stake.link Audit Report

Prepared by Cyfrin
Version 2.0

Lead Auditors

Hans

Assisting Auditors

0kage

August 25, 2023

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, built initially for the Chainlink Network with upcoming support for Ethereum 2.0's beacon chain. stake.link consists of the highest-quality node operators and validators to ensure a seamless and secure way to put tokens to stake.

5 Executive Summary

Over the course of 6 days, the Cyfrin team conducted an audit on the stake.link smart contracts provided by LinkPool. In this period, a total of 6 issues were found.

`StakingQueue.sol` of stake.link repo. Commit Hash: ef7a7f4dbdbe50a435609ed9b98c2d254fec1963

Summary

Project Name	stake.link
Repository	contracts
Commit	ef7a7f4dbdbe...
Audit Timeline	Aug 14th - Aug 21st
Methods	Manual Review

Issues Found

Critical Risk	0
High Risk	0
Medium Risk	2
Low Risk	1
Informational	3
Gas Optimizations	0
Total Issues	6

6 Findings

6.1 Medium Risk

6.1.1 The off-chain mechanism must be ensured to work in a correct order strictly

Severity: Medium

Description: The `PriorityPool` contract relies on the distribution oracle for accounting and the accounting calculation is done off-chain.

According to the communication with the protocol team, the correct workflow for queued deposits can be described as below:

- Whenever there is a new room for deposit in the staking pool, the function `depositQueuedTokens` is called.
- The `PriorityPool` contract is paused by calling `pauseForUpdate()`.
- Accounting calculations happen off-chain using the function `getAccountData()` and `getDepositsSinceLastUpdate()(depositsSinceLastUpdate)` variable to compose the latest Merkle tree.
- The distribution oracle calls the function `updateDistribution()` and this will resume the `PriorityPool`.

The only purpose of pausing the queue contract is to prevent unqueue until the accounting status are updated. Through an analysis we found that the off-chain mechanism MUST follow the order very strictly or else user funds can be stolen. While we acknowledge that the protocol team will ensure it, we decided to keep this finding as a medium risk because we can not verify the off-chain mechanism.

Impact: If the off-chain mechanism occurs in a wrong order by any chance, user funds can be stolen. Given the likelihood is low, we evaluate the impact to be Medium.

Proof of Concept: The below test case shows the attack scenario.

```
it('Cyfrin: off-chain mechanism in an incorrect order can lead to user funds being stolen', async ()
↪  => {
  // try deposit 1500 while the capacity is 1000
  await strategy.setMaxDeposits(toEther(1000))
  await sq.connect(signers[1]).deposit(toEther(1500), true)

  // 500 ether is queued for accounts[1]
  assert.equal(fromEther(await stakingPool.balanceOf(accounts[1])), 1000)
  assert.equal(fromEther(await sq.getQueuedTokens(accounts[1], 0)), 500)
  assert.equal(fromEther(await token.balanceOf(accounts[1])), 8500)

  // unqueue 500 ether should work while no updateDistribution was called
  await sq.connect(signers[1]).unqueueTokens(0, 0, [], toEther(500))
  assert.equal(fromEther(await sq.getQueuedTokens(accounts[1], 0)), 0)
  assert.equal(fromEther(await token.balanceOf(accounts[1])), 9000)

  // deposit again
  await sq.connect(signers[1]).deposit(toEther(500), true)
  assert.equal(fromEther(await token.balanceOf(accounts[1])), 8500)

  // victim deposits 500 ether and it will be queued
  await sq.connect(signers[2]).deposit(toEther(500), true)
  assert.equal(fromEther(await sq.totalQueued()), 1000)

  // max deposit has increased to 1500
  await strategy.setMaxDeposits(toEther(1500))

  // user sees that his queued tokens 500 can be deposited and call depositQueuedTokens
  // this will deposit the 500 ether in the queue
  await sq.connect(signers[1]).depositQueuedTokens()
```

```
    // Correct off-chain mechanism: pauseForUpdate -> getAccountData -> updateDistribution
    // Let us see what happens if getAccountData is called before pauseForUpdate

    // await sq.pauseForUpdate()

    // check account data
    var a_data = await sq.getAccountData()
    assert.equal(ethers.utils.formatEther(a_data[2][1]), "500.0")
    assert.equal(ethers.utils.formatEther(a_data[2][2]), "500.0")

    // user calls unqueueTokens to get his 500 ether back
    // this is possible because the queue contract is not paused
    await sq.connect(signers[1]).unqueueTokens(0, 0, [], toEther(500))

    // pauseForUpdate is called at a wrong order
    await sq.pauseForUpdate()

    // at this point user has 1000 ether staked and 9000 ether in his wallet
    assert.equal(fromEther(await token.balanceOf(accounts[1])), 9000)
    assert.equal(fromEther(await stakingPool.balanceOf(accounts[1])), 1000)

    // now updateDistribution is called with the wrong data
    let data = [
      [ethers.constants.AddressZero, toEther(0), toEther(0)],
      [accounts[1], toEther(500), toEther(500)],
    ]
    let tree = StandardMerkleTree.of(data, ['address', 'uint256', 'uint256'])

    await sq.updateDistribution(
      tree.root,
      ethers.utils.formatBytes32String('ipfs'),
      toEther(500),
      toEther(500)
    )

    // at this point user claims his LSD tokens
    await sq.connect(signers[1]).claimLSDTokens(toEther(500), toEther(500), tree.getProof(1))

    // at this point user has 1500 ether staked and 9000 ether in his wallet
    assert.equal(fromEther(await token.balanceOf(accounts[1])), 9000)
    assert.equal(fromEther(await stakingPool.balanceOf(accounts[1])), 1500)
  })
```

