Security Audit Report for StakeTogether st-v1-contracts

Date: Sep 04, 2023

Version: 1.1

Contact: contact@blocksec.com

BlockSec Audit Team

Digitally signed by BlockSec Audit Team
Date: 2023.09.04 11:06:41 +08'00'

Contents

Report Manifest

Item	Description
Client	StakeTogether
Target	StakeTogether st-v1-contracts

Version History

Version	Date	Description
1.0	Sep 03, 2023	First Release
1.1	Sep 04, 2023	Add feedback for 2.4.4

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The target of this audit is the code repo of the smart contracts [1] of StakeTogether project, which is an Ethereum staking protocol designed especially for communities. Specifically, it allows users to deposit ETH into staking pools, receiving `stpETH` tokens as collateral. When a pool reaches 32 ETH, a validator is created on the Ethereum 2.0 beacon chain. Daily oracle reports trigger automated actions like restaking rewards or processing withdrawal requests by burning `stpETH` to generate `stwETH` tokens.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report.

Project	Version	Commit Hash
st-v1-contracts	`Version 1`	`9f887b12c195c6396ec0cf377c708b22417a215d`
	`Version 2`	`85c0c7112954b25bb1b8e1af7bd1dabcfb84b50a`
	`Version 3`	`ce28ea08185b31aca936f38be831aef21112f304`

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

[1] `https://github.com/staketogether/st-v1-contracts/`

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

- Reentrancy
- DoS
- Access control
- Data handling and data flow
- Exception handling
- Untrusted external call and control flow
- Initialization consistency
- Events operation
- Error-prone randomness
- Improper use of the proxy system

1.3.2 DeFi Security

- Semantic consistency
- Functionality consistency
- Permission management
- Business logic
- Token operation
- Emergency mechanism
- Oracle security
- Whitelist and blacklist
- Economic impact
- Batch transfer

1.3.3 NFT Security

- Duplicated item
- Verification of the token receiver
- Off-chain metadata security

1.3.4 Additional Recommendation

* Gas optimization
* Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

		Likelihood	
		High	*Low*
Impact	*High*	High	Medium
	Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

Chapter 2 Findings

In total, we find **six** potential issues. Besides, we also have **two** recommendations and **four** notes.

- Medium Risk: 1
- Low Risk: 5
- Recommendation: 2
- Note: 4

ID	Severity	Description	Category	Status
1	Low	Deposit revert for the first depositor	Software Security	Fixed
2	Medium	Potential DoS attack when executing the report	DeFi Security	Fixed
3	Low	Lack of existence check when adding `validators`	DeFi Security	Fixed
4	Low	Ineffective check due to incorrect initialization	DeFi Security	Fixed
5	Low	Lack of existence check when blacklisting the `reportOracles`	DeFi Security	Fixed
6	Low	Potential DoS attack in the consensus process	DeFi Security	Fixed
7	-	Add sanity checks for function parameters	Recommendation	Fixed
8	-	Remove duplicate checks	Recommendation	Fixed
9	-	Centralization risk	Note	-
10	-	Ensure the correctness of the configuration	Note	-
11	-	Risk of insufficient report oracles	Note	-
12	-	Potential off-chain risks	Note	-

The details are provided in the following sections.

2.1 Software Security

2.1.1 Deposit revert for the first depositor

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description To avoid inflation attacks, the `StakeTogether` contract creates dead shares (1e18) upon initialization. However, the contract balance remains zero. For the first call to the `_depositBase` function, the `_processStakeEntry` function is invoked to calculate shares based on the provided formula: $\_amount \times \frac{totalShares}{(totalSupply() - \_amount)}$. This formula uses a denominator that will be zero for the first depositor, resulting in a `divisionError` error.

```
  function _depositBase(address _to, DepositType _depositType, address _referral) private {
    require(config.feature.Deposit, 'FD'); // FD = Feature Disabled
    require(msg.value >= config.minDepositAmount, 'MD'); // MD = Min Deposit

    _resetLimits();

    if (msg.value + totalDeposited > config.depositLimit) {
      emit DepositLimitReached(_to, msg.value);
```

```
            revert('DLR');
        }

        _processStakeEntry(_to, msg.value);

        totalDeposited += msg.value;
        emit DepositBase(_to, msg.value, _depositType, _referral);
    }
```

