Stake.Link Staking Proxy Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

0kage

January 20, 2025

Contents

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse 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.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

Stake.link is a liquid staking protocol that allows users to stake their LINK token and earn staking rewards. Current audit was focused on the `StakingProxy`contract that enables stakers to deposit tokens and earn rewards without directly interacting with the priority pool and other core contracts.

5 Audit Scope

Following files were part of the current audit scope.

- StakingProxy.sol
- PriorityPool.sol
- WithdrawalPool.sol
- SDLPool.sol
- RewardsPoolController.sol
- RewardsPool.sol
- RewardsPoolTimeBased.sol
- RewardsPoolWSD.sol

6 Executive Summary

Over the course of 8 days, the Cyfrin team conducted an audit on the Stake.Link Staking Proxy smart contracts provided by Stake.Link. In this period, a total of 3 issues were found.

Summary

Project Name	Stake.Link Staking Proxy
Repository	contracts
Commit	2c5bc0161b40...
Audit Timeline	Jan 7th - Jan 16th
Methods	Manual Review, Stateful Fuzzing

Issues Found

Critical Risk	1
High Risk	0
Medium Risk	0
Low Risk	2
Informational	0
Gas Optimizations	0
Total Issues	3

Summary of Findings

[C-1] Instant withdrawals in priority pool can result in loss of funds for Staking-Proxy contract	Resolved
[L-1] Unrestricted reSDL token deposits with privileged withdrawals can lead to accidental loss of reSDL tokens	Acknowledged
[L-2] Storage collision risk in UUPS upgradeable `StakingProxy` due to missing storage gap	Acknowledged

7 Findings

7.1 Critical Risk

7.1.1 Instant withdrawals in priority pool can result in loss of funds for StakingProxy contract

Description: When instant withdrawals are enabled in the priority pool, `staker` can permanently lose funds when withdrawing through the `StakingProxy` contract. The issue occurs because the withdrawn amount is not properly updated in the priority pool's `_withdraw` function during instant withdrawals, causing the tokens to be stuck in the Priority Pool while users lose their LSTs.

`PriorityPool::_withdraw`

```
function _withdraw(
    address _account,
    uint256 _amount,
    bool _shouldQueueWithdrawal,
    bool _shouldRevertOnZero,
    bytes[] memory _data
) internal returns (uint256) {
    if (poolStatus == PoolStatus.CLOSED) revert WithdrawalsDisabled();

    uint256 toWithdraw = _amount;
    uint256 withdrawn;
    uint256 queued;

    if (totalQueued != 0) {
        uint256 toWithdrawFromQueue = toWithdraw <= totalQueued ? toWithdraw : totalQueued;

        totalQueued -= toWithdrawFromQueue;
        depositsSinceLastUpdate += toWithdrawFromQueue;
        sharesSinceLastUpdate += stakingPool.getSharesByStake(toWithdrawFromQueue);
        toWithdraw -= toWithdrawFromQueue;
        withdrawn = toWithdrawFromQueue; // -----> @audit withdrawn is set here
    }

    if (
        toWithdraw != 0 &&
        allowInstantWithdrawals &&
        withdrawalPool.getTotalQueuedWithdrawals() == 0
    ) {
        uint256 toWithdrawFromPool = MathUpgradeable.min(stakingPool.canWithdraw(), toWithdraw);
        if (toWithdrawFromPool != 0) {
            stakingPool.withdraw(address(this), address(this), toWithdrawFromPool, _data);
            toWithdraw -= toWithdrawFromPool; // -----> @audit BUG withdrawn is not updated here
        }
    }
    // ... rest of the function
}
```

When processing instant withdrawals, the function fails to update the withdrawn variable after successfully withdrawing tokens from the staking pool. This leads to the following sequence:

1. Staker initiates withdrawal through `StakingProxy::withdraw`
2. `StakingProxy` burns LSTs
3. `PriorityPool` receives underlying tokens from Staking Pool
4. But `PriorityPool` doesn't transfer tokens because `withdrawn` wasn't updated
5. Tokens remain stuck in `PriorityPool` while `StakingProxy` loses access to its liquid staking tokens

Impact: `StakingProxy` permanently loses access to its liquid staking tokens when attempting instant withdrawals

