Stake.Link PR152 LINKMigrator Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

Immeas

holydevoti0n

June 4, 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 the first delegated liquid staking protocol built for Chainlink Staking. It allows anyone to provide LINK as collateral and earn a share of rewards from the most reliable and high-performing Chainlink node operators.

4.1 Key Features

This audit introduced a new component to the stake.link ecosystem: `LINKMigrator`. This feature enables existing stakers in the Chainlink community vault to migrate their positions to stake.link without waiting in the `PriorityPool` queue. This is possible because the staker relinquishes their existing position in the community vault in favor of a new one in stake.link.

4.2 Actors and Roles

1. **Protocol Owner**:
    - Deploys the `LINKMigrator` contract and can configure the `queueDepositMin` amount.
    - Upgrades the `PriorityPool` contract to add the new `bypassQueue` call.
    - Sets the `queueBypassController`, the only address authorized to call `PriorityPool::bypassQueue`.
2. **Chainlink Community Pool Stakers**:
    - Migrate their position from the Chainlink community vault to stake.link by executing a batch transaction. In a single atomic transaction, they:
        1. Call `LINKMigrator::initiateMigration`.
        2. Exit their position in the community pool.
        3. Call `ERC677::transferAndCall` with LINK tokens to `LINKMigrator`, which deposits the tokens into `StakingPool`, bypassing the `PriorityQueue`.

- This process requires the user to either upgrade their EOA to an ERC-7702-compatible wallet or use a smart contract wallet from the outset in order to perform the necessary atomic operations.

4.3 Security Considerations

Upgradeability Risks and Implementation Changes The `PriorityPool` contract uses the UUPS (Universal Upgradeable Proxy Standard) pattern, which enables implementation upgrades while preserving contract state. This offers flexibility for deploying bug fixes and new features, but also introduces potential risks. Unlike traditional proxy patterns with separate admin contracts, UUPS embeds upgrade logic within the implementation itself. As a result, a compromised implementation could potentially lead to a complete loss of funds. Upgradeability is controlled by the `PriorityPool` contract owner.

5 Audit Scope

The audit was performed on the changes in PR#152 including changes in files:

```
contracts/core/interfaces/IPriorityPool.sol
contracts/core/priorityPool/PriorityPool.sol
contracts/linkStaking/LINKMigrator.sol
```

6 Executive Summary

Over the course of 5 days, the Cyfrin team conducted an audit on the Stake.Link PR152 LINKMigrator smart contracts provided by Stake.Link. In this period, a total of 8 issues were found.

The audit uncovered three low-severity findings:

1. An unused minimum deposit amount configurable by the owner.
2. A missing pool status check in the `bypassQueue` function.
3. A migration flow that could be exploited to bypass the queue without a genuine exit from the community vault.

Several informational findings were also noted, aimed at improving code quality, along with some gas optimization suggestions.

Overall, the code was well-structured and written with security in mind. The test suite was adequate and effectively covered the newly introduced functionality.

Summary

Project Name	Stake.Link PR152 LINKMigrator
Repository	contracts
Commit	0bd5e1eecd86...
Audit Timeline	May 28th - Jun 3rd, 2025
Methods	Manual Review, Stateful Fuzzing

Issues Found

Critical Risk	0
High Risk	0
Medium Risk	0
Low Risk	3
Informational	3
Gas Optimizations	2
Total Issues	8

Summary of Findings

[L-1] Minimum deposit value not enforced in `LINKMigrator`	Resolved
[L-2] Missing `poolStatus` check in `bypassQueue`	Resolved
[L-3] Existing Chainlink stakers can skip queue by bypassing migration requirements	Resolved
[I-1] Unused error	Resolved
[I-2] Lack of events emitted on important state changes	Acknowledged
[I-3] Consider renaming `LINKMigrator::_isUnbonded` for clarity	Resolved
[G-1] Unchanged state variables can be immutable	Resolved
[G-2] Inefficient storage layout in `LINKMigrator.Migration`	Closed

7 Findings

7.1 Low Risk

7.1.1 Minimum deposit value not enforced in `LINKMigrator`

Description: The `LINKMigrator` contract includes a `queueDepositMin` field intended to define the minimum deposit amount. However, this value is currently unused, allowing users to deposit amounts as small as 1 juel (1e-18 LINK).