Listing 2.1: StakeTogether.sol

```
    function _processStakeEntry(address _to, uint256 _amount) private {
        uint256 sharesAmount = MathUpgradeable.mulDiv(_amount, totalShares, totalSupply() - _amount
            );
        _distributeFees(FeeType.StakeEntry, sharesAmount, _to);
    }
```

Listing 2.2: StakeTogether.sol

Impact The first deposit will revert with `divisionError`.

Suggestion Enforce an increase of the `totalSupply` value before enabling deposits.

2.2 DeFi Security

2.2.1 Potential DoS attack when executing the report

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `Router` contract, any oracle can execute a valid and executable report via the `executeReport` function. This function initially checks the report's validity by invoking the `isReadyToExecute` function. Within `isReadyToExecute`, there is a restriction on Line 362 that requires the `beaconBalance` in the `StakeTogether` contract to be greater than or equal to the sum of `lossAmount` and `withdrawRefundAmount` specified in the report.

```
354    function isReadyToExecute(Report calldata _report) public view returns (bytes32) {
355        bytes32 hash = keccak256(abi.encode(_report));
356        require(!revokedReports[_report.epoch], 'REVOKED_REPORT');
357        require(!executedReports[_report.epoch][hash], 'REPORT_ALREADY_EXECUTED');
358        require(consensusReport[_report.epoch] == hash, 'REPORT_NOT_CONSENSUS');
359        require(totalOracles >= config.minOracleQuorum, 'MIN_ORACLE_QUORUM_NOT_REACHED');
360        require(block.number >= reportDelayBlocks[hash] + config.reportDelayBlocks, '
               TOO_EARLY_TO_EXECUTE');
361        require(
362          _report.lossAmount + _report.withdrawRefundAmount <= stakeTogether.beaconBalance(),
363          'NOT_ENOUGH_BEACON_BALANCE'
364        );
365        require(
366          address(this).balance >=
367            (_report.profitAmount +
            _report.withdrawAmount +
            _report.withdrawRefundAmount +
            _report.routerExtraAmount),
        'NOT_ENOUGH_ETH'
      );
      return hash;
    }
```

Listing 2.3: Router.sol

However, the `beaconBalance` can be manipulated within the `withdrawValidator` function.

```
  function withdrawValidator(
      uint256 _amount,
      Delegation[] memory _delegations
    ) external nonReentrant whenNotPaused {
      require(config.feature.WithdrawValidator, 'FD'); // FD = Feature Disabled
      require(_amount <= beaconBalance, 'IB'); // IB = Insufficient Balance
      _withdrawBase(_amount, WithdrawType.Validator);
      _updateDelegations(msg.sender, _delegations);
      _setBeaconBalance(beaconBalance - _amount);
      withdrawals.mint(msg.sender, _amount);
  }
```

Listing 2.4: StakeTogether.sol

For example, an attacker could front-run the *executeReport* transaction of the `reportOracle`. They could first deposit into the `StakeTogether` contract and then withdraw all funds via the `withdrawValidator` function, thereby reducing the `beaconBalance`. Consequently, the *executeReport* transaction would revert with the `NOT_ENOUGH_BEACON_BALANCE` error in the `isReadyToExecute` function.

Impact The execution of a report approved by consensus may be blocked.

Suggestion Prevent manipulation of the `beaconBalance` within the `StakeTogether` contract.

