Stake.Link Token Vesting Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

Immeas

InAllHonesty

August 2, 2025

Contents

1 About Cyfrin **2**

2 Disclaimer **2**

3 Risk Classification **2**

4 Protocol Summary **2**
4.1 Actors and Roles . . . 2
4.2 Key components . . . 3

5 Audit Scope **3**

6 Executive Summary **3**

7 Findings **5**
7.1 Medium Risk . . . 5
7.1.1 Missing access control in `SDLVesting::stakeReleasableTokens` . . . 5
7.2 Informational . . . 6
7.2.1 Zero-Duration vesting edge case . . . 6
7.2.2 Lack of `_lockTime` validation in `constructor` . . . 6
7.2.3 Consider using `Ownable2Step` . . . 6
7.2.4 Consider disabling the owner to renounce ownership . . . 6
7.2.5 `SDLVesting::withdrawRESDLPositions` enhancements . . . 6
7.2.6 `SDLVesting::claimRESDLRewards()` can be used to drain the entire vesting contract balance in edge case . . . 7
7.2.7 NatSpec enhancements . . . 7
7.3 Gas Optimization . . . 8
7.3.1 Storage variable layout can be optimized . . . 8
7.3.2 `SDLVesting::vestedAmount` gas optimizations . . . 8
7.3.3 `SDLVesting::stakeReleasableTokens` gas optimization by caching variables . . . 8
7.3.4 use fixed length array for `reSDLTokenIds` . . . 8

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 Vesting is a new contract that manages the vesting of SDL tokens for Chainlink node operators staking through the Stake.Link ecosystem. Each node operator has their own vesting contract, which the Stake.Link team seeds with SDL tokens. The node operator chooses a lock duration (0–4 years), and the Stake.Link team runs a bot that periodically stakes the vested tokens into the Stake.Link SDLPool for the specified duration.

4.1 Actors and Roles

- **Actors:**
  - **Stake.Link team:**
    - Deploys vesting contracts for node operators
    - Seeds those contracts with SDL tokens to be vested
    - Can terminate vesting by calling `terminateVesting`
  - **Node operators:**
    - Configure the lock duration in their vesting contract via `setLockTime`
    - Withdraw vested SDL tokens without staking via `release`
    - Claim rewards from locked reSDL positions via `claimRESDLRewards`
    - Withdraw their reSDL position NFTs from the vesting contract via `withdrawRESDLPositions`
  - **Staking bot:**
    - Periodically calls `stakeReleasableTokens` to automatically stake vested SDL tokens into the SDLPool
- **Roles:**

  - **Owner:** The Stake.Link wallet that deploys vesting contracts
  - **Beneficiary:** The node operator's wallet

4.2 Key components

- **SDLVesting:** Handles the vesting schedule and staking of SDL tokens for node operators.

5 Audit Scope

Summary of the audit scope and any specific inclusions/exceptions.

6 Executive Summary

Over the course of 3 days, the Cyfrin team conducted an audit on the Stake.Link Token Vesting smart contracts provided by Stake.Link. In this period, a total of 12 issues were found.

The audit uncovered one medium-severity finding: missing access-control in `stakeReleasableTokens` that allows anyone to lock vested SDL tokens into the SDLPool on the node operator's behalf, potentially front running calls to `release`

In addition, we identified several informational findings around a rare vesting edge case, opportunities for extra input validation and best-practice hardening, and a handful of gas-optimization suggestions.

Summary

Project Name	Stake.Link Token Vesting
Repository	contracts
Commit	3462c0d04ff9...
Fix Commit	128c33560d8f...
Audit Timeline	Jul 30th - Aug 1st, 2025
Methods	Manual Review

Issues Found

Critical Risk	0
High Risk	0
Medium Risk	1
Low Risk	0
Informational	7
Gas Optimizations	4
Total Issues	12