Impact: Without enforcement, users can submit deposits smaller than the protocol likely intended, potentially increasing overhead or disrupting expected deposit behavior.

Recommended Mitigation: Consider either removing the unused `queueDepositMin` field or enforcing it in the `initiateMigration` function:

```
  function initiateMigration(uint256 _amount) external {
-     if (_amount == 0) revert InvalidAmount();
+     if (_amount < queueDepositMin) revert InvalidAmount();
```

stake.link: Fixed in 0abd4f8

Cyfrin: Verified. `queueDepositMin` is removed. `minStakeAmount` is now fetched from the community pool and verified against `_amount`:

```
(uint256 minStakeAmount, ) = communityPool.getStakerLimits();
if (_amount < minStakeAmount) revert InvalidAmount();
```

7.1.2 Missing `poolStatus` check in `bypassQueue`

Description: The `bypassQueue` function in `PriorityPool.sol` doesn't check the pool's status before depositing tokens directly into the staking pool.

```
function bypassQueue(
    address _account,
    uint256 _amount,
    bytes[] calldata _data
) external onlyQueueBypassController {
    token.safeTransferFrom(msg.sender, address(this), _amount);

    uint256 canDeposit = stakingPool.canDeposit();
    if (canDeposit < _amount) revert InsufficientDepositRoom();

    stakingPool.deposit(_account, _amount, _data);
}
```

The pool status check is part of the protocol's emergency response system. The RebaseController can set the pool status to `CLOSED` during emergency situations, such as when the strategy is leading to a loss of funds. This reason can be seen on `RebaseController` when reopening the pool:

```
@>      * @notice Reopens the priority pool and security pool after they were paused as a result
@>      * of a loss and updates strategy rewards in the staking pool
     * @param _data encoded data to pass to strategies
     */
    function reopenPool(bytes calldata _data) external onlyOwner {
        if (priorityPool.poolStatus() == IPriorityPool.PoolStatus.OPEN) revert PoolOpen();


        priorityPool.setPoolStatus(IPriorityPool.PoolStatus.OPEN);
        if (address(securityPool) != address(0) && securityPool.claimInProgress()) {
            securityPool.resolveClaim();
```

```
        }
        _updateRewards(_data);
    }
```

This missing check in the `bypassQueue` function allows user funds to be deposited when the pool is `CLOSED`, potentially causing deposited tokens to be lost during protocol emergency shutdowns.

Impact: LINK tokens can be deposited via `LINKMigrator` even when the `PriorityPool` is `CLOSED` or `DRAINING`, effectively bypassing the emergency pause mechanism intended for use during security incidents. This could potentially result in users losing funds. However, this risk is considered low in the context of the Chainlink community pool, as there is no inherent mechanism for loss, slashing only occurs in the operator pool. As such, the scenario would only pose a threat if one of the involved contracts were compromised and a user still migrates to it.

Recommended Mitigation: Add a pool status check in the bypassQueue function:

```
function bypassQueue(
    address _account,
    uint256 _amount,
    bytes[] calldata _data
) external onlyQueueBypassController {
+    if (poolStatus != PoolStatus.OPEN) revert DepositsDisabled();
    ...
}
```

stake.link: Fixed in c595886

Cyfrin: Verified. The recommended mitigation was implemented.

7.1.3 Existing Chainlink stakers can skip queue by bypassing migration requirements

Description: The purpose of `LINKMigrator` is that users can migrate their existing position from the Chainlink community pool to stake.link vaults, even when the community pool is at full utilization, since vacating a position frees up space. For users without an existing position, a queueing system (`PriorityQueue`) is used to wait for available slots in the community pool.