2.2.2 Lack of existence check when adding `validators`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `StakeTogether` contract, the role identified by `VALIDATOR_ORACLE_MANAGER_ROLE` can add or remove instances of `validatorOracle` using the `addValidatorOracle` and `removeValidatorOracle` functions, respectively. New `validatorOracle`s are appended to the `validatorsOracle` array, and their corresponding indices are recorded in `validatorsOracleIndices`.

It's crucial to highlight that the `addValidatorOracle` function does not verify the existence of the `validatorOracle` being added. As a result, if a duplicate `ValidatorOracle` is added, the original one cannot be removed from the `validatorsOracle` array, because the `removeValidatorOracle` function will have already deleted the associated index.

```
  function addValidatorOracle(address _account) external onlyRole(VALIDATOR_ORACLE_MANAGER_ROLE)
       {
      _grantRole(VALIDATOR_ORACLE_ROLE, _account);
```

```
        validatorsOracle.push(_account);
        validatorsOracleIndices[_account] = validatorsOracle.length;
        emit AddValidatorOracle(_account);
    }
```

Listing 2.5: StakeTogether.sol

```
    function removeValidatorOracle(address _account) external onlyRole(
        VALIDATOR_ORACLE_MANAGER_ROLE) {
        require(validatorsOracleIndices[_account] > 0, 'NF');

        uint256 index = validatorsOracleIndices[_account] - 1;

        if (index < validatorsOracle.length - 1) {
          address lastAddress = validatorsOracle[validatorsOracle.length - 1];
          validatorsOracle[index] = lastAddress;
          validatorsOracleIndices[lastAddress] = index + 1;
        }

        validatorsOracle.pop();

        delete validatorsOracleIndices[_account];
        _revokeRole(VALIDATOR_ORACLE_ROLE, _account);
        emit RemoveValidatorOracle(_account);
    }
```

Listing 2.6: StakeTogether.sol

Impact Duplicates of `ValidatorOracle` added cannot be removed.

Suggestion Check the existence in the `addValidatorOracle` function.

2.2.3 Ineffective check due to incorrect initialization

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The state variable `nextReportBlock` in the `Router` contract is used for comparing with `block.number` and incrementing `config.reportFrequency` after a report submission. However, it is assigned a value of 1 in the `initialize` function, a value that is significantly lower than the current `block.number`.

```
    function initialize(address _airdrop, address _withdrawals) external initializer {
        __Pausable_init();
        __AccessControl_init();
        __UUPSUpgradeable_init();

        _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);
        _grantRole(ADMIN_ROLE, msg.sender);
        _grantRole(UPGRADER_ROLE, msg.sender);
        _grantRole(ORACLE_REPORT_MANAGER_ROLE, msg.sender);

        version = 1;
```

```
80
81        airdrop = Airdrop(payable(_airdrop));
82        withdrawals = Withdrawals(payable(_withdrawals));
83
84        totalOracles = 0;
85        nextReportBlock = 1;
86        lastConsensusEpoch = 0;
87        lastExecutedEpoch = 0;
88    }
```

Listing 2.7: Router.sol

It’s important to note that the increment here is significantly smaller than the `block.number`.

```
241   function submitReport(
242       uint256 _epoch,
243       Report calldata _report
244     ) external nonReentrant whenNotPaused activeReportOracle {
245       bytes32 hash = isReadyToSubmit(_epoch, _report);
246
247       if (block.number >= nextReportBlock + config.reportFrequency) {
248         nextReportBlock += config.reportFrequency;
249         emit SkipNextReportFrequency(_epoch, nextReportBlock);
250       }
251
252       reports[_epoch][hash].push(msg.sender);
253       reportVotes[_epoch][hash]++;
254       oracleVotes[_epoch][msg.sender] = true;
255
256       if (consensusReport[_epoch] == bytes32(0)) {
257         if (reportVotes[_epoch][hash] >= config.oracleQuorum) {
258           consensusReport[_epoch] = hash;
259           lastConsensusEpoch = _report.epoch;
260           reportDelayBlocks[hash] = block.number;
261           emit ConsensusApprove(_report, hash);
262         } else {
263           emit ConsensusNotReached(_report, hash);
264         }
265       }
266
267       emit SubmitReport(_report, hash);
268   }
```