Recommended Mitigation: Consider to force pause the contract at the end of the function `_depositQueuedTokens`.

Client: Acknowledged. The protocol team will ensure the correct order of the off-chain mechanism.

Cyfrin: Acknowledged.

6.1.2 User's funds are locked temporarily in the PriorityPool contract

Severity: Medium

Description: The protocol intended to utilize the deposit queue for withdrawal to minimize the stake/unstake interaction with the staking pool. When a user wants to withdraw, they are supposed to call the function `PriorityPool::withdraw()` with the desired amount as a parameter.

```
function withdraw(uint256 _amount) external {//@audit-info LSD token
    if (_amount == 0) revert InvalidAmount();
    IERC20Upgradeable(address(stakingPool)).safeTransferFrom(msg.sender, address(this),
    ↪ _amount);//@audit-info get LSD token from the user
    _withdraw(msg.sender, _amount);
}
```

As we can see in the implementation, the protocol pulls the `_amount` of LSD tokens from the user first and then calls `_withdraw()` where the actual withdrawal utilizing the queue is processed.

```
function _withdraw(address _account, uint256 _amount) internal {
    if (poolStatus == PoolStatus.CLOSED) revert WithdrawalsDisabled();

    uint256 toWithdrawFromQueue = _amount <= totalQueued ? _amount : totalQueued;//@audit-info if the
    ↪ queue is not empty, we use that first
    uint256 toWithdrawFromPool = _amount - toWithdrawFromQueue;

    if (toWithdrawFromQueue != 0) {
        totalQueued -= toWithdrawFromQueue;
        depositsSinceLastUpdate += toWithdrawFromQueue;//@audit-info regard this as a deposit via the
        ↪ queue
    }

    if (toWithdrawFromPool != 0) {
        stakingPool.withdraw(address(this), address(this), toWithdrawFromPool);//@audit-info withdraw
        ↪ from pool into this contract
    }

    //@audit-warning at this point, toWithdrawFromQueue of LSD tokens remain in this contract!

    token.safeTransfer(_account, _amount);//@audit-info
    emit Withdraw(_account, toWithdrawFromPool, toWithdrawFromQueue);
}
```

But looking in the function `_withdraw()`, only `toWithdrawFromPool` amount of LSD tokens are withdrawn (burn) from the staking pool and `toWithdrawFromQueue` amount of LSD tokens remain in the `PriorityPool` contract. On the other hand, the contract tracks the queued amount for users by the mapping `accountQueuedTokens` and this leads to possible mismatch in the accounting. Due to this mismatch, a user's LSD tokens can be locked in the `PriorityPool` contract while the user sees his queued amount (`getQueuedTokens()`) is positive. Users can claim the locked LSD tokens once the function `updateDistribution` is called. Through the communication with the protocol team, it is understood that `updateDistribution` is expected to be called *probably every 1-2 days unless there were any new deposits into the staking pool*. So it means user's funds can be locked temporarily in the contract which is unfair for the user.

