Tally Fixed Liquid Staking Review

December 13, 2024

Prepared for DappHero Corp

Conducted by:

Kurt Willis (phaze)

Richie Humphrey (devtooligan)

About the Tally Fixed Liquid Staking Review

The Tally Protocol's goal is to fully actualize the value of the systems that token holders own and participate in.

UniLst is a convenient liquid token wrapper on top of UniStaker. UNI holders can stake their UNI for stUNI. UniLst automates claiming rewards and delegating governance power. It's like what stETH does for ETH staking.

UniLst is a rebasing token. This review focused on the fixed version of UniLst. The fixed version is a privileged wrapper that allows for fixed unit accounting (similar to wstETH) but it also has permissioned hooks inside the UniLst contract that allow holders of the fixed token to delegate their underlying voting power.

About Offbeat Security

Offbeat Security is a boutique security company providing unique security solutions for complex and novel crypto projects. Our mission is to elevate the blockchain security landscape through invention and collaboration.

Summary & Scope

The src/ folder of the `stUNI` repo was reviewed at commit 7aa0b92.

The following **3 contracts** were in scope:

- src/contracts/FixedLstAddressAlias.sol
- src/contracts/FixedUniLst.sol
- src/contracts/UniLst (relevant changes)

Summary of Findings

Identifier	Title	Severity	Fixed
L-01	Event emission inconsistencies	Low	Fixed in PR's #175, #176
L-02	Share price inflation when there are no deposits leads to loss of funds	Low	
L-03	Share value casting in UniLst may truncate values	Low	#174
L-04	Division by zero in share calculation when total supply is zero	Low	
I-01	Simplify rescue	Info	

Code Quality Recommendations

Please find code quality recommendations in Appendix A.

Detailed Findings

Low Findings

[L-01] Event emission inconsistencies

As part of our review, the project team asked us to do a deep dive on the event logging system and make recommendations. During our review, we noted inconsistencies in how

events are emitted between workflows. A detailed analysis can be found in Appendix B.

There were two recurring themes that came up during the analysis. One was the use of the aliased user address in certain events. This causes a mismatch when reconciling those events with others that referenced the Fixed UniLst contract address for related actions. Furthermore, the use of the aliased address may create confusion.

In addition we found some inconsistencies with the use of the `Staked` and `Unstaked` event on the Fixed UniLst contract. Part of the inconsistency may be attributable to the fact that both the UniLst and the FixedUniLst contracts have these same-named events but they are used for similar but distinctly different actions.

Recommendations

Detailed recommendations can be found in Appendix B.

[L-02] Share price inflation when there are no deposits leads to loss of funds

Note: This finding was noted during the review but is not directly related to the FixedUniLst code in scope.

The UniLst contract's `claimAndDistributeReward()` function allows manipulation of the share price through reward distribution mechanics. In extreme cases, this can create a state where the total supply is non-zero while total shares are zero, causing new deposits to be permanently lost. The vulnerability stems from allowing reward distribution without requiring an initial deposit or minimum share balance.

The vulnerability lies in the share price calculation mechanics of the UniLst contract. The contract tracks two key values:

- `totalSupply`: The total amount of staked tokens
- `totalShares`: The total number of shares issued

The share price is calculated as `totalSupply / totalShares`. By calling `claimAndDistributeReward()` without any deposits, an attacker can manipulate these values to create extreme share prices:

1. `claimAndDistributeReward()` increases `totalSupply` by adding the reward amount
2. When no shares exist, this creates an imbalanced state where `totalSupply > 0` but `totalShares = 0`
3. New deposits use `_calcSharesForStake()` to calculate shares, which divides by `totalSupply`

4. In the manipulated state, this calculation results in extremely small share amounts or complete loss of funds

The issue is compounded by the ability to iterate through stake/unstake cycles to further manipulate the share price.

Impact Explanation

The impact is medium as it can lead to permanent loss of user funds. When the share price is manipulated, new deposits can be either significantly devalued or completely lost in the system with no way to recover them.

Likelihood Explanation

The likelihood is low. The attack requires specific timing and conditions as well as the attacker sacrificing the reward amount to complete the attack. The existing `SHARE_SCALE_FACTOR` of 1e10 also provides some protection against stealing deposits, but doesn't prevent the possibility of burning them.

Proof of Concept

```
function test_claimAndDistributeReward_DOS() public {
    uint256 _payoutAmount = 2e18;
    uint16 _feeBips = 0.001e4;

    address bob = makeAddr("bob");
    address alice = makeAddr("alice");
    address _feeCollector = makeAddr("feeCollector");

    vm.prank(lstOwner);
    withdrawGate.setDelay(0);

    _setRewardParameters(uint80(_payoutAmount), _feeBips, _feeCollector);

    uint256 aliceBalance = 1e18;

    deal(address(stakeToken), bob, 100e18);
    deal(address(stakeToken), alice, aliceBalance);

    uint256 snap = vm.snapshot();

    // Alice stakes and unstakes
    vm.startPrank(alice);
    stakeToken.approve(address(lst), type(uint256).max);
    lst.stake(aliceBalance);
    lst.unstake(lst.balanceOf(alice));
    assertEq(stakeToken.balanceOf(alice), aliceBalance);
    vm.stopPrank();

    // Go back to snapshot
    vm.revertTo(snap);
```

```
    // Bob pays the stake token amount without any reward
    vm.startPrank(bob);
    stakeToken.approve(address(lst), type(uint256).max);
    lst.claimAndDistributeReward(bob, 0);
    vm.stopPrank();

    // Now Alice stakes. Any stake is lost.
    vm.startPrank(alice);
    stakeToken.approve(address(lst), type(uint256).max);
    lst.stake(aliceBalance);
    lst.unstake(lst.balanceOf(alice));
    assertEq(stakeToken.balanceOf(alice), 0);
}
```