Summary of Findings

[M-1] Missing access control in `SDLVesting::stakeReleasableTokens`	Resolved
[I-1] Zero-Duration vesting edge case	Resolved
[I-2] Lack of `_lockTime` validation in `constructor`	Resolved
[I-3] Consider using `Ownable2Step`	Acknowledged
[I-4] Consider disabling the owner to renounce ownership	Acknowledged
[I-5] `SDLVesting::withdrawRESDLPositions` enhancements	Resolved
[I-6] `SDLVesting::claimRESDLRewards()` can be used to drain the entire vesting contract balance in edge case	Resolved
[I-7] NatSpec enhancements	Resolved
[G-1] Storage variable layout can be optimized	Acknowledged
[G-2] `SDLVesting::vestedAmount` gas optimizations	Acknowledged
[G-3] `SDLVesting::stakeReleasableTokens` gas optimization by caching variables	Resolved
[G-4] use fixed length array for `reSDLTokenIds`	Resolved

7 Findings

7.1 Medium Risk

7.1.1 Missing access control in `SDLVesting::stakeReleasableTokens`

Description: The `SDLVesting::stakeReleasableTokens` function is meant to be driven by a trusted staking bot, run by the Stake.Link team, to periodically take any newly-vested SDL and lock it into the SDLPool under the beneficiary's chosen duration. However, it currently has no access control.

Thus anyone can call `SDLVesting::stakeReleasableTokens`. Using this an attacker can front run calls to `SDLVesting::release` by locking the beneficiaries tokens.

Impact: An adversary can repeatedly deny the beneficiary access to vested tokens by front running their `release()` transactions.

Proof of Concept:

```
it('should prevent griefing attacks', async () => {
    const { signers, accounts, start, vesting, sdlPool, sdlToken } = await loadFixture(deployFixture)

    await vesting.connect(signers[1]).setLockTime(4) // 4-year lock
    await time.increase(DAY)

    const releasableAmount = await vesting.releasable()
    console.log("Tokens victim wanted as liquid:", fromEther(releasableAmount), "SDL")

    await vesting.stakeReleasableTokens() //Someone frontruns and forces staking
    const lockIdAfter = await sdlPool.lastLockId()
    console.log("Frontrun with `stakeReleasableTokens`, position created with ID:",
    ↪   lockIdAfter.toString())
    console.log("Position owner:", await sdlPool.ownerOf(lockIdAfter), "(vesting contract)")

    // Get the staking position details
    const locks = await sdlPool.getLocks([lockIdAfter])
    const lock = locks[0]

    const currentTime = await time.latest()
    const lockDuration = Number(lock.duration)
    const unlockInitiationTime = Number(lock.startTime) + lockDuration / 2
    const fullUnlockTime = Number(lock.startTime) + lockDuration

    console.log("Base SDL staked:", fromEther(lock.amount), "SDL")
    console.log("Boost received:", fromEther(lock.boostAmount), "SDL")
    console.log("Total effective staking power:", fromEther(lock.amount + lock.boostAmount), "SDL")
    console.log("Lock duration:", lockDuration / (365 * 86400), "years")

    console.log("Years until unlock initiation allowed:", (unlockInitiationTime - currentTime) / (365
    ↪   * 86400))
    console.log("Years until full withdrawal possible:", (fullUnlockTime - currentTime) / (365 *
    ↪   86400))

    // Victim has almost no liquid tokens
    await vesting.connect(signers[1]).release() // Get the tiny remainder
    const victimLiquidBalance = await sdlToken.balanceOf(accounts[1])
    console.log("Victim's liquid balance:", fromEther(victimLiquidBalance), "SDL (instead of",
    ↪   fromEther(releasableAmount), "SDL)")

    // Victim transfers position to themselves, but it's still locked
    console.log("Transferring position to beneficiary...")
    await vesting.connect(signers[1]).withdrawRESDLPositions([4])
    console.log("Position now owned by:", await sdlPool.ownerOf(lockIdAfter))
    try {
      await sdlPool.connect(signers[1]).initiateUnlock(lockIdAfter)
      console.log("UNEXPECTED: Immediate unlock initiation succeeded!")
    } catch (error) {
      console.log("EXPECTED: Beneficiary cannot initiate unlock yet")
      console.log("Must wait 2 years before unlock can even be INITIATED, and another 2 years for full
      ↪  withdrawal")
    }
  })
```

Recommended Mitigation:

1. Introduce a `stakingBot` address that is the only non-beneficiary permitted to call the staking function:

```
+    /// @notice Address of the trusted bot allowed to call stakeReleasableTokens
+    address public stakingBot;
```

