certora

# Security Assessment Final Report

# Shieldnet Staking

January 2026

Prepared for Safe Foundation

## Table of contents

# Project Summary

## Project Scope

| Project Name | Repository (link) | Initial commit | Final commit | Platform |
|---|---|---|---|---|
| Shieldnet Staking | safe-research/shieldnet | 687bdc0 | e97ed617 | EVM |

## Project Overview

This document describes the verification of **Shieldnet Staking** using manual code review. The work was undertaken from **January 5th, 2026** to **January 8th, 2026**.

The team performed a manual audit of the following Solidity contracts:

- `contracts/src/Staking.sol`

During the manual audit, the Certora team discovered bugs in the Solidity contracts code, as listed on the following page.

## Protocol Overview

Shieldnet is a distributed transaction checking network that enhances the security of Safe's smart wallet ecosystem. The audited Staking contract enables network participants to stake SAFE tokens toward registered validators, managing deposits and time-delayed withdrawals.

This non-upgradeable contract serves as the foundational layer for Shieldnet's validator network, tracking stake allocations and ensuring orderly withdrawal processes through a configurable delay mechanism. The contract focuses solely on stake management without implementing rewards or slashing, providing a simple and secure foundation for the network's operation.

## Findings Summary

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

| Severity | Discovered | Confirmed | Fixed |
|---|---|---|---|
| Critical | - | - | - |
| High | 1 | 1 | 1 |
| Medium | - | - | - |
| Low | 3 | 3 | 1 |
| Informational | 7 | 7 | 5 |
| **Total** | 11 | - | - |

## 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 |
|---|---|---|---|
| H-01 | Missing withdrawal ownership validation allows queue griefing or accidental fund loss | High | Fixed |
| L-01 | Single-step ownership transfer risks permanent loss of admin control | Low | Acknowledged |
| L-02 | Unrestricted ownership renouncement risks permanent loss of admin control | Low | Acknowledged |
| L-03 | No protection against overwriting pending validator proposals | Low | Fixed |
| I-01 | Staking is allowed on validators pending deregistration | Informational | Fixed |
| I-02 | proposeValidators() allows duplicate validators which can cause confusion | Informational | Acknowledged |
| I-03 | claimWithdrawal() is hard to integrate into DeFi | Informational | Fixed |
| I-04 | Staking contract can be registered as a validator | Informational | Fixed |
| I-05 | Lack of batch operations for gas efficiency | Informational | Acknowledged |

|  |  |  |  |
|---|---|---|---|
| I-06 | Cache storage reads to avoid redundant SLOADs | Informational | Fixed |
| I-07 | Cache storage pointers to avoid redundant keccak256 hashing | Informational | Fixed |

## High Severity Issues

| **H-01 Missing withdrawal node ownership validation allows permanent fund lock via queue corruption** | | |
|---|---|---|
| Severity: **High** | Impact: **High** | Likelihood: **High** |
| Files: [Staking.sol#L442-L480]() | Status: Fixed | |

**Description:** The `initiateWithdrawalAtPosition()` function fails to verify that the provided `previousId` belongs to the caller's withdrawal queue. This allows attackers to insert their withdrawals into other users' queues, corrupting the linked list structure and permanently locking victim funds.

The vulnerability stems from missing ownership validation. When users call `initiateWithdrawalAtPosition()`, they can specify any withdrawal ID as the insertion point, including IDs from other users' queues. The function only checks timestamp ordering but not queue ownership. By targeting another user's `tail` node, an attacker breaks a core invariant: `tail` nodes must have `next = 0`. But the function updates the `tail` of the attacker and not the victim. This results in a corrupted state for the victim queue which has a new node appended to it but without getting its `tail` updated. This corruption causes future withdrawals to become orphaned and unreachable.

The attack causes permanent fund loss. When victims try to withdraw after their corrupted `tail` node is claimed and deleted, the contract's logic tries to insert at a non-existent position. The new withdrawal becomes orphaned and not connected to the linked list but with funds already deducted from the user's balance. These funds remain locked in the contract forever with no recovery mechanism.