Listing 2.8: Router.sol

As a result, the check for the verification for `nextReportBlock` on Line 342 will always succeed.

```
340   function isReadyToSubmit(uint256 _epoch, Report calldata _report) public view returns (bytes32
          ) {
341       bytes32 hash = keccak256(abi.encode(_report));
342       require(block.number > nextReportBlock, 'BLOCK_NUMBER_NOT_REACHED');
343       require(totalOracles >= config.minOracleQuorum, 'MIN_ORACLE_QUORUM_NOT_REACHED');
344       require(_report.epoch > lastConsensusEpoch, 'EPOCH_NOT_GREATER_THAN_LAST_CONSENSUS');
345       require(!executedReports[_report.epoch][hash], 'REPORT_ALREADY_EXECUTED');
346       require(!oracleVotes[_epoch][msg.sender], 'ORACLE_ALREADY_VOTED');
```

```
        return hash;
    }
```

Listing 2.9: Router.sol

Impact N/A

Suggestion Revise the logic accordingly.

2.2.4 Lack of existence check when blacklisting the `reportOracles`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `blacklistReportOracle` function in the `Router` contract is used to blacklist a `reportOracle`. However, this function neither verifies the existence of the `reportOracle` nor checks if it's already blacklisted. Furthermore, the function decreases the `totalOracle` count subsequently. Thus, blacklisting an invalid `reportOracle` could potentially lead to an inconsistent `totalOracle` count.

```
function blacklistReportOracle(address _oracle) external onlyRole(ORACLE_SENTINEL_ROLE) {
  oraclesBlacklist[_oracle] = true;
  if (totalOracles > 0) {
    totalOracles--;
  }
  emit BlacklistReportOracle(_oracle);
}
```

Listing 2.10: Router.sol

Impact N/A

Suggestion Check the existence in the `blacklistReportOracle` function.

2.2.5 Potential DoS attack in the consensus process

Severity Low

Status Fixed in `Version 3`

Introduced by `Version 2`

Description The `reportBlock` in the Router contract is used to record the starting block of the current consensus process. The `submitReport` function allows report oracles to submit reports after the `reportBlock` and proceeds to the next `reportBlock` if consensus fails.

The condition in the `submitReport` function (on Line 252) compares the count of unvoted report oracles with the votes required for any submitted report. However, this may introduce a level of unfairness, as the more votes a report received, the fewer votes it requires. A malicious report oracle could exploit this by submitting a fraudulent report to disrupt the consensus when a legitimate report is on the verge of reaching the `oracleQuorum`.

For example, assume a scenario where `totalReportOracles` is 9, `oracleQuorum` is 6, and a valid report `R1` in the current `reportBlock` has received votes from 5 report oracles. A malicious oracle can submit a fake report R2, as the remainingOracles is 4 ($totalReportOracles - totalVotes = 9 - 5 = 4$) and the votes needed is 5 ($oracleQuorum - reportVotesForBlock = 6 - 1 = 5$), resulting in a failed consensus and the reportBlock being forcibly updated. Consequently, the valid report R1 would be compromised.

```
function submitReport(Report calldata _report) external nonReentrant whenNotPaused
    activeReportOracle {
  bytes32 hash = isReadyToSubmit(_report);

  reports[reportBlock][hash].push(msg.sender);
  reportForBlock[reportBlock][msg.sender] = true;
  reportVotesForBlock[reportBlock][hash]++;
  totalVotes[reportBlock]++;

  if (consensusReport[reportBlock] == bytes32(0)) {
    if (totalVotes[reportBlock] >= config.oracleQuorum) {
      if (reportVotesForBlock[reportBlock][hash] >= config.oracleQuorum) {
        consensusReport[reportBlock] = hash;
        lastConsensusBlock = reportBlock;
        reportDelayBlock[reportBlock] = block.number;
        pendingExecution = true;
        emit ConsensusApprove(reportBlock, _report, hash);
      }
    }

    uint remainingOracles = totalReportOracles - totalVotes[reportBlock];
    if ((config.oracleQuorum - reportVotesForBlock[reportBlock][hash]) > remainingOracles) {
      emit ConsensusFail(reportBlock, _report, hash);
      _advanceNextReportBlock();
    }
  }

  emit SubmitReport(_report, hash);
}
```