2. Set it in the constructor alongside `_owner` and `_beneficiary`:

```
     constructor(
         address _sdlToken,
         address _sdlPool,
         address _owner,
         address _beneficiary,
+        address _stakingBot,
         uint64 _start,
         uint64 _duration,
         uint64 _lockTime
     ) {
         _transferOwnership(_owner);
+        stakingBot = _stakingBot;
         ...
     }
```

3. Create a combined access-control modifier allowing only the beneficiary *or* the staking bot:

```
+    modifier onlyBeneficiaryOrBot() {
+        if (msg.sender != beneficiary && msg.sender != stakingBot) {
+            revert SenderNotAuthorized();
+        }
+        _;
+    }
```

4. Apply that modifier to `stakeReleasableTokens()`:

```
-    function stakeReleasableTokens() external {
+    function stakeReleasableTokens() external onlyBeneficiaryOrBot {
         uint256 amount = releasable();
         if (amount == 0) revert NoTokensReleasable();
         ...
     }
```

5. In case the bot's key rotates or the Stake.Link team needs to change the automation address, add an owner-only setter:

```
+    /// @notice Update the trusted staking bot address
+    function setStakingBot(address _stakingBot) external onlyOwner {
+        require(_stakingBot != address(0), "Invalid bot address");
+        stakingBot = _stakingBot;
+    }
```

With these changes, only the designated bot (and the beneficiary themselves, if desired) can trigger the periodic staking—eliminating the griefing vector that could otherwise lock tokens indefinitely.

Stake.Link: Fixed in commit `565b043`

Cyfrin: Verified. An address `staker` is now passed to the constructor. And a modifier `onlyBeneficiaryOrStaker` is applied to `stakeReleasableTokens`.

7.2 Informational

7.2.1 Zero-Duration vesting edge case

Description: When `duration == 0`, calling `vestedAmount(start)` falls through the `else if (_timestamp > start + duration)` check (because `start > start` is false) into the linear-vest branch, executing

```
(totalAllocation * (start - start)) / duration
```

This creates a brief one-second revert window at exactly `start`. Since any later timestamp (> `start`) correctly returns full allocation.

Consider changing the comparison to >=:

```
- else if (_timestamp > start + duration) {
+ else if (_timestamp >= start + duration) {
    return totalAllocation;
}
```

so that `start + duration` (even when zero) immediately yields the "fully vested" branch.

Stake.Link: Fixed in commit e458512

Cyfrin: Verified. Comparison is not >=.

7.2.2 Lack of `_lockTime` validation in `constructor`

Description: The constructor assigns `lockTime = _lockTime` without checking `_lockTime <= MAX_LOCK_TIME`. If an out-of-range value is provided, any subsequent call that indexes `reSDLTokenIds[lockTime]` (e.g. in `stakeReleasableTokens` or `withdrawRESDLPositions`) will revert with an array-bounds error.

Consider adding an explicit check in the constructor to improve UX and fail fast:

```
require(_lockTime <= MAX_LOCK_TIME, "Invalid lock time");
```

Stake.Link: Fixed in commit e458512

Cyfrin: Verified. `_lockTime` now required to not be larger than `MAX_LOCK_TIME`.

7.2.3 Consider using `Ownable2Step`

Description: The contract uses OpenZeppelin's single-step `Ownable`, where ownership transfers immediately upon `transferOwnership`, risking accidental or unwanted transfers. Consider switching to OZ's `Ownable2Step`, which requires the new owner to explicitly call `acceptOwnership`. This two-step pattern prevents mis-sent or unacknowledged ownership transfers and aligns with best-practice "Unchecked 2-Step Ownership Transfer" safeguards.

Stake.Link: Acknowledged.

7.2.4 Consider disabling the owner to renounce ownership

Description: `Ownable`'s default `renounceOwnership()` allows the owner to relinquish control entirely by setting owner to `address(0)`, which can be called unintentionally. To avoid permanent loss of privileged functions, consider overriding `renounceOwnership` to disable or restrict it. For example:

```
function renounceOwnership() public view override onlyOwner {
    revert("Renouncing ownership is disabled");
}
```

This ensures ownership can only change via deliberate `transferOwnership` (or `Ownable2Step`), preventing accidental or irreversible renouncement.