The exploit is straightforward and low-cost. Attackers only need to stake 1 wei and monitor for active withdrawal queues to target. The attack is particularly effective against users with pending withdrawals, and the cost to the attacker is minimal while the damage to victims can be substantial.

**Exploit Scenario:**

1. Alice stakes 100 SAFE
2. After some time Alice initiates withdrawal of 50 SAFE (creates node ID=1)
   - Queue state: `head=1`, `tail=1`
   - Node 1: `previous=0`, `next=0`
3. Bob calls `initiateWithdrawalAtPosition(validator, 1 wei, 1)` using Alice's `tail` ID. This corrupts Alice's queue by setting her tail node's `next` pointer to Bob's node but her `tail` node remains unchanged.
   - Queue state (unchanged): `head=1`, `tail=1`
   - Node 1 (modified): `previous=0`, `next=2`
4. When Alice claims her withdrawal via `claimWithdrawal()`, the head of her queue gets updated correctly but the `tail` pointer still references the deleted node (ID=1)
   - Queue state (modified): `head=2`, `tail=1`
5. Alice's next withdrawal attempt creates an orphaned node disconnected from the queue:
   - Node 3 (unreachable from head): `previous=1`
   - Queue state (modified): `head=2` (Bob's node), `tail=3` (Alice's orphaned node)
6. Once Bob's node is claimed, Alice's queue resets to empty, leaving her 50 SAFE tokens in Node 3 funds permanently lost:
   - Queue state (reset): `head=0`, `tail=0`

**Recommendations:** Extend the `WithdrawalNode` struct with an explicit owner field and enforce ownership checks in `initiateWithdrawalAtPosition()`. The function should revert if the supplied previousId does not belong to msg.sender, ensuring that withdrawals can only be inserted into the caller's own queue.

**Customer's response:** Fixed in PR#123

**Fix Review:** Fixed confirmed

## Low Severity Issues

| L-01 Single-step ownership transfer risks permanent loss of admin control | | |
|---|---|---|
| Severity: **Low** | Impact: **Medium** | Likelihood: **Low** |
| Files: [Staking.sol#L9]() | Status: Acknowledged | |

**Description:** The `Staking` contract uses OpenZeppelin's `Ownable` for access control which relies on a single-step ownership transfer. While secure, this pattern makes admin control fragile. If ownership is transferred to an incorrect or inaccessible address, the transfer is immediate and irreversible.

The owner in the contract has critical privileges, including managing validators, updating configuration values and recovering tokens. Losing ownership would permanently lock these functions. Although the likelihood of such a mistake is low, the risk is non-trivial given the severity of the impact.

**Recommendations:** Replace `Ownable` with OpenZeppelin's `Ownable2Step` which requires the new owner to explicitly accept ownership in a second transaction. This significantly reduces the risk of accidental, permanent loss of admin control.

**Customer's response:** Acknowledged. Since we expect the owner of the staking contract to be the Safe DAO, we do not think that additional overhead of a 2-step ownership transfer process is needed. The transactions will already be subject to a lot of scrutiny, be proposed way before they execute (allowing to back out in case the address is incorrect), and very unlikely to change ownership over the course of the lifetime of the contract.

| L-02 Unrestricted ownership renouncement risks permanent loss of admin control | | |
|---|---|---|
| Severity: **Low** | Impact: **Medium** | Likelihood: **Low** |
| Files: [Staking.sol#L9] | Status: Acknowledged | |

**Description:** The `Staking` contract inherits `renounceOwnership()` from OpenZeppelin's `Ownable`, allowing the owner to permanently relinquish control with a single call. Once triggered, the contract becomes permanently ownerless and the action cannot be reversed.

Losing ownership would freeze all admin functionality, including validator management, configuration updates and token recovery. While the likelihood of this happening is low, the severity of the outcome makes it a meaningful operational risk.

**Recommendations:** Override `renounceOwnership()` to disable it or replace it with a guarded two-step or timelocked process that prevents accidental execution while still allowing intentional renouncement if required.

