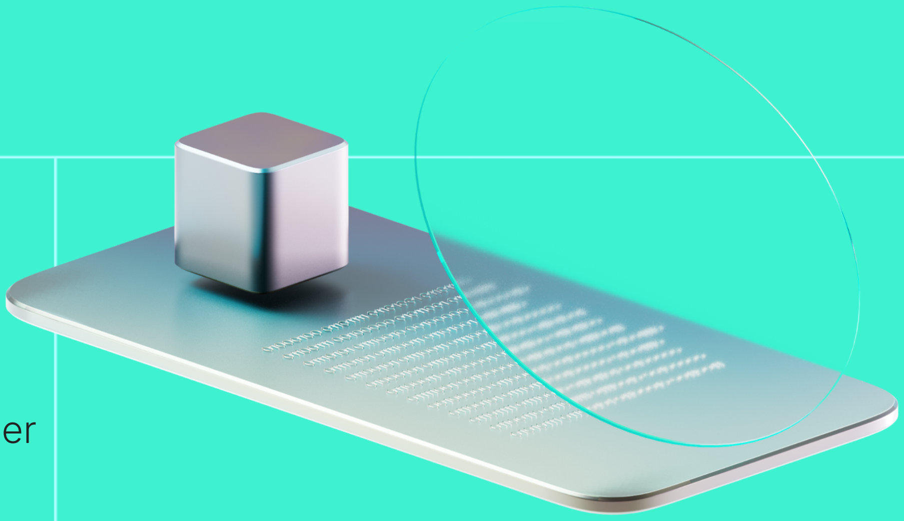
Smart Contract Code Review And Security Analysis Report



Customer: StakeTogether

Date: 15 Nov, 2023

We thank the StakeTogether team for allowing us to conduct a Smart Contract Security Assessment. This document outlines our methodology, limitations, and results of the security assessment.

The StakeTogether Protocol, a pioneering project, delves deep into the specifications of Ethereum's staking landscape. The aim of the project is focusing on precision and efficiency, providing users with a robust framework to participate in transaction validation seamlessly. The StakeTogether Protocol aligns with Ethereum's transition to Proof of Stake, elevating energy efficiency and scalability while fortifying network security. This project sets new standards in the staking arena by optimizing user experience, ensuring high levels of decentralization, and maximizing rewards.

Users can deposit ETH into user-generated pools and receive stpETH tokens. Once a pool accumulates 32.1 ETH, a validator is initiated on the beacon chain. The validator reports automated tasks, such as reinvesting rewards or facilitating withdrawal requests.

Platform: EVM

Language: Solidity

Tags: ERC-20, Liquidity Pools, Proof-of-Stake

Timeline: 27.10.2023 - 15.11.2023

Methodology: Link

Last reviewed scope

Repository	https://github.com/staketogether/st-v1-contracts
Commit	06746c
Remediations PR	https://github.com/staketogether/st-v1-contracts/pull/26/commits

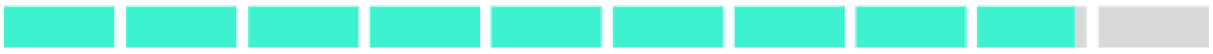
Hacken OU
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia

View full scope

Audit Summary

10/10	9/10	100%	9/10
Security score	Code quality score	Test coverage	Documentation quality score

Total: 8.8/10



The system users should acknowledge all the risks summed up in the risks section of the report.

4	0	0	0
Total Findings	Resolved	Acknowledged	Mitigated

Findings by severity	Findings Number	Resolved	Mitigated	Acknowledged
Critical	0	0	0	0
High	0	0	0	0
Medium	0	0	0	0
Low	0	0	0	0

Document

Name	Smart Contract Code Review and Security Analysis Report for StakeTogether
Approved By	Luciano Ciattaglia | Director of Services at Hacken OÜ Grzegorz Trawiński | SC Audits Expert at Hacken OÜ
Audited By	Ataberk Yavuzer | Senior SC Auditor at Hacken OÜ
Website	https://staketogether.org/
Changelog	09.11.2023 – Preliminary Report 15.11.2023 – Final Report

Introduction

Hacken OÜ (Consultant) was contracted by Client (Customer) to conduct a Smart Contract Code Review and Security Analysis. This report presents the findings of the security assessment of the Customer's smart contracts.