Proof of Concept: Copy the following test into `staking-proxy.test.ts`

```
it('instant withdrawals from staking pool are not transferred to staker', async () => {
   const { stakingProxy, stakingPool, priorityPool, signers, token, strategy, accounts } = await
   ↪  loadFixture(deployFixture)

   // Enable instant withdrawals
   await priorityPool.setAllowInstantWithdrawals(true)

   // Deposit initial amount
   await token.approve(stakingProxy.target, toEther(1000))
   await stakingProxy.deposit(toEther(1000), ['0x'])

   // Setup for withdrawals
   await strategy.setMaxDeposits(toEther(2000))
   await strategy.setMinDeposits(0)

   // Track all relevant balances before withdrawal
   const preTokenBalance = await token.balanceOf(stakingProxy.target)
   const preLSTBalance = await stakingPool.balanceOf(stakingProxy.target)

   const prePPBalance = await token.balanceOf(priorityPool.target)

   const withdrawAmount = toEther(500)

   console.log('=== Before Withdrawal ===')
   console.log('Initial LST Balance - Proxy contract:', fromEther(preLSTBalance))
   console.log('Initial Token Balance - Poxy contract:', fromEther(preTokenBalance))


   // Perform withdrawal
   await stakingProxy.withdraw(
       withdrawAmount,
       0,
       0,
       [],
       [],
       [],
       ['0x']
   )

   // Check all balances after withdrawal
   const postTokenBalance = await token.balanceOf(stakingProxy.target)
   const postPPBalance = await token.balanceOf(priorityPool.target)
   const postLSTBalance = await stakingPool.balanceOf(stakingProxy.target)

   console.log('=== After Withdrawal ===')
   console.log('Priority Pool - token balance change:', fromEther(postPPBalance - prePPBalance))
   console.log('Staking Proxy - token balance change:', fromEther(postTokenBalance - preTokenBalance))
   console.log('Staking Proxy - LST balance change:', fromEther(postLSTBalance - preLSTBalance))

   const lstsRedeemed = fromEther(preLSTBalance - postLSTBalance)

   // Assertions

   // 1. Staking Proxy has redeeemed all his LSTs
   assert.equal(
     lstsRedeemed,
       500,
       "Staker redeemed 500 LSTs"
   )
```

```
    // 2. But staking proxy doesn't receive underlying tokens
    assert.equal(
      fromEther(postTokenBalance - preTokenBalance),
        0,
        "Staking Proxy didn't receive any tokens despite losing LSTs"
    )

    // 3. The tokens are stuck in Priority Pool
    assert.equal(
        fromEther(postPPBalance- prePPBalance),
        500,
        "Priority Pool is holding the withdrawn tokens"
    )
  })
```

Recommended Mitigation: Consider updating the `withdrawn` variable when processing instant withdrawals in `PriorityPool::_withdraw`

Stake.Link: Fixed in commit 5f3d282

Cyfrin: Verified.

7.2 Low Risk

7.2.1 Unrestricted reSDL token deposits with privileged withdrawals can lead to accidental loss of reSDL tokens

Description: The `StakingProxy` contract implements ERC721 receiver functionality allowing it to receive reSDL tokens (ERC721) from any address. However, only the contract owner has the ability to withdraw these tokens.

This creates a risk where user owned reSDL tokens can get stuck if sent to the proxy accidentally or without understanding the withdrawal restrictions.

`StakingProxy.sol`

```
// ----> @audit Anyone can transfer reSDL tokens to the proxy
function onERC721Received(address, address, uint256, bytes calldata) external returns (bytes4) {
    return this.onERC721Received.selector;
}

// ----> @audit Only owner can withdraw reSDL tokens
function withdrawRESDLToken(uint256 _tokenId, address _receiver) external onlyOwner {
    if (sdlPool.ownerOf(_tokenId) != address(this)) revert InvalidTokenId();
    IERC721(address(sdlPool)).safeTransferFrom(address(this), _receiver, _tokenId);
}
```

The reSDL tokens represent time-locked SDL staking positions that earn rewards. While the proxy's ability to hold reSDL tokens is an intended functionality for reward earning purposes, the unrestricted acceptance of transfers combined with privileged withdrawals creates unnecessary risk.

Impact: Any user accidentally transferring their reSDL tokens to the proxy has no direct way of recovering them without manual intervention of protocol team.

Recommended Mitigation: Consider gating the `onERC721Received` function to only accept transfers from authorized addresses that can be configured in the StakingProxy contract.

Stake.link: Acknowledged.

Cyfrin: Acknowledged.

7.2.2 Storage collision risk in UUPS upgradeable `StakingProxy` due to missing storage gap

Description: `StakingProxy` contract inherits from `UUPSUpgradeable` and `OwnableUpgradeable` but does not implement storage gaps to protect against storage collisions during upgrades.

`StakingProxy` is intended to be used by third parties/DAOs. It is possible that this contract gets inherited by external contracts with their own storage variables. In such a scenario, adding new storage variables to `StakingProxy` during an upgrade can shift storage slots and cause serious storage collision risks.

`StakingProxy.sol`

```
contract StakingProxy is UUPSUpgradeable, OwnableUpgradeable {
    // address of asset token
    IERC20Upgradeable public token;
    // address of liquid staking token
    IStakingPool public lst;
    // address of priority pool
    IPriorityPool public priorityPool;
    // address of withdrawal pool
    IWithdrawalPool public withdrawalPool;
    // address of SDL pool
    ISDLPool public sdlPool;
    // address authorized to deposit/withdraw asset tokens
    address public staker; // ---> @audit missing storage slots
}
```

Impact: Potential storage collision can corrupt data and cause contract to malfunction.

Recommended Mitigation: Consider adding a storage gap at the end of the contract to reserve slots for future inherited contract variable. A slot size of 50 is the OpenZeppelin's recommended pattern for upgradeable contracts.

Stake.link: Acknowledged.

Cyfrin: Acknowledged.