Impact: User's LSD tokens can be locked temporarily in the PriorityPool contract

Proof of Concept:

```
it('Cyfrin: user funds can be locked temporarily', async () => {
  // try deposit 1500 while the capacity is 1000
  await strategy.setMaxDeposits(toEther(1000))
  await sq.connect(signers[1]).deposit(toEther(1500), true)

  // 500 ether is queued for accounts[1]
```

```
    assert.equal(fromEther(await stakingPool.balanceOf(accounts[1])), 1000)
    assert.equal(fromEther(await sq.getQueuedTokens(accounts[1], 0)), 500)
    assert.equal(fromEther(await token.balanceOf(accounts[1])), 8500)
    assert.equal(fromEther(await sq.totalQueued()), 500)
    assert.equal(fromEther(await stakingPool.balanceOf(sq.address)), 0)

    // at this point user calls withdraw (maybe by mistake?)
    // withdraw swipes from the queue and the deposit room stays at zero
    await stakingPool.connect(signers[1]).approve(sq.address, toEther(500))
    await sq.connect(signers[1]).withdraw(toEther(500))

    // at this point getQueueTokens[accounts[1]] does not change but the queue is empty
    // user will think his queue position did not change and he can simply unqueue
    assert.equal(fromEther(await stakingPool.balanceOf(accounts[1])), 500)
    assert.equal(fromEther(await sq.getQueuedTokens(accounts[1], 0)), 500)
    assert.equal(fromEther(await token.balanceOf(accounts[1])), 9000)
    assert.equal(fromEther(await sq.totalQueued()), 0)
    // NOTE: at this point 500 ethers of LSD tokens are locked in the queue contract
    assert.equal(fromEther(await stakingPool.balanceOf(sq.address)), 500)

    // but unqueueTokens fails because actual totalQueued is zero
    await expect(sq.connect(signers[1]).unqueueTokens(0, 0, [], toEther(500))).to.be.revertedWith(
      'InsufficientQueuedTokens()'
    )

    // user's LSD tokens are still locked in the queue contract
    await stakingPool.connect(signers[1]).approve(sq.address, toEther(500))
    await sq.connect(signers[1]).withdraw(toEther(500))
    assert.equal(fromEther(await stakingPool.balanceOf(accounts[1])), 0)
    assert.equal(fromEther(await sq.getQueuedTokens(accounts[1], 0)), 500)
    assert.equal(fromEther(await token.balanceOf(accounts[1])), 9500)
    assert.equal(fromEther(await sq.totalQueued()), 0)
    assert.equal(fromEther(await stakingPool.balanceOf(sq.address)), 500)

    // user might try withdraw again but it will revert because user does not have any LSD tokens
    await stakingPool.connect(signers[1]).approve(sq.address, toEther(500))
    await expect(sq.connect(signers[1]).withdraw(toEther(500))).to.be.revertedWith(
      'Transfer amount exceeds balance'
    )

    // in conclusion, user's LSD tokens are locked in the queue contract and he cannot withdraw them
    // it is worth noting that the locked LSD tokens are credited once updateDistribution is called
    // so the lock is temporary
  })
```

Recommended Mitigation: Consider add a feature to allow users to withdraw LSD tokens from the contract directly.

Client: Fixed in this PR.

Cyrfin: Verified.

6.2 Low Risk

6.2.1 Do not use deprecated library functions

```
File: PriorityPool.sol

103:        token.safeApprove(_stakingPool, type(uint256).max);
```

Client: Fixed in this PR.

Cyfrin: Verified.

6.3 Informational

6.3.1 Unnecessary event emissions

`PriorityPool::setPoolStatusClosed` does not check if pool status is already `CLOSED` and emits `SetPoolStatus` event. Avoid event emission if the pool status is already closed. Avoid this. The same applies to the function `setPoolStatus` as well.

Client: Fixed in this PR.

Cyfrin: Verified.

6.3.2 Missing checks for `address(0)` when assigning values to address state variables

```
File: PriorityPool.sol

399:        distributionOracle = _distributionOracle;
```

Client: Acknowledged.

Cyfrin: Acknowledged.

6.3.3 Functions not used internally could be marked external

```
File: PriorityPool.sol

89:     function initialize(

278:     function depositQueuedTokens() public {
```

Client: Acknowledged.

Cyfrin: Acknowledged.

7 Additional Comments

- The protocol relies on some off-chain mechanism for accounting and we could not verify them.
- The contract name was changed from `StakingQueue` to `PriorityPool` during the mitigation.