, fakeLeaf, 0);
assert res1 == res2;
```

There is also a risk that if leaf values are used directly by applications, for example, as destination addresses for an airdrop, an attacker could do a griefing or spamming attack even with random values or a small amount of work to satisfy formatting requirements.

Recommendations: One fix is to require the verifier to supply a trusted tree depth for an inclusion proof, which would indicate the shape of the tree and confirm that the proof is of the correct length for the tree height.

An alternative is for the contract to have “domain separation” between leaves and nodes by differentiating how they are hashed. This could include simply hashing the leaves differently than the nodes, such as using a different hash function for leaves, double-hashing them, or appending a “salt” to leaves to differentiate them from nodes.

Customer’s response: “Acknowledged. We have documentation recommending for users to salt their leaves or use different hashing algorithms, but do not assert it within the library itself.”

L-05 Keccak functions allow one-leaf trees, but SHA-256 functions don't		
Severity: **Low**	Impact: **Low**	Likelihood: **Medium**
Files: Merkle.sol	Status: Acknowledged.	

Description:

The functions which use Keccak as the hash function—`processInclusionProofKeccak()`, `merkleizeKeccak()`, and `getProofKeccak()`—correctly process trees with one leaf. On the other hand, the functions using SHA-256—`processInclusionProofSha256()`, and `merkleizeSha256()`—do not process trees with one leaf. This violates the NatSpec, which says that power-of-two length leaves are valid inputs.

This inconsistency also creates problems for future library users. The library presents two hash function implementations that behave differently for the same logical operation, forcing developers to implement different logic paths and edge case handling when supporting both hash functions. Remedying the behavioral inconsistency also requires more extensive documentation.

Recommendations: Standardize the behavior of the Keccak256 and SHA-256 functions regarding the handling of one-leaf trees. Update the documentation accordingly. If one-leaf trees are intentionally unsupported, provide clear error messages about this.

Customer's response: "Acknowledged and documented the discrepancy between the two functions due to how they're used for the EigenLayer protocol."

Informational Issues

I-01. Inefficient proof generation

Description: The `getProofKeccak()` function implements the following loop:

```
JavaScript
for (uint256 i = 0; i < layer.length; i++) {
      if (i == index) {
            uint256 siblingIndex = i + 1 - 2 * (i % 2);
            proof = abi.encodePacked(proof, layer[siblingIndex]);
            index /= 2;
        }
  }
```

At each iteration of a `layer` from the tree, it runs a nested loop to check if the `index` is inside the layer nodes. This is highly inefficient and becomes more expensive as the size of the tree grows

Recommendation: Consider using the following approach, which is simpler and significantly more efficient:

```
JavaScript
while (numNodesInLayer != 1) {

  if (index < layer.length) {
    uint256 siblingIndex = index + 1 - 2 * (index % 2);
    proof = abi.encodePacked(proof, layer[siblingIndex]);
    index /= 2;
  }
  ....
}
```

Customer's response: Fixed in f7d3a44

Fix Review: Fix confirmed

I-02. Validate that the provided index is correct

Description: The `getProofKeccak()` function takes an `index` argument, which is used to find the node in the `layer` array that is constructed. In case the provided `index` is outside of the bounds of the created layer array, the proof would always remain 0.

```JavaScript
for (uint256 i = 0; i < layer.length; i++) {
      if (i == index) {
            uint256 siblingIndex = i + 1 - 2 * (i % 2);
            proof = abi.encodePacked(proof, layer[siblingIndex]); // only updated if index is
valid
            index /= 2;
        }
  }
```

Recommendation: After building the `layer` array, make sure to check that the `index` is inside the boundaries (`layer.length`) and return early:

```JavaScript
if(index >= layer.length) revert WrongIndex()
```

Customer's response: Fixed in 5ef8d06.

Fix Review: Fix confirmed.

I-03. Inconsistency between merkleizeKeccak() and merkleizeSha256()

Description:

In `merkleizeSha256()`, the `numNodesInLayer` is calculated in the following way:

```
JavaScript
 uint256 numNodesInLayer = leaves.length / 2;
```

It is also stated in the NatSpec that the following precondition must be true in order for a proper result:

```
JavaScript
 *  @return The computed Merkle root of the tree.
     *  @dev A pre-condition to this function is that leaves.length is a power of two.  If
not, the function will merkleize the inputs incorrectly.
     */
    function merkleizeSha256(
```

However, in `merkleizeKeccak()`, the number of nodes is computed internally:

```
JavaScript
 uint256 numNodesInLayer = 1;
        while (numNodesInLayer < leaves.length) {
            numNodesInLayer *= 2;}
```

So for `merkleizeSha256()`, the proper number of leaves is expected to be provided in advance, while `merkleizeKeccak()` would pad the missing leaves in case the length is not a power of 2. This is inconsistent and makes using the library more challenging.

Recommendation:
Consider enforcing the approach from merkleizeKeccak() in both functions for consistent behaviour. Alternatively, enforce that the number of leaves is a valid power of two.

Customer's response: Fixed in 2f70510

Fix Review: Fix confirmed.

I-04. Empty proofs not explicitly handled in `processInclusionProofKeccak()`

Description: The current implementation of processInclusionProofKeccak() relies on an implicit behavior where, if the proof length is zero, the function returns the leaf without entering the loop. Although the existing comment mentions this behavior, it's not explicitly handled in the code. It is subtle and might not be immediately clear to readers or maintainers of the code.

Recommendation: Consider adding an explicit check at the start of the function to immediately return the leaf when the proof is empty:

```JavaScript
 function processInclusionProofKeccak(
        bytes memory proof,
        bytes32 leaf,
        uint256 index
 ) internal pure returns (bytes32) {
        require(proof.length % 32 == 0, InvalidProofLength());

        // Explicitly return leaf if proof is empty for readability
+       if (proof.length == 0) {
+           return leaf;
+       }
```

Customer's response: Fixed in d4fe4a0

Fix Review: Fix confirmed.

I-05. Missing index validation in Merkle proof verification

Description: The processInclusionProofKeccak() and processInclusionProofSha256() functions do not check that the index is fully reduced to 0 after proof traversal. This allows malformed or incomplete proofs to return a computed hash even if the proof or index is invalid. While this doesn't impact contracts that properly compare the result to a known root, it may lead to incorrect assumptions in library integration or misuse.

Recommendation: Add a final check to ensure the proof and index were fully consumed:

```javascript
require(index == 0, "Merkle: leftover index after proof");
```

Customer's response: Fixed in 99ad027.

Fix Review: Fix confirmed.

I-06. Misleading OZ top-level comment

Description: At the top of the Merkle library is a comment that advises using the OpenZeppelin MerkleTree and MerkleProof libraries to generate trees and proofs that are safe against the attack from L-04.

The contract should not advise the use of the OpenZeppelin MerkleTree library, since their Merkle trees are incompatible. The OZ MerkleTree library deals with two kinds of trees: "Standard" Merkle trees and "Simple" Merkle trees. In Standard Merkle trees, the leaves and nodes are sorted by hash, and leaves are double-hashed. In Simple Merkle trees, leaves are still individually hashed, which is not done in this library. Given that the trees generated by the OZ library are incompatible with this one, it is not a good practice to advise using trees and proofs generated by it.

Recommendation: Remove the parts of the comment that refer to the OpenZeppelin Merkle libraries.

Customer's response: Fixed in 0d856c4.

Fix Review: Fix confirmed

I-07. OZ versioning information in NatSpec

Description: The NatSpecs for `processInclusionProofKeccak()` and `processInclusionProofSha256()` include the line "_*Available since v4.4.*_" which refers to the OpenZeppelin library versioning information. The Eigenlayer Merkle library is not versioned the same way as the OZ Merkle trees library. This comment creates confusion by implying a similar versioning scheme and therefore compatibility between the two libraries.

Recommendation: Remove this function from the function NatSpecs.

Customer's response: Fixed in a2dde9e.

Fix Review: Fix confirmed.

I-08. Missing NatSpec for getProofKeccak

Description: The function `getProofKeccak()` lacks Natural Language Specification (NatSpec) documentation, while other functions in the library include a NatSpec comment documenting their behavior. The absence of NatSpec documentation hurts the developer and user experience as it enables automatic generation of user interfaces, developer tools, and human-readable contract descriptions. The missing documentation hides context about the function's behavior and potential risks from developers.

Recommendation: Add comprehensive NatSpec documentation to `getProofKeccak()` following the same format and detail level used for other functions in the library. The documentation should include parameter descriptions, return value specifications, and any relevant usage notes or warnings.

Customer's response: Fixed in e6a9481.

Fix Review: Fix confirmed.

I-09. Missing getProofSha256 function

Description: The library provides `getProofKeccak()`, which takes a `leaves` array and a leaf `index` parameter, returning a Merkle `proof` for the specified leaf. However, there is no corresponding `getProofSha256()` function, creating an asymmetry in the library's API. While all other functions have corresponding Keccak and SHA-256 variants, the proof generation functionality is only available for Keccak-based trees.

The missing `getProofSha256()` function forces developers using SHA-256-based Merkle trees to either implement proof generation themselves or rely on external libraries while using this library for verification. Developers implementing it themselves create greater risks, as custom implementations may have subtle differences and may not have the security of this library as audited. External library dependencies introduce potential incompatibilities between different proof generation implementations and increased dependency management complexity. Both approaches result in inconsistent behavior across the application stack, increased development time, and potential bugs. The API asymmetry may discourage usage of the SHA-256 functionality by integrators.

Recommendation: Implement `getProofSha256()` with the same interface and behavior as `getProofKeccak()` using SHA-256 as the hash function. This will ensure developers can generate and verify SHA-256 Merkle proofs using a single, consistent library.

Customer's response: Fixed in 5d53a67.

Fix Review: Fix confirmed.

I-10. Inconsistent NatSpec styles

Description: The library uses inconsistent NatSpec documentation styles across its functions. Some functions like `merkleizeSha256()` use the comprehensive `@notice/@param/@return` format, others like `verifyInclusionKeccak()` and `processInclusionProofKeccak()` use only `@dev` comments, and `getProofKeccak()` has no NatSpec documentation at all. Inconsistent documentation styles make it harder to quickly understand function interfaces and behavior, degrading the developer and user experience. The inconsistency also impacts automated tools that rely on structured NatSpec tags.

Recommendation: Standardize all functions to use the comprehensive `@notice/@param/@return/@dev` NatSpec format, which is used across Eigenlayer functions, both internal and external.

Customer's response: Fixed in e6a9481.

Fix Review: Fix confirmed.

I-11. Comment conflates Keccak with SHA-3

Description: The NatSpecs for `verifyInclusionKeccak()` and `processInclusionProofKeccak()` state: "*Note this is for a Merkle tree using the keccak/sha3 hash function…*" which incorrectly conflates Keccak with SHA-3. While Keccak is the family of hash functions that won the NIST competition for what would become SHA-3, the final SHA-3 standard released in 2014 differs from the original Keccak submission. NIST modified the padding in the submitted function, resulting in completely different hash outputs. Keccak-256 produces `c5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470` on empty input, while SHA3-256 produces `a7ffc6f8bf1ed76651c14756a061d662f580ff4de43b49fa82d80a4b80f8434a`.

This documentation error creates confusion for developers who may mistakenly believe the function accepts SHA-3-based Merkle proofs. Since Keccak-256 and SHA3-256 produce entirely different outputs, their respective Merkle trees are completely incompatible. Developers relying on the documentation might attempt to use SHA-3-generated proofs with `verifyInclusionKeccak()`, leading to verification failures. The confusion is particularly problematic since SHA-3 is not present as an opcode or precompile on Ethereum.

Recommendation: Remove the reference to SHA-3 from the NatSpec comments. The documentation should specify only Keccak or Keccak-256 to accurately reflect the hash function actually used by the verification logic.

Customer's response: Fixed in 87e89a6.

Fix Review: Fix confirmed.

I-12. Power-of-two padding enables a frontrunning DoS attack

Description: The `merkleizeKeccak()` and `getProofKeccak()` functions pad the number of leaves up to the next power of two. If the number of leaves is just under a power of two, a transaction involving a call to one of these functions can be front-run by a transaction that grows the tree beyond the power of two, doubling the computation required by the functions. This attack vector allows malicious actors to cause a denial of service against legitimate transactions by manipulating gas consumption through frontrunning. For example, consider the following contract:

```javascript
contract Test {
    bytes32[] arr;
    constructor() {
        for (uint i; i < 512; i++)
        {
            arr.push(keccak256(abi.encodePacked(i)));
        }
    }

    function testMerkleize() view external returns (bytes32 root) {
        root = Merkle.merkleizeKeccak(arr);
    }

    function frontrunner() external {
        arr.push(keccak256(abi.encodePacked(uint256(256))));
    }
}
```

`testMerkleize` initially costs 1,906,878 gas. After calling `frontrunner`, it costs 2,634,812 gas, a 38% increase. The 38% gas increase significantly exceeds the typical 10-20% safety margins included in gas estimations, causing transactions to fail with "Out of Gas" errors. An attacker can monitor the mempool for a transaction calling one of these functions and strategically increase the leaf count beyond the next power of two to push gas usage beyond the estimated gas cost. While the DoS is temporary, the attack can disrupt time-sensitive operations and cause oracle updates to fail.

Recommendation: Use precalculated hashes for padded parts of the tree to reduce the increase in computation required after crossing a power-of-two threshold.
Customer's response: Acknowledged.

I-13. Functions panic on empty and 1-length leaves arrays

Description: `merkleizeSha256()`, `merkleizeKeccak()`, and `getProofKeccak()` panic on empty leaves array inputs because they all access `leaves[0]` at some point without bounds checking. `merkleizeSha256()` also panics when `leaves.length == 1` because it creates a `layer` array with zero length when the input has only one element.

A panic should never occur in bug-free code. Instead, an error should be raised, as an error indicates a user input error rather than a system failure. Panics are difficult to handle gracefully in calling contracts and are less helpful for the user in determining the root cause of the error than a descriptive error. For a library intended for wide adoption, robust input validation is essential to prevent integration issues.

Recommendation: Explicitly check that leaves arrays meet minimum size requirements and return the appropriate descriptive error condition using `require` statements instead of allowing panics.

Customer's response: Fixed in b8bd6c1.

Fix Review: Fix confirmed.

I-14. Missing precondition documentation for non-empty proof requirement

Description: `processInclusionProofSha256()` contains the requirement:

```JavaScript
require(proof.length != 0 && proof.length % 32 == 0, InvalidProofLength())
```

However, the NatSpec does not document that empty proof arrays are invalid. Zero-length proof arrays correspond to the tree shape of one-leaf trees, making this restriction equivalent to rejecting single-leaf tree proofs.

The undocumented proof length restriction creates a hidden precondition that developers may only discover at runtime. The SHA-256 function `merkleizeSha256()` also rejects one-leaf trees, maintaining consistency within SHA-256 functions, but this behavior differs from the Keccak implementations, which accept single-leaf trees. Since there is no `getProofSha256()` function (see I-09), developers must use external implementations to generate SHA-256 proofs, increasing the risk of differences in behavior around edge cases like one-leaf trees. Developers may generate valid single-leaf proofs using external tools that will be unexpectedly rejected by this library's verification functions.

Recommendation: Document this precondition in the NatSpec.

Customer's response: Fixed in e6a9481.

Fix Review: Fix confirmed.

I-15. Merkle library lacks dedicated unit tests

Description: The `Merkle` library lacks dedicated unit tests, which other libraries such as `BytesLib`, `SlashingLib`, and `Snapshot` have. Without comprehensive testing, bugs may go undetected until production deployment. This is particularly dangerous for the Merkle library where incorrect verification logic could allow invalid proofs to pass, potentially critically compromising the security of dependent systems. The lack of tests also makes it difficult to validate that both Keccak and SHA-256 implementations produce the same logical results with their respective hash functions and specifications.

Recommendations: Consider adding comprehensive unit tests with complete code coverage for the Merkle library.

Customer's response: Acknowledged.

Disclaimer

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects, and incident response.