stake.link
Security Review

OCTOBER, 2024

www.chaindefenders.xyz
https://x.com/ChDefendersEth

Lead Auditors

PeterSR

0x539.eth

Table of Contents

Protocol Summary

stake.link is a first-of-its-kind liquid delegated staking platform delivering DeFi composability for Chainlink Staking. Built by premier Chainlink ecosystem developer LinkPool, powered by Chainlink node operators, and governed by the stake.link DAO, stake.link's extensible architecture is purpose-built to support Chainlink Staking and to extend participation in the Chainlink Network.

Disclaimer

The ChainDefenders team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

Likelihood/Impact	High	Medium	Low
High	H	H/M	M
Medium	H/M	M	M/L
Low	M	M/L	L

Audit Details

Scope

Id	Files in scope
1	LSTRewardsSplitter.sol
2	LSTRewardsSplitterController.sol
3	PriorityPool.sol
4	WithdrawalPool.sol
5	StakingPool.sol
6	Vault.sol
7	VaultControllerStrategy.sol
8	CommunityVault.sol
9	CommunityVCS.sol
10	FundFlowController.sol
11	OperatorStakingPool.sol
12	OperatorVault.sol
13	OperatorVCS.sol

Roles

Id	Roles
1	Staker
2	Queued Staker
3	Queued Withdrawer
4	Operator
5	Operator Rewards Receiver
6	Owner

Executive Summary

Issues found

Severity	Count	Description
High	0	Critical vulnerabilities
Medium	3	Significant risks
Low	0	Minor issues with low impact
Informational	0	Best practices or suggestions
Gas	0	Optimization opportunities

Findings

Medium

Mid 01 Malicious User Can DoS Withdraw

Summary

The `WithdrawalPool` contains an optimization for managing withdrawal batches. The function `updateWithdrawalBatchIdCutoff` is responsible for updating the `withdrawalBatchIdCutoff` value. However, if a user has an incomplete withdrawal for a specific batch, this update cannot proceed until the user completes their withdrawal. A malicious user could exploit this situation to indefinitely prevent the update, leading to a denial-of-service (DoS) condition

Vulnerability Details

Look at the following scenario:

1. A user submits a withdrawal request, which is assigned an ID of 130 and is associated with batch 35.
2. The withdrawalBatchIdCutoff remains at 35 until the user's withdrawal request is finalized.
3. If the withdrawal never completes, the getBatchIds function will iteratively process all batches for every withdrawal ID, potentially causing significant delays or halting other processes.

```
function getBatchIds(uint256[] memory _withdrawalIds) public view
    returns (uint256[] memory) {
        uint256[] memory batchIds = new uint256[](_withdrawalIds.
    length);

        for (uint256 i = 0; i < _withdrawalIds.length; ++i) {
            uint256 batchId;
            uint256 withdrawalId = _withdrawalIds[i];

            for (uint256 j = withdrawalBatchIdCutoff; j <
    withdrawalBatches.length; ++j) {
                uint256 indexOfLastWithdrawal = withdrawalBatches[j].
    indexOfLastWithdrawal;

                if (withdrawalId ⩽ indexOfLastWithdrawal) {
                    batchId = j;
                    break;
                }
            }

            batchIds[i] = batchId;
        }

        return batchIds;
     }
```

```
function updateWithdrawalBatchIdCutoff() external { // audit-mid
    possible dos
        uint256 numWithdrawals = queuedWithdrawals.length;
        uint256 newWithdrawalIdCutoff = withdrawalIdCutoff;

        // find the first withdrawal that has funds remaining
        for (uint256 i = newWithdrawalIdCutoff; i < numWithdrawals; ++
    i) {
            newWithdrawalIdCutoff = i;

            Withdrawal memory withdrawal = queuedWithdrawals[i];
            if (withdrawal.sharesRemaining ≠ 0 || withdrawal.
    partiallyWithdrawableAmount ≠ 0) {
                break;
            }
        } // this depends does the withdraw is already excuted

        uint256 numBatches = withdrawalBatches.length;
        uint256 newWithdrawalBatchIdCutoff = withdrawalBatchIdCutoff;

        // find the last batch where all withdrawals have no funds
```

```
    remaining
19          for (uint256 i = newWithdrawalBatchIdCutoff; i < numBatches;
    ++i) {
20              if (withdrawalBatches[i].indexOfLastWithdrawal >=
    newWithdrawalIdCutoff) {
21                  break;
22              }
23
24              newWithdrawalBatchIdCutoff = i;
25          }
26
27          withdrawalIdCutoff = uint128(newWithdrawalIdCutoff);
28          withdrawalBatchIdCutoff = uint128(newWithdrawalBatchIdCutoff);
29      }
```

Impact

This vulnerability could allow a malicious user to block the withdrawal process for other users by preventing the system from advancing the withdrawalBatchIdCutoff. This can lead to a DoS attack where no new withdrawals can be processed until the malicious withdrawal is completed, affecting the overall availability of the system.