System Overview

StakeTogether is a staking protocol with the following contracts:

- **Airdrop** — distributes stpETH incentives through Oracle Reports via Merkle Tree. Users with Merkle proofs claim rewards for specific block numbers.
- **Router** — handles routing functionalities via submitting and executing Oracle reports. It also manages access control functionality for Report Oracles.
- **StakeTogether** — the primary entry point for interaction with the protocol. It allows users to create pools and deposits to claim protocol rewards. It also has anti-fraud functionality to prevent blacklisted addresses.

  StakeTogether token has the following attributes:

  - Name: Stake Together Protocol
  - Symbol: stpETH
  - Decimals: 18

- **StakeTogetherWrapper** — the wrapper contract for StakeTogether token. It allows users to convert their stpETH into wstpETH.

  It has the following attributes:

  - Name: Wrapped Stake Together Protocol
  - Symbol: wstpETH
  - Decimals: 18

- **Withdrawals** — handles all withdrawal-related activities within the StakeTogether protocol. It allows users to withdraw their staked tokens and interact with the associated stake contracts. It also follows the ERC-20 standard.

  It has the following attributes:

  - Name: Stake Together Withdrawals
  - Symbol: stwETH
  - Decimals: 18

Privileged roles

- **DEFAULT_ADMIN_ROLE:**
  - This role is the most privileged role in the protocol. Users with this role can grant other roles below to protocol users. This role should be granted to multi-sig wallet only due to centralization risks.

- **UPGRADER_ROLE:**
    - The *UPGRADER_ROLE* is responsible for all contract upgrades in the protocol.
- **ADMIN_ROLE:**
    - This role can call *pause()* and *unpause()* functionalities for contracts in case of any emergency. It is also responsible for setting contract addresses and protocol configurations.
- **POOL_MANAGER_ROLE:**
    - The *POOL_MANAGER_ROLE* has authority on removing pools from the StakeTogether contract.
- **VALIDATOR_MANAGER_ROLE:**
    - The only functionality of this role is removing validators from the StakeTogether contract in case of emergency.
- **VALIDATOR_ORACLE_MANAGER_ROLE:**
    - This role is responsible for managing Validator Oracles in the protocol. Any user with *VALIDATOR_ORACLE_MANAGER_ROLE* can add new Validator Oracles to protocol or remove Validator Oracles from the protocol.
- **ORACLE_REPORT_MANAGER_ROLE:**
    - The ORACLE_REPORT_MANAGER_ROLE is responsible for adding *Report Oracles* to the Router contract.

- **REPORT ORACLES:**
  - Users with the Report Oracle role have multiple responsibilities such as submitting and executing oracle reports to the router contract to update Withdraw and Beacon balances. They are able to set reward incentives for users for specific blocks by setting Merkle Roots. In addition, Report Oracles can process rewards with protocol profits. Report Oracles should reach consensus via the *executeReport()* functionality to be able to execute submitted reports.
- **VALIDATOR_ORACLE_SENTINEL_ROLE:**
  - This role is responsible for forcing the selection of the next validator oracle via the *forceNextValidatorOracle()* function.
- **ANTI_FRAUD_MANAGER_ROLE:**
  - Users with the *ANTI_FRAUD_MANAGER_ROLE* can add addresses to the anti-fraud list in case of fraud detection in the protocol. They are also allowed to remove addresses from the anti-fraud list.
- **ANTI_FRAUD_SENTINEL_ROLE**
  - This role is very similar to the *ANTI_FRAUD_MANAGER_ROLE*. The only difference is users with this role are unable to remove addresses from the fraud-list.

Executive Summary

The score measurement details can be found in the corresponding section of the scoring methodology.