However, a user with an existing position can exploit this mechanism by faking a migration. By moving their position to another address (e.g., a small contract they control), they can bypass the queue and open a new position in stake.link if space is available.

Migration begins with a call to `LINKMigrator::initiateMigration`:

```
function initiateMigration(uint256 _amount) external {
    if (_amount == 0) revert InvalidAmount();

    uint256 principal = communityPool.getStakerPrincipal(msg.sender);

    if (principal < _amount) revert InsufficientAmountStaked();
    if (!_isUnbonded(msg.sender)) revert TokensNotUnbonded();

    migrations[msg.sender] = Migration(
        uint128(principal),
        uint128(_amount),
        uint64(block.timestamp)
    );
}
```

Here, the user's principal is recorded. Later, the migration is completed via `transferAndCall`, which triggers `LINKMigrator::onTokenTransfer`:

```
uint256 amountWithdrawn = migration.principalAmount -
    communityPool.getStakerPrincipal(_sender);
```

```
if (amountWithdrawn < _value) revert InsufficientTokensWithdrawn();
```

This compares the recorded and current principal to verify the withdrawal. However, it does not validate that the total staked amount in the community pool has decreased. As a result, a user can withdraw their position, transfer it to a contract they control, and still pass the check, allowing them to deposit directly into stake.link and bypass the queue.

Impact: A user with an existing position in the Chainlink community vault can circumvent the queue system and gain direct access to stake.link staking. This requires being in the claim period, having sufficient LINK to stake again, and available space in the Chainlink community vault. It also resets the bonding period, meaning the user would need to wait another 28 days (the Chainlink bonding period at the time of writing) before interacting with the new position. Nevertheless, this behavior could lead to unfair queue-skipping and undermine the fairness of the protocol.

Proof of Concept: Add the following test to `link-migrator.ts` which demonstrates the queue bypass by simulating a migration and re-staking via a third contract::

```
it('can bypass queue using existing position', async () => {
  const { migrator, communityPool, accounts, token, stakingPool } = await loadFixture(
    deployFixture
  )

  // increase max pool size so we have space for the extra position
  await communityPool.setMaxPoolSize(toEther(3000))

  // deploy our small contract to hold the existing position
  const chainlinkPosition = (await deploy('ChainlinkPosition', [
    communityPool.target,
    token.target,
  ])) as ChainlinkPosition

  // get to claim period
  await communityPool.unbond()
  await time.increase(unbondingPeriod)

  // start batch transaction
  await ethers.provider.send('evm_setAutomine', [false])

  // 1. call initiate migration
  await migrator.initiateMigration(toEther(1000))

  // 2. unstake
  await communityPool.unstake(toEther(1000))

  // 3. transfer the existing position to a contract you control
  await token.transfer(chainlinkPosition.target, toEther(1000))
  await chainlinkPosition.deposit()

  // 4. transferAndCall a new position bypassing the queue
  await token.transferAndCall(
    migrator.target,
    toEther(1000),
    ethers.AbiCoder.defaultAbiCoder().encode(['bytes[]'], [[encodeVaults([])]])
  )
  await ethers.provider.send('evm_mine')
  await ethers.provider.send('evm_setAutomine', [true])

  // user has both a 1000 LINK position in stake.link StakingPool and chainlink community pool
  assert.equal(fromEther(await communityPool.getStakerPrincipal(accounts[0])), 0)
  assert.equal(fromEther(await stakingPool.balanceOf(accounts[0])), 1000)
  assert.equal(fromEther(await communityPool.getStakerPrincipal(chainlinkPosition.target)), 1000)
```

```
  // community pool is full again
  assert.equal(fromEther(await communityPool.getTotalPrincipal()), 3000)
  assert.equal(fromEther(await stakingPool.totalStaked()), 2000)
  assert.deepEqual(await migrator.migrations(accounts[0]), [0n, 0n, 0n])
})
```