Recommendation

Consider requiring an initial deposit during contract initialization that gets "burned".

```
constructor() {
    // ... existing logic ...
    // Burn initial stake to prevent share price manipulation
    uint256 initialStake = 0.001e18; // Choose appropriate amount
    STAKE_TOKEN.transferFrom(msg.sender, address(this), initialStake);
    _stake(address(this), initialStake);
}
```

[L-03] Share value casting in UniLst may truncate values

Note: This finding was noted during the review but is not directly related to the FixedUniLst code in scope.

The UniLst contract performs unsafe casting operations when handling share values. While comments suggest the casts are safe due to balance validation, this is only true for token amounts (UNI balance) but not for share values which can have different ratios to the underlying assets.

In the staking and unstaking functions:

1. In `_stake()`:

```
// Unsafe casts that could truncate values
_totals.shares = _totals.shares + uint160(_newShares);
_holderState.shares = _holderState.shares + uint128(_newShares);
```

2. In `_unstake()`:

```
// Unsafe casts that could truncate values
_holderState.shares -= uint128(_sharesDestroyed);
```

```
_totals.shares = _totals.shares - uint160(_sharesDestroyed);
```

These casts could silently truncate values if share amounts exceed the bit width of their target types (uint128/uint160). This is particularly concerning since shares do not have a 1:1 relationship with assets and could grow significantly larger through various share price manipulations.

The comment "sharesForStake would fail if overflowed" is correct, however it does not provide safety guarantees for values being truncated. Similarly, the comment "cast is safe because we've subtracted the shares from user" is invalid since the truncation happens before the subtraction.

Recommendation

Consider using OpenZeppelin's SafeCast library to ensure share value casts are performed safely:

```
+using SafeCast for uint256;

// In _stake()
-    _holderState.shares = _holderState.shares + uint128(_newShares);
+    _holderState.shares = _holderState.shares + _newShares.toUint128();

// In _unstake()
-    _holderState.shares -= uint128(_sharesDestroyed);
+    _holderState.shares -= _sharesDestroyed.toUint128();
```

This will revert the transaction if share values exceed their intended bit widths rather than silently truncating them.

[L-04] Division by zero in share calculation when total supply is zero

Note: This finding noted during the review but is not directly related to the FixedUniLst code in scope.

Description

The `calcSharesForStakeUp()` function uses the `mulmod` operation to perform rounding up when calculating shares. However, when `totals.supply` is zero, the `mulmod` operation will revert due to division by zero:

```
if (mulmod(_amount, _totals.shares, _totals.supply) > 0) {
  _result += 1;
}
```

This issue occurs when the vault has no total supply (e.g., during initial deployment or after all funds have been withdrawn) and a user attempts to stake tokens. While the main share calculation would handle zero supply correctly, the rounding logic causes the entire function to revert unexpectedly.

Recommendation

Consider adding a zero supply check at the beginning of the function:

```
function calcSharesForStakeUp(uint256 _amount, Totals memory _totals) internal pu
    uint256 _result = calcSharesForStake(_amount, _totals);
-   if (mulmod(_amount, _totals.shares, _totals.supply) > 0) {
+   if (_totals.supply != 0 && mulmod(_amount, _totals.shares, _totals.supply) >
        _result += 1;
    }
    return _result;
}
```

Note: The recommendation in L-01 would also mitigate this risk since the supply would become non-zero after minting new shares in the constructor.

Informational Findings

[I-01] Simplify rescue

The `rescue` function is intended to be used in case LST tokens are transferred to an alias address by accident. Unlike other similar rescue functions, this function does not transfer the LST tokens back to the owner directly. It deposits the tokens in the `FixedUniLst` contract.

The rescue function also potentially creates inconsistencies with event emissions as discussed in Appendix B.

Recommendation

Consider simpliflying the `rescue` function to transfer the LST tokens back from the aliased address to the user address without depositing them in the fixed contract. At that point, if the user wants to, they can call `convertToFixed` on the `FixedUniLst` contract themselves. This reduces overall complexity and eliminates potential issues related to events discussed in Appendix B.

```
// FixedUniLst.sol
function _rescue(address _account) internal virtual {
  // Shares not accounted for inside this Fixed LST accounting system are the on
  uint256 _sharesToRescue = LST.sharesOf(_account.fixedAlias()) - shareBalances[
  uint256 _stakeTokens = LST.rescue(_account, _sharesToRescue);
```

```
  emit Rescued(_account, _stakeTokens);
  }
```

```
// UniLst.sol
function rescue(address _account, uint256 _shares) external returns (uint256 _am
  _revertIfNotFixedLst();
  (, _amount) = _transfer(_account.fixedAlias(), _account, stakeForShares(_share
}
```

Appendix A. Code Quality and Gas Optimization Recommendations

- For consistency and convenience, consider adding the following functions to the `FixedUniLst` contract:
  - **previewStake() / previewUnstake()** - These functions are useful for integrators.
  - **stakeWithAttribution()** - Adding this function would be consistent with `UniLst` and may be useful to integrators.