Stake.Link: Acknowledged.

7.2.5 `SDLVesting::withdrawRESDLPositions` enhancements

Description: The `SDLVesting::withdrawRESDLPositions()` function violates the Checks-Effects-Interactions (CEI) pattern by performing external calls before updating state. The function also lacks input validation for lock times and may attempt to transfer non-existent token IDs, causing transaction reverts and poor user experience.

While not exploitable due to SDLPool's ownership checks, the function updates state after external calls, creating a potential reentrancy vector as well as bad user experience in case of incorrect input.

Consider adding validation for `_lockTimes` and move the state changes before the external call:

```
    function withdrawRESDLPositions(uint256[] calldata _lockTimes) external onlyBeneficiary {
        for (uint256 i = 0; i < _lockTimes.length; ++i) {

-           sdlPool.safeTransferFrom(address(this), beneficiary, reSDLTokenIds[_lockTimes[i]]);
-           delete reSDLTokenIds[_lockTimes[i]];

+           if (_lockTimes[i] > MAX_LOCK_TIME) revert InvalidLockTime();
+           uint256 tokenId = reSDLTokenIds[_lockTimes[i]]; // Cache to facilitate the deletion before
↪   transfer
+           if (tokenId == 0) continue; // Skip if no reSDL position exists so we don't break execution
↪   but also don't attempt to transfer 0.
+           delete reSDLTokenIds[_lockTimes[i]];
+           sdlPool.safeTransferFrom(address(this), beneficiary, tokenId);
        }
    }
```

Stake.Link: Fixed in commit e458512

Cyfrin: Verified. `_lockTime[i]` now verified to not be larger than `MAX_LOCK_TIME` and delete is done before call to `safeTransfer`.

7.2.6 `SDLVesting::claimRESDLRewards()` can be used to drain the entire vesting contract balance in edge case

Description: The `SDLVesting::claimRESDLRewards` function transfers the entire token balance to the beneficiary for each specified reward token. If the admin mistakenly adds SDL token as a reward token in the `RewardsPool-Controller`, the beneficiary could call `SDLVesting::claimRESDLRewards([sdlToken])` to instantly drain all vested and unvested SDL tokens from the contract, completely bypassing the vesting schedule.

```
    function claimRESDLRewards(address[] calldata _tokens) external onlyBeneficiary {
        sdlPool.withdrawRewards(_tokens);

        for (uint256 i = 0; i < _tokens.length; ++i) {
            IERC20 token = IERC20(_tokens[i]);
            uint256 balance = token.balanceOf(address(this));

            if (balance != 0) {
                token.safeTransfer(beneficiary, balance);
            }
        }
    }
```

This happens because the entire `token.balanceOf(address(this));` is transferred to the beneficiary.

Consider adding a check to prevent SDL token from being claimed as a reward:

```
function claimRESDLRewards(address[] calldata _tokens) external onlyBeneficiary {
    sdlPool.withdrawRewards(_tokens);

    for (uint256 i = 0; i < _tokens.length; ++i) {
+       if (_tokens[i] == address(sdlToken)) continue; // Skip SDL token
```

```
        IERC20 token = IERC20(_tokens[i]);
        uint256 balance = token.balanceOf(address(this));
        if (balance != 0) {
            token.safeTransfer(beneficiary, balance);
        }
    }
}
```

Stake.Link: Fixed in commit e458512

Cyfrin: Verified. `_tokens[i]` now checked to not be equal to `sdlToken`.

7.2.7 NatSpec enhancements

Description: * SDLVesting#L11: `beneficary` should be `beneficiary`

- SDLVesting::onlyBeneficiary#L90: extra space in `not  beneficiary`
- SDLVesting::setLockTime#L187: `_lockTime lock time in seconds` should be `_lockTime lock time in years`
- SDLVesting::vestedAmount: Doesn't document the return: `@return amount of tokens vested at the given timestamp. Returns full allocation if terminated.`

Stake.Link: Fixed in commits e458512 and 565b043

Cyfrin: Verified.