Listing 2.11: Router.sol

Impact Undermine the fairness of the consensus process.

Suggestion Fix the conditions leading to consensus failure, or promptly blacklist malicious report oracles.

2.3 Additional Recommendation

2.3.1 Add sanity checks for function parameters

Status Fixed in Version 2

Introduced by Version 1

Description For example, in the StakeTogether contract, verify that the _address is not zero within the setFeeAddress function.

```
function setFeeAddress(FeeRole _role, address payable _address) external onlyRole(ADMIN_ROLE)
    {
    feesRole[_role] = _address;
    emit SetFeeAddress(_role, _address);
}
```

Listing 2.12: StakeTogether.sol

Impact N/A

Suggestion Add sanity checks to avoid unexpected behaviors.

2.3.2 Remove duplicate checks

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `Router` contract, the second condition (i.e., `!isReportOracleBlackListed(msg.sender)`) in the `activateReportOracle` modifier is redundant as it's already checked in the first condition (i.e., `isReportOracle(msg.sender)`).

```
modifier activeReportOracle() {
    require(isReportOracle(msg.sender) && !isReportOracleBlackListed(msg.sender), '
        ONLY_ACTIVE_ORACLE');
    _;
}
```

Listing 2.13: Router.sol

Impact N/A

Suggestion Remove duplicate checks.

2.4 Note

2.4.1 Centralization risk

Description Several privileged functions exist within the `StakeTogether` protocol that possess the ability to modify the protocol's state. This introduces a centralization risk, as these privileged accounts can influence the functionality and security of the protocol. Here are a few examples:

- The `DEFAULT_ADMIN_ROLE` has the authority to grant other roles within the protocol.
- The `ADMIN_ROLE` is responsible for modifying relationships between different contracts, such as the `setStakeTogether` function.
- The `UPGRADER_ROLE` facilitates the upgrade of the implementation contract using the UUPS proxy pattern.

Specifically, take the `StakeTogether` and `Router` contracts as examples.

- In the `StakeTogether` contract:
  - The `ADMIN_ROLE` has the following privileges
    1) Modify the configuration settings through the `setConfig` function.

    2) Change the fee addresses for any `feeRole` using the `setFeeAddress` function.
    3) Config the fee ratio and its distributions through the `setFee` function.
  - The `POOL_MANAGER_ROLE` has the privilege to remove a pool by its address using the `removePool` function.
  - The `VALIDATOR_ORACLE_MANAGER_ROLE` has the privilege to add or remove a validator oracle through the `addValidatorOracle` and `removeValidatorOracle` functions.
- In the `Router` contract:
  - The `ADMIN_ROLE` has the following privileges:
    1) Grant/Revoke an `ORACLE_SENTINEL_ROLE` using the `addSentinel`/`removeSentinel` functions.
    2) Update the `lastConsensusEpoch` through the `setLastConsensusEpoch` function.
  - The `ORACLE_SENTINEL_ROLE` can:
    1) Revoke a consensus-approved report by utilizing the `revokeConsensusReport` function.
    2) Blacklist/Unblacklist a report oracle through the `blacklistReportOracle`/`unBlacklistReportOracle` functions.
  - The `ORACLE_REPORT_MANAGER_ROLE` can add or remove a report oracle via the `addReportOracle` and `removeReportOracle` functions.