Tools Used

Manual review

Recommendations

To mitigate this issue, consider implementing a timeout or fallback mechanism where an admin can finalize or cancel old withdrawal requests after a certain period of inactivity.

Mid 02 VaultMapping Is Not Correctly Updated

Summary

When a vault is added to the `OperatorVCS` the function `addVault` is used:

```
function addVault(
    address _operator,
    address _rewardsReceiver,
    address _pfAlertsController
) external onlyOwner {
    bytes memory data = abi.encodeWithSignature(
        "initialize(address,address,address,address,address,address,
    address)",
        address(token),
        address(this),
        address(stakeController),
        stakeController.getRewardVault(),
        _pfAlertsController,
        _operator,
        _rewardsReceiver
    );
    _deployVault(data);
    vaultMapping[address(vaults[vaults.length - 1])] = true;
    emit VaultAdded(_operator);
}
```

We can see that the `vaultMapping` for the concrete vault is set to true. When removing a vault the function `queueVaultRemoval` is used and then the function `removeVault`:

```
function queueVaultRemoval(uint256 _index) external {
    address vault = address(vaults[_index]);

    if (!IVault(vault).isRemoved()) revert OperatorNotRemoved();
    for (uint256 i = 0; i < vaultsToRemove.length; ++i) {
        if (vaultsToRemove[i] == vault) revert
    VaultRemovalAlreadyQueued();
    }

    vaultsToRemove.push(address(vaults[_index]));

    // update group accounting if vault is part of a group
    if (_index < globalVaultState.depositIndex) {
        uint256 group = _index % globalVaultState.numVaultGroups;
        uint256[] memory groups = new uint256[](1);
        groups[0] = group;
        fundFlowController.updateOperatorVaultGroupAccounting(groups);

        // if possiible, remove vault right away
        if (vaults[_index].claimPeriodActive()) {
            removeVault(vaultsToRemove.length - 1);
        }
```

```
    }
}

function removeVault(uint256 _queueIndex) public {
    address vault = vaultsToRemove[_queueIndex];

    vaultsToRemove[_queueIndex] = vaultsToRemove[vaultsToRemove.length - 1];
    vaultsToRemove.pop();

    _updateStrategyRewards();
    (uint256 principalWithdrawn, uint256 rewardsWithdrawn) = IOperatorVault(vault).exitVault();

    totalDeposits -= principalWithdrawn + rewardsWithdrawn;
    totalPrincipalDeposits -= principalWithdrawn;

    uint256 numVaults = vaults.length;
    uint256 index;
    for (uint256 i = 0; i < numVaults; ++i) {
        if (address(vaults[i]) == vault) {
            index = i;
            break;
        }
    }
    for (uint256 i = index; i < numVaults - 1; ++i) {
        vaults[i] = vaults[i + 1];
    }
    vaults.pop();

    token.safeTransfer(address(stakingPool), token.balanceOf(address(this)));
}
```

However, we can see that in both function this `vaultMapping` that was set to true is never set to false. The mapping is used in `withdrawOperatorRewards`:

```
function withdrawOperatorRewards(
    address _receiver,
    uint256 _amount
) external returns (uint256) {
    if (!vaultMapping[msg.sender]) revert SenderNotAuthorized();

    IERC20Upgradeable lsdToken = IERC20Upgradeable(address(stakingPool));
    uint256 withdrawableRewards = lsdToken.balanceOf(address(this));
    uint256 amountToWithdraw = _amount > withdrawableRewards ?
```

```
        withdrawableRewards : _amount;

        unclaimedOperatorRewards -= amountToWithdraw;
        lsdToken.safeTransfer(_receiver, amountToWithdraw);

        return amountToWithdraw;
    }
```

This means that even if a vault is “removed” it will still be able to withdraw operator rewards.

Vulnerability Details

Let’s have the following scenario:

1. Vault A is added via `addVault`.
2. Vault A is removed via `queueVaultRemoval` -> `removeVault`.
3. However, Vault A is able to call `withdrawOperatorRewards` setting `_receiver` and `_amount` and withdrawing the rewards even though it should not be able to.