7.3 Gas Optimization

7.3.1 Storage variable layout can be optimized

Description: The current storage layout uses 4 slots.

```
    // maximum lock time in years
    uint256 public constant MAX_LOCK_TIME = 4;

    // address of SDL token
    IERC677 public immutable sdlToken;
    // address of SDL pool
    ISDLPool public immutable sdlPool;

    // whether vesting has been terminated
    bool public vestingTerminated;

    // amount of tokens claimed by the beneficiary
    uint256 public released;
    // address to receive vested SDL
    address public immutable beneficiary;
    // start time of vesting in seconds
    uint64 public immutable start;
    // duration of vesting in seconds
    uint64 public immutable duration;

    // lock time in years to use for staking vested SDL
    uint64 public lockTime;
    // list of reSDL token ids for each lock time
    uint256[] private reSDLTokenIds;
```

```
--------------------+-----------+------+--------+-------+---------------------------------------------
| Name               | Type      | Slot | Offset | Bytes | Contract                                    |
+========================================================================================================+
| _owner             | address   | 0    | 0      | 20    | contracts/vesting/SDLVesting.sol:SDLVesting |
|--------------------+-----------+------+--------+-------+---------------------------------------------|
| vestingTerminated  | bool      | 0    | 20     | 1     | contracts/vesting/SDLVesting.sol:SDLVesting |
|--------------------+-----------+------+--------+-------+---------------------------------------------|
| released           | uint256   | 1    | 0      | 32    | contracts/vesting/SDLVesting.sol:SDLVesting |
|--------------------+-----------+------+--------+-------+---------------------------------------------|
| lockTime           | uint64    | 2    | 0      | 8     | contracts/vesting/SDLVesting.sol:SDLVesting |
|--------------------+-----------+------+--------+-------+---------------------------------------------|
| reSDLTokenIds      | uint256[] | 3    | 0      | 32    | contracts/vesting/SDLVesting.sol:SDLVesting |
--------------------+-----------+------+--------+-------+---------------------------------------------
```

By moving the `lockTime` after `vestingTerminated` we can reduce it to 3 slots:

```
--------------------+-----------+------+--------+-------+---------------------------------------------
| Name               | Type      | Slot | Offset | Bytes | Contract                                    |
+========================================================================================================+
| _owner             | address   | 0    | 0      | 20    | contracts/vesting/SDLVesting.sol:SDLVesting |
|--------------------+-----------+------+--------+-------+---------------------------------------------|
| vestingTerminated  | bool      | 0    | 20     | 1     | contracts/vesting/SDLVesting.sol:SDLVesting |
|--------------------+-----------+------+--------+-------+---------------------------------------------|
| lockTime           | uint64    | 0    | 21     | 8     | contracts/vesting/SDLVesting.sol:SDLVesting |
|--------------------+-----------+------+--------+-------+---------------------------------------------|
| released           | uint256   | 1    | 0      | 32    | contracts/vesting/SDLVesting.sol:SDLVesting |
|--------------------+-----------+------+--------+-------+---------------------------------------------|
| reSDLTokenIds      | uint256[] | 2    | 0      | 32    | contracts/vesting/SDLVesting.sol:SDLVesting |
--------------------+-----------+------+--------+-------+---------------------------------------------
```

```
    // maximum lock time in years
    uint256 public constant MAX_LOCK_TIME = 4;
    // address of SDL token
    IERC677 public immutable sdlToken;
    // address of SDL pool
    ISDLPool public immutable sdlPool;

    // whether vesting has been terminated
    bool public vestingTerminated;

+   // lock time in years to use for staking vested SDL
+   uint64 public lockTime;

    // amount of tokens claimed by the beneficiary
    uint256 public released;
    // address to receive vested SDL
    address public immutable beneficiary;
    // start time of vesting in seconds
    uint64 public immutable start;
    // duration of vesting in seconds
    uint64 public immutable duration;

-   // lock time in years to use for staking vested SDL
-   uint64 public lockTime;
    // list of reSDL token ids for each lock time
    uint256[] private reSDLTokenIds;
```

Stake.Link: Acknowledged.