Feedback from the Project In this case I will update to `DEFAULT_ADMIN_ROLE` for 1 wallet. And this wallet will be on a multisig with time lock actions. This wallet will be responsible for allowing other roles. All actions will have time lock actions with OpenZeppelin Defender.

2.4.2 Ensure the correctness of the configuration

Description Several crucial configuration parameters in the `StakeTogether` and `Router` contracts have been manually set without sufficient constraints. It's important to handle these configurations with care, implementing appropriate validation and security measures to ensure safe operation of the system.

2.4.3 Risk of insufficient report oracles

Description In the `Router` contract, the report oracle plays a vital role in submitting and voting for reports. Once the votes exceed the threshold defined in `config.oracleQuorum`, the submitted report becomes executable at a later date.

The `ORACLE_REPORT_MANAGER_ROLE` can add or remove a report oracle and synchronously update the `_updateQuorum`. Specifically, if the count of active report oracles is insufficient, the `_updateQuorum` function adjusts the `oracleQuorum` to match the `minOracleQuorum` stored in the config. However, this adjustment poses a potential risk. If a submitted report fails to gather the necessary votes for consensus, it could result in asset lockup.

2.4.4 Potential off-chain risks

Description Some features in the `StakeTogether` contract, such as referral and delegations, are implemented off-chain and might affect the airdrop distribution program. Since off-chain logic isn't covered by this audit, its design and correctness can't be guaranteed, which consequently poses some potential risks.

For example, the `_referral` parameter in the `_depositBase` function is used to incentivize users to encourage others to stake. However, `_referral` isn't necessarily different from the depositor themselves, and it's unclear whether the backend service will filter out duplicate events. This introduces a potential risk: a malicious user might inflate their airdrop rewards shares by repeatedly depositing on their own behalf and then immediately withdrawing, thus emitting duplicate events.

```
function _depositBase(address _to, DepositType _depositType, address _referral) private {
    require(config.feature.Deposit, 'FD'); // FD = Feature Disabled
    require(msg.value >= config.minDepositAmount, 'MD'); // MD = Min Deposit

    _resetLimits();

    if (msg.value + totalDeposited > config.depositLimit) {
      emit DepositLimitReached(_to, msg.value);
      revert('DLR');
    }

    _processStakeEntry(_to, msg.value);

    totalDeposited += msg.value;
    emit DepositBase(_to, msg.value, _depositType, _referral);
}
```

Listing 2.14: StakeTogether.sol

Moreover, the `_validateDelegations` function in the `StakeTogether` contract only verifies the delegations provided by users when their shares are non-zero. However, the `_updateDelegations` function always emits an `UpdateDelegations` event subsequently. This introduces a potential vulnerability: a rogue user could zero out their shares first, bypass the `_validateDelegations` function, and emit an invalid event, such as delegating 10000e18 (10,000%) to a pool.

```
function _updateDelegations(address _account, Delegation[] memory _delegations) private {
    _validateDelegations(_account, _delegations);
    emit UpdateDelegations(_account, _delegations);
}
```

Listing 2.15: StakeTogether.sol

```
function _validateDelegations(address _account, Delegation[] memory _delegations) private view
    {
    if (shares[_account] > 0) {
      require(_delegations.length <= config.maxDelegations, 'MD'); // MD = Max Delegations
      uint256 delegationShares = 0;
      for (uint i = 0; i < _delegations.length; i++) {
        require(pools[_delegations[i].pool], 'PNF'); // PNF = Pool Not Found
        delegationShares += _delegations[i].percentage;
      }
      require(delegationShares == 1 ether, 'IPS'); // IPS = Invalid Percentage Sum
    }
}
```

Listing 2.16: StakeTogether.sol

Feedback from the Project I believe items 2.15 and 2.16 were already fixed in version 3; this behavior can't occur because of the nature of our product.