Documentation quality

The total Documentation Quality score is **9** out of **10**.

- Functional requirements are explained in detail.
- Technical descriptions are provided.
- The whitepaper does not explain everything about the protocol when it is compared with the README section on project repository.

Code quality

The total Code Quality score is **9** out of **10**.

- The code has unused variables and functions in several places.
- The development environment is correctly configured.

Test coverage

Code coverage of the project is **75.17%** (branch coverage), with a mutation score of 100%.

- Deployment and basic user interactions are covered with tests.
- Access controls are covered.
- Staking and transfer related tests are covered.
- Share calculation tests are covered.

Security score

As a result of the audit, the code contains **0** Low and **0** Informational issues. The security score is **10** out of **10**.

All found issues are displayed in the "Findings" section.

Summary

According to the assessment, the Customer's smart contract has the following score: **8.8**. The system users should acknowledge all the risks summed up in the risks section of the report.

Findings

■■■■ Critical

No critical severity issues were found.

■■■ High

No high severity issues were found.

■■ Medium

No medium severity issues were found.

■ Low

L01. The addPool() function may not work when StakeTogether is empty

Impact	Low
Likelihood	Low

The *addPool()* function on the *StakeTogether* contract allows users to create pools and perform staking. They will receive **stpETH** for their staking. However, it has been noticed that the *_processStakePool()* internal function on this function does not control the *StakeTogether* balance. If there is no ether on the *StakeTogether* contract, the *addPool()* function will give a **Division by Zero** error and will not work until someone increases the balance of the *StakeTogether* contract.

```
function addPool(
    address _pool,
    bool _listed,
    bool _social
  ) external payable nonReentrant whenNotPaused {
    if (_pool == address(0)) revert ZeroAddress();
    if (pools[_pool]) revert PoolExists();
    if (!hasRole(POOL_MANAGER_ROLE, msg.sender) || msg.value > 0) {
      if (!config.feature.AddPool) revert FeatureDisabled();
      if (msg.value != fees[FeeType.StakePool].value) revert InvalidValue();
      _processStakePool();
    }
    pools[_pool] = true;
    emit AddPool(_pool, _listed, _social, msg.value);
  }
```

The *sharesAmount* calculation will be reverted if totalSupply is zero.

```
function _processStakePool() private {
    uint256 amount = fees[FeeType.StakePool].value;
    uint256 sharesAmount = Math.mulDiv(amount, totalShares, totalSupply() - amount);
    _distributeFees(FeeType.StakePool, sharesAmount, address(0));
  }
```

Path:

- StakeTogether.sol#L808

Recommendation: Add a sanity check to prevent users calling the *addPool()* function when *totalSupply* is zero.

Found in: 06746c

Status: Fixed (Revised commit: Pull Request no. 26 - 926dd6)

Remediation: The StakeTogether contract checks and reverts if the stpETH balance is less than 1 ETH. As a result of this change, it will no longer be possible to receive a division by zero error.

L02. Missing sanity check can lead to reward miscalculations

Impact	Low
Likelihood	Low

Users in the Report Oracle role send new oracle reports to the protocol using the Router contract. These reports include some fields such as profitAmount, profitShares, lossAmount, withdrawAmount, withdrawRefundAmount and routerExtraAmount. If the report to be sent contains profitAmount and profitShares, the reward will be distributed. However, the *isReadyToSubmit()* function does not check the condition where there is positive *profitAmount* and zero *profitShares* in the report. Therefore, reward distribution can be miscalculated in this edge case scenario:

```
if (_report.profitAmount > 0) {
      if (_report.lossAmount != 0) {
        revert LossMustBeZero();
```

Path:

- Router.sol#L395-L398

Recommendation: Consider adding a sanity check by also controlling the *_report.profitShares* variable to prevent the possibility of this edge case scenario.

Found in: 06746c

Status: Mitigated (Revised commit: n/a)

Remediation: The StakeTogether team stated that this sanity check already exists on the Oracle code and that these checks are being carried out along with other similar checks via Sentinel Oracles.

