WstETH-Token Audit Report

Smart Contract Security Audit Report

Prepared by: Audit911

Date: December 3rd, 2025

Disclaimer & Confidentiality

> THIS IS A SECURITY AUDIT REPORT DOCUMENT WHICH MAY CONTAIN CONFIDENTIAL INFORMATION.
> This document includes potential vulnerabilities and malicious code analysis. It must be referred to internally and should only be made available to the public after issues are resolved.
>
> **Audit911** has analyzed this smart contract in accordance with the best industry practices. However, this audit does not guarantee explicit security of the audited smart contracts. All investors/users are advised to do their own due diligence.

---

1. Project Overview

1.1 Audit Scope

Item	Description
Project Name	WstETH (Wrapped liquid staked Ether 2.0)
Website	Lido Finance
Platform	Ethereum
Language	Solidity
Compiler Version	0.6.12
Audit Date	December 3rd, 2025

1.2 Introduction

The purpose of this audit was to ensure that the `WstETH` contract functions correctly as a wrapper for `stETH`, maintaining proper exchange rates between

shares and balances, and to identify any security vulnerabilities or business logic flaws present in the smart contract.

1.3 Project Background

The `WstETH` contract is a wrapper for the `StETH` token. Since `StETH` is a rebase token where balances change daily, it is not compatible with many DeFi protocols (like Uniswap or MakerDAO). `WstETH` wraps `StETH` into a non-rebase token where the underlying share value increases over time instead of the balance increasing.

---

2. Audit Summary

According to the standard audit assessment, the Customer's smart contract is **[Secured]**.

The contract contains no critical vulnerabilities that would lead to direct fund theft. However, several logic flaws regarding rounding precision and standard coding practices were identified.

2.1 Vulnerability Count

Severity	Count
🔴 **Critical**	0
🟠 **High**	0
🟡 **Medium**	0
🟢 **Low**	5
🔵 **Informational**	0

2.2 Auditor's Conclusion

We found 0 critical, 0 high, 0 medium, and 5 low issues. The contract logic is concise and relies heavily on the underlying `stETH` contract. The identified issues primarily concern edge-case rounding precision, griefing scenarios with dust amounts, and deviations from "Checks-Effects-Interactions" best practices.

---

3. Technical & Business Analysis

3.1 Technical Quick Stats

Main Category	Subcategory	Result
Contract Programming	Solidity version too old	[Moderated] (v0.6.12)
	Integer overflow/underflow	[Passed] (SafeMath used)
	Function input parameters lack check	[Moderated]
	Access control management	[Passed]
	Reentrancy / Race condition	[Passed]
Code Specification	Function visibility explicitly declared	[Passed]
	Unused code	[Passed]
Gas Optimization	"Out of Gas" Issue	[Passed]
	High consumption loops	[Passed]

3.2 Business Risk Analysis

Category	Result
Buy/Sell Tax	0%
Can Mint?	Yes (Anyone can mint by depositing stETH)
Blacklist Function?	No
Honeypot Risk?	Not Detected
Anti-Whale/Bot?	Not Detected
Hidden Owner?	Not Detected
Can Take Ownership?	No
Auditor Confidence	High

4. Code Quality & Security

4.1 Code Quality

The contract represents a standard wrapper implementation. It correctly inherits from `ERC20Permit` and `ERC20`. The usage of `SafeMath` protects against overflows. The logic for share conversion depends entirely on the external `stETH` contract.

4.2 Documentation

The code includes NatSpec comments explaining the purpose of `wrap` and `unwrap` functions, which is helpful.

4.3 Use of Dependencies

The contract relies heavily on OpenZeppelin contracts for ERC20 standards and `SafeMath`. It also relies on the `IStETH` interface for interaction with the underlying asset.

4.4 Centralization Risk

The contract is immutable and has no ownership functions or admin privileges. There is no centralization risk within *this specific contract*, although it is dependent on the `stETH` contract which may be upgradable.

---

5. Audit Findings

5.1 Severity Definitions

Level	Description
Critical	Vulnerabilities that can lead to token loss or asset theft.
High	Significant impact on contract execution or unauthorized access.
Medium	Important to fix, but may not lead to direct asset loss.
Low	Outdated code, style violations, business logic edge cases, or minor issues.

5.2 Detailed Findings

🔴 Critical

No Critical severity vulnerabilities were found.

🟠 High

No High severity vulnerabilities were found.

🟡 Medium

No Medium severity vulnerabilities were found.

🟢 Low / Informational

(1) Potential Dust Lock due to Rounding in UnwrapDescription: In the `unwrap` function, the contract calculates the amount of `stETH` to return based on shares burned: `stETHAmount = stETH.getPooledEthByShares(_wstETHAmount)`. Due to integer division (floor rounding), when `stETH` is transferred back to the user, the `stETH` contract converts this amount *back* to shares to deduct from the `WstETH` contract's balance. It is mathematically possible for the shares deducted to be slightly less than the `_wstETHAmount` burned by the user.
Recommendation: This leads to a tiny accumulation of "dust" shares in the WstETH contract that are unclaimable. While the economic impact is negligible (wei-level), strict accounting would require checking the actual shares burned in the underlying contract.
Status: Open

(2) Zero-Value Minting Griefing (Dust Loss)Description: In the `wrap` function, the contract requires `_stETHAmount > 0`. However, due to the share rate calculation, a very small `_stETHAmount` (e.g., 1 wei) could result in `wstETHAmount` being 0.
The code executes `_mint(msg.sender, 0)` and transfers the user's stETH to the contract. The user loses their dust amount of stETH and receives 0 WstETH.
Recommendation: Add a requirement to ensure the output amount is valid: `require(wstETHAmount > 0, "wstETH: zero wrap amount");`.
Status: Open

(3) Violation of Checks-Effects-Interactions (CEI) PatternDescription: In the `wrap` function, the `_mint` (state change/effect) occurs before `stETH.transferFrom` (external interaction).

```
_mint(msg.sender, wstETHAmount);
stETH.transferFrom(msg.sender, address(this), _stETHAmount);
```

While `stETH` is currently a trusted contract, this violates the standard CEI security pattern. If `stETH` were to add hooks or be upgraded to include callbacks, this could theoretically open up reentrancy vectors.
Recommendation: Perform the transfer first, confirm success (or let it revert), and then mint the tokens.
Status: Open

(4) Missing Zero Address Check in ConstructorDescription: The `constructor` initializes the `stETH` contract address but does not validate that `_stETH` is not `address(0)`.
Recommendation: Add `require(_stETH != address(0), "Invalid address");` in the constructor to prevent deployment errors.
Status: Open

(5) Lack of Specific Business EventsDescription: The `wrap` and `unwrap` functions only emit the standard ERC20 `Transfer` events (via `_mint` and `_burn`). There are no specific events that emit the input/output amounts of the conversion (e.g., `event Wrapped(address indexed user, uint256 stETHAmount, uint256 wstETHAmount)`). This makes off-chain indexing and analytics more difficult.
Recommendation: Define and emit custom events for `wrap` and `unwrap` operations.
Status: Open

---

6. Methodology

Audit911 uses a collaborative and transparent process to improve system quality.

1. **Manual Code Review:** We examine code logic, error handling, protocol parsing, and cryptographic errors.
2. **Vulnerability Analysis:** We investigate threat models, attack surfaces, and known vulnerabilities (e.g., Reentrancy, Front-running).
3. **Automated Tools:** We utilize industry-standard tools such as Slither, Solhint, and Remix IDE to detect common issues.
4. **Reporting:** We document all findings, verify their feasibility, and suggest remediation steps.