7.3.2 `SDLVesting::vestedAmount` gas optimizations

Description: The `SDLVesting::vestedAmount` function computes the `totalAllocation` before the `if (_timestamp < start) return 0;` check. In the event that check returns true and the function returns 0 then the gas spent on that computation is wasted. Moreover, given that the condition of `_timestamp < start` is checked the final return `return (totalAllocation * (_timestamp - start)) / duration;` can be done in an unchecked block to save gas:

```
    function vestedAmount(uint64 _timestamp) public view returns (uint256) {
+       if (_timestamp < start) {
+            return 0;

        uint256 totalAllocation = sdlToken.balanceOf(address(this)) + released;

-      if (_timestamp < start) {
-           return 0;
-       } else if (_timestamp > start + duration) {
-           return totalAllocation;
-       } else if (vestingTerminated) {
-           return totalAllocation;
-       } else {
-           return (totalAllocation * (_timestamp - start)) / duration;
+       if (_timestamp > start + duration) {
+           return totalAllocation;
+      } else if (vestingTerminated) {
+           return totalAllocation;
+      } else {
+           unchecked {
+               return (totalAllocation * (_timestamp - start)) / duration;
+           }
        }
    }
```

Stake.Link: Acknowledged.

7.3.3 `SDLVesting::stakeReleasableTokens` gas optimization by caching variables

Description:

```
    function stakeReleasableTokens() external {
        uint256 amount = releasable();
        if (amount == 0) revert NoTokensReleasable();

        released += amount;
        sdlToken.transferAndCall(
            address(sdlPool),
            amount,
            abi.encode(reSDLTokenIds[lockTime], lockTime * (365 days))
        );

        if (reSDLTokenIds[lockTime] == 0) {
            reSDLTokenIds[lockTime] = sdlPool.lastLockId();
        }

        emit Staked(amount);
    }
```

Currently `lockTime` and `reSDLTokenIds[lockTime]` are read from storage for multiple times. Both variables should be cached to save gas:

```
    function stakeReleasableTokens() external {
        uint256 amount = releasable();
        if (amount == 0) revert NoTokensReleasable();

        released += amount;
-       sdlToken.transferAndCall(
-           address(sdlPool),
-           amount,
-           abi.encode(reSDLTokenIds[lockTime], lockTime * (365 days))
-       );
-
-       if (reSDLTokenIds[lockTime] == 0) {
-           reSDLTokenIds[lockTime] = sdlPool.lastLockId();
-       }

+       uint64 cachedLockTime = lockTime;
+       uint256 cachedTokenId = reSDLTokenIds[cachedLockTime];
+
+       sdlToken.transferAndCall(
+          address(sdlPool),
+          amount,
+         abi.encode(cachedTokenId, cachedLockTime * (365 days))
+      );
+
+      if (cachedTokenId == 0) {
+          reSDLTokenIds[cachedLockTime] = sdlPool.lastLockId();
+      }

       emit Staked(amount);
    }
```

Stake.Link: Fixed in commit 128c335

Cyfrin: Verified. `tokenId` now cached.

7.3.4 use fixed length array for `reSDLTokenIds`

Description: The contract currently declares

```
uint256[] private reSDLTokenIds;
```

and in the constructor uses a `for`-loop with `.push(0)` to initialize it to length `MAX_LOCK_TIME + 1`. This incurs:

- A dynamic-array length slot in storage
- A pointer slot for the array data
- Multiple storage writes (one per `.push`)

Since the array's length is always exactly `MAX_LOCK_TIME + 1` (5), a static array:

```
uint256[MAX_LOCK_TIME + 1] private reSDLTokenIds;
```

removes the dynamic-array overhead and eliminates the initialization loop.

Consider replacing the dynamic array with a fixed-length array and remove the constructor loop:

```diff
-   // list of reSDL token ids for each lock time
-   uint256[] private reSDLTokenIds;

+   // list of reSDL token ids for each lock time (0-4 years)
+   uint256[MAX_LOCK_TIME + 1] private reSDLTokenIds;

    constructor(...) {
        ...
-       for (uint256 i = 0; i <= MAX_LOCK_TIME; ++i) {
-           reSDLTokenIds.push(0);
-       }
     }
```

This change collapses two storage slots (length + data pointer) into one and removes the costly initialization loop, reducing both deployment and per-read gas costs.

Stake.Link: Fixed in commit 128c335

Cyfrin: Verified. `reSDLTokenIds` now static.