L03. The _blockNumber variable should be a part of Merkle Tree in the claimAirdrop() function

Impact	Low
Likelihood	Low

The Airdrop contract can distribute rewards by setting Merkle Root on oracle reports to users through reports accepted by Report Oracles for blocks.

In case of a root reuse, it is safe to include the *_blockNumber* variable as part of the Merkle tree. In case a Report Oracle submits the same Merkle Root twice for different Oracle Reports, the previous proofs should not be re-used.

```
bytes32 leaf = keccak256(bytes.concat(keccak256(abi.encode(_index, _account,
_sharesAmount))));
```

Path:

- Airdrop.sol#L141

Recommendation: It is recommended to include the *_blockNumber* parameter to Merkle leaf to prevent replay attacks.

Found in: 06746c

Status: Fixed (Revised commit: Pull Request no. 26 - 142c27)

Remediation: The *_blockNumber* variable renamed as *_reportBlock.* In addition, the *_reportBlock* variable was included in the leaf hash to prevent possible Signature Replay attacks.

Informational

I01. Missing __ReentrancyGuard_init() function calls on contracts

Most contracts on the protocol use Upgradeable and Proxy patterns and their implementations require the use of *initialize()* functions instead of constructors. Nearly all inheritance contracts were initialized with these *_init()* functions to prevent any unwanted situations on the protocol.

However, the *initialize()* functions on protocol contracts do not call *__ReentrancyGuard_init()* function to set the Reentrancy Guard storage variable to *NOT_ENTERED* which equals to *uint256(1)*.

```
function __ReentrancyGuard_init() internal onlyInitializing {
    __ReentrancyGuard_init_unchained();
}

function __ReentrancyGuard_init_unchained() internal onlyInitializing {
    ReentrancyGuardStorage storage $ = _getReentrancyGuardStorage();
    $._status = NOT_ENTERED;
}
```

Path:

- Airdrop.sol#L25
- Router.sol#L27
- StakeTogether.sol#L33
- StakeTogetherWrapper.sol#L31
- Withdrawals.sol#L31

Recommendation: Consider calling the *__ReentrancyGuard_init()* function to *initialize()* functions of inherited contracts.

Found in: 06746c

Status: Fixed (Revised commit: Pull Request no. 26 - 9c819a)

Remediation: The *__ReentrancyGuard_init()* function call was implemented to be called in the *initialize()* function of all inherited contracts in the protocol.

I02. Unneeded initializations of uint256 and bool variable to 0/false

In Solidity, it is common practice to initialize variables with default values when declaring them. However, initializing **uint256** variables to **0** and **bool** variables to **false** when they are not subsequently used in the code can lead to unnecessary gas consumption and code clutter. This issue points out instances where such initializations are present but serve no functional purpose.

There are several impacts of unneeded initializations:

1. **Gas Consumption:** Unneeded initializations of uint256 variables to 0 consume gas when deploying and executing smart contracts on the Ethereum blockchain. While the gas cost may be relatively small for individual variables, it can add up significantly in larger and more complex contracts, leading to higher transaction fees.
2. **Code Clutter:** Redundant initializations make the code less clean and harder to read. They can be confusing to developers who might wonder if the initialized values have some significance or are intended for future use.
3. **Maintenance Overhead:** Unnecessary initializations introduce additional lines of code that need to be maintained. If the initializations are forgotten or not updated when the variable's purpose changes, it can lead to bugs and inconsistencies in the contract logic.

Path:

- StakeTogether.sol#L517
- StakeTogether.sol#L520
- StakeTogether.sol#L738
- StakeTogether.sol#L739
- StakeTogether.sol#L762

Recommendation: Consider removing unneeded variable initializations from contracts.

Found in: 06746c

Status: Fixed (Revised commit: Pull Request no. 26 - 0229ab)

Remediation: Unnecessary variable initializations were removed from the code base in order to optimize gas consumption.