**Customer's response:** Acknowledged. Given that the contract does not have a 2-step ownership transfer process, blocking `renounceOwnership` doesn't seem worthwhile (Given that it can be emulated with `transferOwnership(0xeee...eee)` for example). Additionally, we want to keep the possibility of signaling that there will be no more changes to the validators or configurations by allowing the Safe DAO to renounce ownership.

| L-03 No protection against overwriting pending validator proposals | | |
|---|---|---|
| Severity: **Low** | Impact: **Low** | Likelihood: **Low** |
| Files: Staking.sol#L9 | Status: Fixed | |

**Description:** The `proposeValidators()` function does not check whether a `pendingValidatorChangeHash` is already set. This allows the owner to silently overwrite an existing pending proposal by creating a new one, potentially leading to operational mistakes or confusion about which validator set is actually pending. There is also no dedicated function to explicitly cancel a pending proposal, forcing the owner to either execute it or overwrite it with another proposal.

**Recommendations:** Add a check to verify that `pendingValidatorChangeHash == bytes32(0)` before allowing a new proposal, or emit a clear warning/event. Additionally, implement a dedicated `cancelValidatorProposal()` function to allow explicit cancellation of pending proposals for better operational clarity.

**Customer's response:** Fixed in PR#151. The behaviour where a pending validator proposal is overwritten is intentional, documented as such.

**Fix review:** Fix confirmed

## Informational Issues

### I-01. Staking is allowed on validators pending deregistration

**Description:** The staking contract allows users to continue staking with validators that have been proposed for deregistration but are still within the timelock period. This behavior is intentional but it is not documented in the contract specification. While the spec states that deregistration does not block withdrawals, it does not clarify whether new staking remains allowed during this window.

This can confuse users and they may stake with a validator that is about to be removed and then need to withdraw shortly after, incurring unnecessary gas costs.

**Recommendation:** Update the contract specification and documentation to explicitly state that staking is allowed for validators pending deregistration during the timelock period.

**Customer's response:** Fixed in PR#150. It is possible to stake for validators that are being deregistered (via proposeValidators). Documented as such.

**Fix review:** Fix confirmed

### I-02. proposeValidators() allows duplicate validators which can cause confusion

**Description:** The `proposeValidators()` function accepts arrays of validators and flags but does not check for duplicates. If a validator is included multiple times with conflicting flags (e.g., `[ValA, ValA], [true, false]`), the final state is determined by the last entry, but the proposal hash includes both. This leads to confusing state, event spam, and potentially unexpected governance outcomes.

**Recommendation:** Add a check to ensure no duplicate addresses exist in the `validators` array.

**Customer's response:** Acknowledged. Since `proposeValidators` is a privileged function which can only be called by the owner (which, as mentioned above is expected to be the Safe DAO), we can expect certain sanity checks, and the worst that could happen in this case is that the owner is spending more gas than intended. Adding a check for non duplicates would also add more gas cost to the function.

### I-03. claimWithdrawal() is hard to integrate into DeFi

**Description:** `claimWithdrawal()` can be called by anyone, allowing malicious actors to claim withdrawals on behalf of users at strategically unfavorable times (e.g., to manipulate tax implications or exploit market conditions). This permissionless design also creates significant accounting complications for DeFi integrators and can disrupt user intent regarding withdrawal timing.

**Recommendation:** Modify `claimWithdrawal()` to require that `msg.sender == staker`.

**Customer's response:** Fixed in PR#147

**Fix review:** Fix confirmed

### I-04. Staking contract can be registered as a validator

**Description:** The protocol design principle states that "Shieldnet contract cannot operate on itself / cannot be a validator / cannot be a staker."

However, the implementation of `proposeValidators()` only validates that `validators[i] != address(0)` (line 545) but does not prevent the contract itself (`address(this)`) from being proposed as a validator.

**Recommendation:** Add explicit checks in `proposeValidators()` to reject `address(this)`:

```
None
require(validators[i] != address(this), InvalidAddress());
```

**Customer's response:** Fixed in PR#148

**Fix review:** Fix confirmed

## I-05. Lack of batch operations for gas efficiency

**Description:** The contract lacks batch functions for staking or withdrawing from multiple validators in one transaction. This forces users to execute multiple separate transactions, increasing gas costs and transaction overhead.