Impact

Such an issue has a critical impact to the security of the `OperatorVCS` contract as no one is able to stop a vault from operating even after removing it.

Tools Used

Manual Review

Recommendations

Update the `vaultMapping` to reflect that a given vault was removed:

```
function removeVault(uint256 _queueIndex) public {
    address vault = vaultsToRemove[_queueIndex];
+   vaultMapping[vault] = false;
    vaultsToRemove[_queueIndex] = vaultsToRemove[vaultsToRemove.length
    - 1];
    vaultsToRemove.pop();
```

```
6
7       _updateStrategyRewards();
8       (uint256 principalWithdrawn, uint256 rewardsWithdrawn) =
        IOperatorVault(vault).exitVault();
9
10      totalDeposits -= principalWithdrawn + rewardsWithdrawn;
11      totalPrincipalDeposits -= principalWithdrawn;
12
13      uint256 numVaults = vaults.length;
14      uint256 index;
15      for (uint256 i = 0; i < numVaults; ++i) {
16          if (address(vaults[i]) == vault) {
17              index = i;
18              break;
19          }
20      }
21      for (uint256 i = index; i < numVaults - 1; ++i) {
22          vaults[i] = vaults[i + 1];
23      }
24      vaults.pop();
25
26      token.safeTransfer(address(stakingPool), token.balanceOf(address(
        this)));
27  }
```

Mid 03 Remove Splitter Will Not Work In Certain Situations

Summary

The `LSTRewardsSplitterController` :: `removeSplitter` function is responsible for removing a splitter and withdrawing all associated funds. The funds are transferred to an account specified as a parameter. Before this happens, the function calls `splitRewards` on the splitter to distribute any outstanding rewards to fee receivers. However, there is a flaw in this process: after splitting rewards, the function attempts to withdraw the original balance (before the split) instead of the updated balance, which will likely cause the transaction to revert if the balance has changed.

Vulnerability Details

Consider the following scenario to illustrate the issue:

1. A splitter has a balance of 100 tokens.
2. After splitting the rewards, part of the balance (e.g., 10 tokens) is distributed to fee receivers, leaving a remaining balance of 90 tokens.
3. The `removeSplitter` function, however, tries to withdraw the original balance of 100 tokens, which exceeds the actual available balance of 90 tokens. This mismatch causes a failure or revert when the withdrawal is attempted.

This issue arises due to the fact that the function does not update the balance after calling `splitRewards`. It attempts to withdraw the pre-split balance, which is incorrect.

The vulnerability can be observed in the code snippet below:

```
function removeSplitter(address _account) external onlyOwner {
    ILSTRewardsSplitter splitter = splitters[_account];
    if (address(splitter) == address(0)) revert SplitterNotFound();

    uint256 balance = IERC20(lst).balanceOf(address(splitter));  // Original balance is captured
    uint256 principalDeposits = splitter.principalDeposits();
    if (balance != 0) {
        if (balance != principalDeposits) splitter.splitRewards();  // Split rewards, changing the balance
        splitter.withdraw(balance, _account);  // Attempts to withdraw the old balance, likely causing a revert
}
```

In the above function, the balance is initially fetched before the rewards are split. This results in an outdated balance being passed to the `withdraw` function, leading to a revert if the balance has changed due to the rewards distribution.

Impact

This vulnerability can cause the `removeSplitter` function to revert when attempting to withdraw funds, preventing the successful removal of the splitter and the withdrawal of remaining funds. This could halt operations or cause disruptions, especially if multiple splitters are being managed within the system.

Tools Used

Manual review

Recommendations

To fix this issue, the balance should be updated after the `splitRewards` call to ensure that the correct amount is withdrawn. Below is the recommended change to the `removeSplitter` function:

```
function removeSplitter(address _account) external onlyOwner {
        ILSTRewardsSplitter splitter = splitters[_account];
        if (address(splitter) == address(0)) revert SplitterNotFound();

        uint256 balance = IERC20(lst).balanceOf(address(splitter));
    // Initial balance before rewards split
        uint256 principalDeposits = splitter.principalDeposits();
        if (balance != 0) {
            if (balance != principalDeposits) splitter.splitRewards();
      // Split rewards, changing the balance
+           balance = IERC20(lst).balanceOf(address(splitter));  //
    Update balance after rewards are split
            splitter.withdraw(balance, _account);  // Withdraw updated
     balance
        }
```

By re-fetching the balance after the `splitRewards` function call, the correct (post-split) balance will be withdrawn, preventing any reverts and ensuring smooth operation.