I03. For loop optimizations

In for loops, the index variable is incremented using **i++ (post-fix).** It is known that in loops, using **++i (pre-fix)** costs less gas per iteration than **i++**. It has also been detected that some **uint256** variables are initialized to **0** in for loops.

In addition, starting from pragma version **0.8.0**, implementing the **unchecked** keyword for arithmetical operations can reduce gas usage on contracts where overflow/underflow is extremely unlikely.

Path:

- StakeTogether.sol#L520
- StakeTogether.sol#L739
- StakeTogether.sol#L764
- StakeTogether.sol#L774

Recommendation: It is suggested to apply the following pattern for for-loops in Solidity pragma version 0.8.0 and later. All for loops in contracts should be replaced with the following pattern if possible:

```
for (uint i; i < arrayLength; ) {
    ...
    unchecked {
        ++i;
    }
}
```

Found in: 06746c

Status: Fixed (Revised commit: Pull Request no. 26 - 6920f8)

Remediation: Unchecked loop increments were introduced on Solidity version 0.8.22. The StakeTogether team upgraded the Solidity version to 0.8.22 for all contracts in order to optimize gas consumption in for-loops.

- Solidity 0.8.22 Release Announcement

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited smart contracts.

Appendix 1. Severity Definitions

When auditing smart contracts Hacken is using a risk-based approach that considers the potential impact of any vulnerabilities and the likelihood of them being exploited. The matrix of impact and likelihood is a commonly used tool in risk management to help assess and prioritize risks.

The impact of a vulnerability refers to the potential harm that could result if it were to be exploited. For smart contracts, this could include the loss of funds or assets, unauthorized access or control, or reputational damage.

The likelihood of a vulnerability being exploited is determined by considering the likelihood of an attack occurring, the level of skill or resources required to exploit the vulnerability, and the presence of any mitigating controls that could reduce the likelihood of exploitation.

Risk Level	High Impact	Medium Impact	Low Impact
High Likelihood	Critical	High	Medium
Medium Likelihood	High	Medium	Low
Low Likelihood	Medium	Low	Low

Risk Levels

Critical: Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.

High: High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.

Medium: Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.

Low: Major deviations from best practices or major Gas inefficiency. These issues will not have a significant impact on code execution, do not affect security score but can affect code quality score.

Impact Levels

High Impact: Risks that have a high impact are associated with financial losses, reputational damage, or major alterations to contract state. High impact issues typically involve invalid calculations, denial of service, token supply manipulation, and data consistency, but are not limited to those categories.

Medium Impact: Risks that have a medium impact could result in financial losses, reputational damage, or minor contract state manipulation. These risks can also be associated with undocumented behavior or violations of requirements.

Low Impact: Risks that have a low impact cannot lead to financial losses or state manipulation. These risks are typically related to unscalable functionality, contradictions, inconsistent data, or major violations of best practices.

Likelihood Levels

High Likelihood: Risks that have a high likelihood are those that are expected to occur frequently or are very likely to occur. These risks could be the result of known vulnerabilities or weaknesses in the contract, or could be the result of external factors such as attacks or exploits targeting similar contracts.

Medium Likelihood: Risks that have a medium likelihood are those that are possible but not as likely to occur as those in the high likelihood category. These risks could be the result of less severe vulnerabilities or weaknesses in the contract, or could be the result of less targeted attacks or exploits.

Low Likelihood: Risks that have a low likelihood are those that are unlikely to occur, but still possible. These risks could be the result of very specific or complex vulnerabilities or weaknesses in the contract, or could be the result of highly targeted attacks or exploits.

Informational

Informational issues are mostly connected to violations of best practices, typos in code, violations of code style, and dead or redundant code.

Informational issues are not affecting the score, but addressing them will be beneficial for the project.

Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

Scope details

Repository	https://github.com/staketogether/st-v1-contracts
Commit	06746c
Whitepaper	Link
Requirements	Link
Technical Requirements	Link

Contracts in Scope

contracts/Airdrop.sol
contracts/Router.sol
contracts/StakeTogether.sol
contracts/StakeTogetherWrapper.sol
contracts/Withdrawals.sol