**Recommendation:** Consider implementing batch functions to improve gas efficiency for users managing stakes across multiple validators. This would significantly reduce gas costs for users managing diversified validator positions, especially important for institutional stakers or protocols building on top of the staking system.

**Customer's response:** Acknowledged. Adding batch operations is deemed unnecessary by the team. We want to optimize the flows for Smart Accounts like the Safe, which can always use MultiSend to batch things. Therefore we do not see sufficient added benefit to implementing a batched operation in the contract for the additional code surface required.

## I-06. Cache storage reads to avoid redundant SLOADs

**Description:** The `initiateWithdrawal()` function reads the same storage variables multiple times, wasting gas through redundant SLOADs. Each warm storage read costs ~100 gas.

When inserting at the tail (most common case), `queue.tail` is read 4 times. When inserting at the head, `queue.head` is read twice. These redundant reads can be eliminated by caching the values in memory.

**Recommendation:** Cache frequently accessed storage values:

```
None
// At tail insertion (saves ~300 gas):

uint64 tailCache = queue.tail;  // 1 SLOAD
uint64 currentId = tailCache;
if (currentId == tailCache) {
    withdrawalNodes[withdrawalId].previous = tailCache;
    withdrawalNodes[tailCache].next = withdrawalId;
    queue.tail = withdrawalId;
}
```

```
None
// At head insertion (saves ~100 gas):
uint64 headId = queue.head;
withdrawalNodes[withdrawalId].next = headId;
withdrawalNodes[headId].previous = withdrawalId;
```

This optimization saves approximately 300-400 gas per withdrawal with minimal code changes.

**Customer's response:** Fixed in PR#149

**Fix review:** Fix confirmed

### I-07. Cache storage pointers to avoid redundant keccak256 hashing

**Description:** Accessing mapping elements repeatedly causes redundant keccak256 hash computations (~60 gas each). The contract performs unnecessary hash calculations when accessing `withdrawalNodes[id]` multiple times within the same function.

In `initiateWithdrawal()`, during mid-queue insertion, the same node IDs are accessed multiple times resulting in 5 hash computations where only 3 are needed:

```
None
// Current: 5 keccak256 computations
uint64 nextId = withdrawalNodes[currentId].next;
withdrawalNodes[withdrawalId].previous = currentId;
withdrawalNodes[withdrawalId].next = nextId;
withdrawalNodes[currentId].next = withdrawalId;
withdrawalNodes[nextId].previous = withdrawalId;
```

Similarly, `initiateWithdrawalAtPosition()` performs 3 hash computations for previousId where only 1 is required:

```
None
// Current: 3 keccak256 + redundant SLOAD
require(withdrawalNodes[previousId].claimableAt > 0, InvalidWithdrawalNode());
require(withdrawalNodes[previousId].claimableAt <= claimableAt, InvalidOrdering());
nextId = withdrawalNodes[previousId].next;
```

**Recommendation:** Cache storage pointers to eliminate redundant hashing:

```
None
// initiateWithdrawal() - saves ~120 gas
WithdrawalNode storage currentNode = withdrawalNodes[currentId];
WithdrawalNode storage newNode = withdrawalNodes[withdrawalId];
uint64 nextId = currentNode.next;
newNode.previous = currentId;
newNode.next = nextId;
currentNode.next = withdrawalId;
withdrawalNodes[nextId].previous = withdrawalId;
```

```
None
// initiateWithdrawalAtPosition() - saves ~220 gas
WithdrawalNode storage prevNode = withdrawalNodes[previousId];
uint128 prevClaimableAt = prevNode.claimableAt;
require(prevClaimableAt > 0, InvalidWithdrawalNode());
require(prevClaimableAt <= claimableAt, InvalidOrdering());
nextId = prevNode.next;
```

This optimization reduces gas costs by approximately 120-220 gas per withdrawal operation.

**Customer's response:** Fixed in PR#149

**Fix review:** Fix confirmed

# 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.

The review was conducted over a limited timeframe and may not have uncovered all relevant issues or vulnerabilities.

# 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.