code4rena

Lido Finance

Smart Contract Security Assessment

Audit dates: Jul 16 — Aug 11, 2025

Overview

About C4

Code4rena (C4) is a competitive audit platform where security researchers, referred to as Wardens, review, audit, and analyze codebases for security vulnerabilities in exchange for bounties provided by sponsoring projects.

A C4 audit is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit outlined in this document, C4 conducted an analysis of the Lido Finance smart contract system. The audit took place from July 16 to August 11, 2025.

Final report assembled by Code4rena.

Summary

The C4 analysis yielded an aggregated total of 0 High and Medium vulnerabilities. Additionally, C4 analysis included 8 reports detailing issues with a risk rating of LOW severity or non-critical.

All of the issues presented here are linked back to their original finding, which may include relevant context from the judge and Lido Finance team.

Scope

The code under review can be found within the C4 Lido Finance repository, and is composed of 30 smart contracts written in the Solidity programming language and includes 5,896 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website, specifically our section on Severity Categorization.

Low Risk and Non-Critical Issues

For this audit, 8 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **won** received the top score from the judge.

The following wardens also submitted reports: 0xAura, Ahmerdrarerh, K42, madxcyber, MakeIChop, Sparrow, and Teycir.

[L-01] Batch Operation Fully Reverts on Invalid Ejection Entry

Type: Logic Error / Design Flaw

Summary

The `processBadPerformanceProof` function in the `CSStrikes` contract, which handles batch ejections for validators with excessive strikes, will revert the entire transaction if any single validator in the batch does not meet the required strike threshold for ejection. This "all-or-nothing" behavior is caused by a hard `revert` within the internal `_ejectByStrikes` function when it encounters an invalid entry.

Consequently, all other valid ejections within the same batch are also rolled back, preventing their timely processing. This design introduces operational inefficiency, as the entire batch must be filtered off-chain and resubmitted, potentially delaying necessary enforcement actions against poorly performing validators.

Description

In Lido's Community Staking Module, when processing batch validator ejections for excessive strikes, the entire batch operation will revert if any single entry in the batch fails a required precondition. This means no state changes or side effects are committed, even for entries that satisfy all requirements.

Concretely, in `CSStrikes.processBadPerformanceProof`, a list of validator keys is processed for potential ejection. For each entry, the function calls the internal `_ejectByStrikes` method, which calculates the total strikes and checks them against the configured threshold. If any entry in the batch has `strikes < threshold`, `_ejectByStrikes` reverts with `NotEnoughStrikesToEject()`. Due to EVM atomicity, this causes the entire transaction to revert, undoing all previous state changes and external calls, including for those entries that otherwise qualified for ejection.

in CSStrikes#L243-L245:

```
if (strikes < threshold) {
    revert NotEnoughStrikesToEject();
}
```

While enforcing the threshold check on-chain is reasonable to ensure protocol-level consistency and security, reverting the entire transaction due to a single invalid entry is unnecessarily restrictive. This design can prevent the timely processing of valid entries within the batch and introduce operational friction.

PoC

This PoC was implemented directly in the existing `CSStrikesProofTest` suite, not `PoC.t.sol`, due to the low severity and straightforward nature of the issue.

The scenario is:

- A user (or automation) submits a batch proof for ejection:
  - `[Key1 (sufficient strikes), Key2 (sufficient strikes), Key3 (insufficient strikes)]`
- `processBadPerformanceProof` begins processing.
- On reaching Key3, the function reverts, causing the entire transaction to revert.
- No validator is ejected, no penalties are recorded, and the operation must be retried offchain with a filtered list.

```
function
test_processBadPerformanceProof_RevertWhen_OneOfManyHasNotEnoughStrikes()
    public
{
    // 1. Setup test environment and manual Merkle tree.
    // Set the strikes threshold for ejection to 50.
    uint256 STRIKES_THRESHOLD = 50;
    module.PARAMETERS_REGISTRY().setStrikesParams(0, 6,
STRIKES_THRESHOLD);

    // Manually create leaves for success/failure scenarios.
    // Successful entry 1 (strikes > 50)
    (bytes memory pubkey0, ) = keysSignatures(1, 0);
    uint256[] memory strikesData0 = UintArr(60); // total strikes: 60
    leaves.push(Leaf(ICSStrikes.KeyStrikes({ nodeOperatorId: 0, keyIndex:
0, data: strikesData0 }), pubkey0));

    // Successful entry 2 (strikes > 50)
    (bytes memory pubkey1, ) = keysSignatures(1, 1);
    uint256[] memory strikesData1 = UintArr(70); // total strikes: 70
```

```
    leaves.push(Leaf(ICSStrikes.KeyStrikes({ nodeOperatorId: 1, keyIndex:
0, data: strikesData1 }), pubkey1));

    // Failing entry (strikes < 50)
    (bytes memory pubkey2, ) = keysSignatures(1, 2);
    uint256[] memory strikesData2 = UintArr(40); // total strikes: 40
    leaves.push(Leaf(ICSStrikes.KeyStrikes({ nodeOperatorId: 2, keyIndex:
0, data: strikesData2 }), pubkey2));

    // Build the Merkle tree with the constructed leaves.
    tree.pushLeaf(abi.encode(0, pubkey0, strikesData0));
    tree.pushLeaf(abi.encode(1, pubkey1, strikesData1));
    tree.pushLeaf(abi.encode(2, pubkey2, strikesData2));

    // Submit the Merkle root to the contract.
    bytes32 root = tree.root();
    vm.prank(oracle);
    strikes.processOracleReport(root, someCIDv0());

    // 2. Prepare proof data and mocking.
    uint256[] memory indicies = UintArr(0, 1, 2);
    ICSStrikes.KeyStrikes[] memory keyStrikesList = new
ICSStrikes.KeyStrikes[](indicies.length);
    for(uint256 i = 0; i < indicies.length; ++i) {
        keyStrikesList[i] = leaves[i].keyStrikes;
    }

    (bytes32[] memory proof, bool[] memory proofFlags) =
tree.getMultiProof(indicies);

    // Mock getSigningKeys calls for all entries in the loop.
    for (uint256 i = 0; i < indicies.length; i++) {
        Leaf memory leaf = leaves[indicies[i]];
        vm.mockCall(
            address(module),
            abi.encodeWithSelector(
                ICSModule.getSigningKeys.selector,
                leaf.keyStrikes.nodeOperatorId,
                leaf.keyStrikes.keyIndex,
                1
            ),
            abi.encode(leaf.pubkey)
        );
    }

    // 3. Expect revert and call the function.
```

```
    // The proof is valid, so the revert will occur inside
_ejectByStrikes due to threshold failure.
    vm.expectRevert(ICSStrikes.NotEnoughStrikesToEject.selector);

    this.processBadPerformanceProof{ value: keyStrikesList.length }(
        keyStrikesList,
        proof,
        proofFlags,
        address(0)
    );

    // Because the revert happens, even the entries that should have
succeeded do not have their state updated.
}
```

The following log shows that `EjectorMock::ejectBadPerformer` is successfully called for the first two entries in the batch (with sufficient strikes), but the transaction ultimately reverts on the third entry (`NotEnoughStrikesToEject()`), rolling back all prior state changes as expected.

```
Ran 1 test for test/CSStrikes.t.sol:CSStrikesProofTest
[PASS]
test_processBadPerformanceProof_RevertWhen_OneOfManyHasNotEnoughStrikes()
(gas: 880667)
...
|   |   ├─ [285] EjectorMock::ejectBadPerformer{value: 1}(0, 0,
CSStrikesProofTest: [...])
|   |   |   └─ ← [Stop]
|   |   ├─ [583] ExitPenaltiesMock::processStrikesReport(0, ...)
|   |   |   └─ ← [Stop]
|   |   ├─ [285] EjectorMock::ejectBadPerformer{value: 1}(1, 0,
CSStrikesProofTest: [...])
|   |   |   └─ ← [Stop]
|   |   ├─ [583] ExitPenaltiesMock::processStrikesReport(1, ...)
|   |   |   └─ ← [Stop]
|   |   └─ ← [Revert] NotEnoughStrikesToEject()
|   └─ ← [Revert] NotEnoughStrikesToEject()
└─ ← [Stop]
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 12.50ms
(1.93ms CPU time)
```

Impact

- This can lead to unnecessary delays in enforcement, particularly when large batches are submitted, as the entire batch must be retried after removing the invalid entry.
- When reporting large batches of poor-performing validators, a single invalid entry can cause unnecessary transaction failures and delay enforcement across the protocol.