Along with `ChainlinkPosition.sol`:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity 0.8.15;

import "./interfaces/IStaking.sol";
import "../core/interfaces/IERC677.sol";

contract ChainlinkPosition {

    IStaking communityPool;
    IERC677 link;

    constructor(address _communityPool, address _link) {
        communityPool = IStaking(_communityPool);
        link = IERC677(_link);
    }

    function deposit() public {
        link.transferAndCall(address(communityPool), link.balanceOf(address(this)), "");
    }
}
```

Recommended Mitigation: In `LINKMigrator::onTokenTransfer`, consider validating that the total principal in the community pool has decreased by at least `_value`, to ensure the migration reflects an actual exit from the community pool.

stake.link: Fixed in de672a7

Cyfrin: Verified. Recommended mitigation was implemented. Community pool total principal is now recorded in `initiateMigration` then compared to the new pool total principal in `onTokenTransfer`.

7.2 Informational

7.2.1 Unused error

Description: In LINKMigrator.sol#L47 the error `InvalidPPState` is unused, consider using or removing it.

stake.link: Fixed in 6827d9d

Cyfrin: Verified.

7.2.2 Lack of events emitted on important state changes

Description: `LINKMigrator::setQueueDepositMin` and `PriorityPool::setQueueBypassController` change internal state without emitting events. Events are important for off-chain tracking and transparency. Consider emitting events from these functions.

stake.link: Acknowledged.

7.2.3 Consider renaming `LINKMigrator::_isUnbonded` for clarity

Description: In the `LINKMigrator` contract, the function `_isUnbonded` checks whether a user is currently within the claim period for Chainlink staking:

```
function _isUnbonded(address _account) private view returns (bool) {
    uint256 unbondingPeriodEndsAt = communityPool.getUnbondingEndsAt(_account);
    if (unbondingPeriodEndsAt == 0 || block.timestamp < unbondingPeriodEndsAt) return false;

    return block.timestamp <= communityPool.getClaimPeriodEndsAt(_account);
}
```

While functionally correct, the name `_isUnbonded` may not clearly convey its purpose, as it specifically checks whether a user is in the claim period. For improved clarity and consistency with Chainlink's naming convention—such as in `StakingPoolBase::_inClaimPeriod`—renaming it could make the intent more immediately clear:

```
function _inClaimPeriod(Staker storage staker) private view returns (bool) {
  if (staker.unbondingPeriodEndsAt == 0 || block.timestamp < staker.unbondingPeriodEndsAt) {
    return false;
  }

  return block.timestamp <= staker.claimPeriodEndsAt;
}
```

Recommended Mitigation: Consider renaming `_isUnbonded` to `_inClaimPeriod` to better reflect its logic and improve code readability.

stake.link: Fixed in 9d710bf

Cyfrin: Verified.

7.3 Gas Optimization

7.3.1 Unchanged state variables can be immutable

Description: None of:

- `LINKMigrator.linkToken`
- `LINKMigrator.communityPool`
- `LINKMigrator.priorityPool`

Are changed outside of the constructor. Consider making them `immutable` to save on gas when accessing them.

stake.link: Fixed in `6f9d9b7`

Cyfrin: Verified.

7.3.2 Inefficient storage layout in `LINKMigrator.Migration`

Description: Here's the `LINKMigrator.Migration` struct:

```
struct Migration {
    // amount of principal staked in Chainlink community pool
    uint128 principalAmount;
    // amount to migrate
    uint128 amount;
    // timestamp when migration was initiated
    uint64 timestamp;
}
```

This struct occupies more than 256 bits and therefore spans two storage slots. However, any LINK amount can be safely stored in a `uint96`, since the total LINK supply is 1 billion (10^9 * 10^18), which is well below the maximum value representable by a `uint96` (~7.9 * 10^28). By changing both amount fields to `uint96`, the struct would comprise `uint96 + uint96 + uint64`, which fits neatly within a single 256-bit storage slot.

Cyfrin: Not applicable after fix in `de672a7`