Recommendations

To improve operational efficiency, the contract should be updated to process all valid entries within a batch and simply skip any invalid ones, emitting an event for each skipped entry. This ensures that valid reports are handled without delay, and skipped entries can still be tracked off-chain.

```
if (strikes >= threshold) {
    ejector.ejectBadPerformer{ value: value }(
        keyStrikes.nodeOperatorId,
        keyStrikes.keyIndex,
        refundRecipient
    );
    EXIT_PENALTIES.processStrikesReport(keyStrikes.nodeOperatorId,
pubkey);
} else {
        // emit event
}
```

[L-02] Mid-Season Merkle Root Update Invalidates Previously Valid Proofs

Type: Operational / Design Limitation

Summary

In the `VettedGate` referral reward system, the on-chain contract stores a single active `merkleRoot` and verifies claims with respect to that root. If the root is updated mid-season — even without removing a legitimate referrer from the set — all proofs generated against the previous root become invalid and cannot be used to claim rewards.

This behavior is inherent to Merkle proof verification: proofs are root-specific, so any root change (due to leaf removal, addition, or re-ordering) requires distributing new proofs to all eligible participants.

While this is not a security bug, it introduces an operational dependency: off-chain systems must re-issue updated proofs to all still-eligible referrers whenever the root changes.

Description

During a referral season, administrators may call:

in VettedGate.sol#L291-297:

```
vettedGate.setTreeParams(newRoot, newCid);
```

to replace the current Merkle root and CID. This may happen to ban malicious addresses or add new reward addresses. However:

- The contract does not store historical roots. and `claimReferrerBondCurve` always calls `verifyProof` against the current root.
- Because Merkle proofs are tied to a specific root, any proof generated for a previous root will become invalid after an update, even if the address remains in the new tree.

Example sequence:

1. Season starts, tree = `[NodeOperator, Stranger, AnotherNodeOperator]`.
2. Stranger reaches referral threshold.
3. Admin updates root mid-season to `[Stranger, NodeOperator]` to ban `AnotherNodeOperator`.
4. Stranger tries to claim with old proof (built for index=1 in old tree) → `InvalidProof` revert.
5. Stranger must obtain new proof (index=0 in new tree) to succeed.

This matches expected Merkle mechanics but can surprise operators if not accounted for.

PoC

This PoC was implemented directly in the existing `VettedGateReferralProgramTest` suite, not `PoC.t.sol`, due to the low severity and straightforward nature of the issue.

```
function test_proofBreaksAfterRootUpdate_whenIndexShifts() public {
    _addReferrals();
    bytes32[] memory oldProof = merkleTree.getProof(1); // stranger's
original index is 1

    MerkleTree newTree = new MerkleTree();
    newTree.pushLeaf(abi.encode(stranger)); // index now 0
    newTree.pushLeaf(abi.encode(anotherNodeOperator));
    bytes32 newRoot = newTree.root();

    vm.startPrank(admin);
    vettedGate.grantRole(vettedGate.SET_TREE_ROLE(), admin);
    vettedGate.setTreeParams(newRoot, "cid");
```

```
        vm.stopPrank();

        NodeOperatorManagementProperties memory no;
        no.rewardAddress = stranger;
        CSMMock(csm).mock_setNodeOperatorManagementProperties(no);

        // Old proof fails
        vm.expectRevert(IVettedGate.InvalidProof.selector);
        vm.prank(stranger);
        vettedGate.claimReferrerBondCurve(0, oldProof);

        // New proof works
        bytes32[] memory newProof = newTree.getProof(0);
        vm.prank(stranger);
        vettedGate.claimReferrerBondCurve(0, newProof);
    }
```

Test Output:

```
Ran 1 test for test/VettedGate.t.sol:VettedGateReferralProgramTest
[PASS] test_proofBreaksAfterRootUpdate_whenIndexShifts() (gas: 1228763)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 69.27ms
(4.04ms CPU time)
```

Impact

- Operational: Every root change forces all still-eligible participants to obtain a new proof before claiming.
- Timing risk: If root updates occur between reaching threshold and claiming, legitimate claims can fail until new proofs are distributed.
- User experience: Users unaware of the root update may face unexpected `InvalidProof` errors.

Recommendations

1. Document this behavior in the admin/operator playbook so off-chain systems automatically re-generate and distribute proofs upon root update. or
2. Consider keyed Merkle tree or index-stable design (e.g., sparse Merkle tree with address-based leaves) to minimize proof regeneration cost, though root changes will still invalidate old proofs. or
3. Optionally store previous root(s) temporarily and accept them for a grace period to reduce operational friction. or

4. A stricter on-chain safeguard could block `setTreeParams` when `isReferralProgramSeasonActive == true`, preventing mid-season root updates entirely — but this would significantly restrict operational flexibility (e.g., urgent malicious address removal) and may not be desirable in practice.

Disclosures

C4 is an open organization governed by participants in the community.

C4 audits incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Audit submissions are judged by a knowledgeable security researcher and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.