- Incorrect comments
  - `/// @notice Internal helper method which reverts with FixedUniLst__SignatureExpired if the signature is invalid` This incorrect comment appears twice. The correct error is `FixedUniLst__InvalidSignature`
  - In `FixedUniLst`, both the `stake` and `_stake` functions, the comment incorrectly states the caller must approve "the rebasing LST contract" but the caller actually must approve the `FixedUniLst` contract itself.

- The `permit()` function in both `UniLst` and `FixedUniLst` wraps the calculation of `_structHash` in an `unchecked` block with a comment indicating it's for nonce increment safety. However, this block is unnecessary since it contains no arithmetic operations - only a hash calculation. The nonce increment occurs in the `_useNonce()` function call which handles its own arithmetic safety. The `unchecked` block provides no gas savings in this context.

- In `UniLst`, the following functions perform a redundant storage read to calculate the difference in balance which is already performed and is the return value for the internal function called.
  - `stakeAndConvertToFixed` - can use the return values of `_stake`
  - `transferFixed` and `convertToFixed` - can use the return values of `_transfer`

Appendix B. Event Logging Review

As part of our review, the project team asked us to do a deep dive on the event logging system and make recommendations.

Stake and Unstake

The following events are emitted when a user calls `stake` and `unstake` directly on the `UniLst` contract (excluding events emitted by UniStaker and Stake Token contracts):

```
// STAKE REBASING

// UniLst.stake(stakeTokens) emits the following events:
Transfer(address(0), USER ADDRESS, stakeTokens)
Staked(USER ADDRESS, stakeTokens)

// UNSTAKE REBASING

// UniLst.unstake(liquidStakedTokens) emits the following events:
Transfer(USER ADDRESS, address(0), stakeTokens)
Unstaked(USER ADDRESS, liquidStakedTokens)
```

Compared with the events emitted by both the UniLst and FixedUniLst contracts when calling `stake` / `unstake` on `FixedUniLst`:

```
// STAKE FIXED

// FixedUniLst.stake(stakeTokens) emits the following events:
Transfer(address(0), USER ADDRESS, fixedTokens);
Staked(USER ADDRESS, stakeTokens);

// UniLst.stakeAndConvertToFixed emits the following events:
Transfer(address(0), ALIASED USER ADDRESS, stakeTokens)
Staked(ADDRESS OF FIXED LST, stakeTokens)

// UNSTAKE FIXED

// FixedUniLst.unstake(fixedTokens) emits the following events:
Transfer(USER ADDRESS, address(0), fixedTokens);
Unstaked(USER ADDRESS, stakeTokens);

// UniLst.convertToRebasingAndUnstake emits the following events:
Transfer(ALIASED USER ADDRESS, address(0), stakeTokens)
Unstaked(ADDRESS OF FIXED LST, stakeTokens)
```

The main difference between the events emitted by `UniLst` and `FixedUniLst` for `stake` and `unstake` are that the `Transfer` and `Staked` events emitted by the `UniLst` contract

use the ALIASED USER ADDRESS and ADDRESS OF FIXED LST addresses respectively instead of the actual address of the caller.

As stated in the code comments:

```
// Externally, we model this as the Fixed LST contract staking on behalf
// of the account in question, so we emit an event that shows the
// Fixed LST contract as the staker.
```

So the `Staked` / `Unstaked` events correctly reference the ADDRESS OF FIXED LST.

However, the `Transfer` events use the ALIASED USER ADDRESS. This discrepancy may have unexpected effects on indexers. For example, tracking the sum of the amounts staked per recipient adjusted by transfers will not equal the sum of the amounts transferred starting from the zero address per recipient. Additionally, the transfer event exposes the ALIASED USER ADDRESS which may create confusion.

Another important point is related to the event names. Both `FixedUniLst` and `UniLSt` have `stake` and `unstake` events, but technically they are used differently. Having the same name may lead to confusion.

Recommendations

- The `Transfer` event emitted by the `UniLst` contract should use the ADDRESS OF FIXED LST instead of the ALIASED USER ADDRESS in both `stakeAndConvertToFixed` and `convertToRebasingAndUnstake`.

- The `FixedUniLst` should not emit the `Staked` / `Unstaked` events. Instead let thr UniLst contrwct use those events and the `FixedUniLst` comtract should emit the `Fixed` / `Unfixed` events.

Update Delegatee

The following event is emitted when a user calls `updateDeposit` directly on the `UniLst` contract:

```
// UPDATE DEPOSIT ID REBASING

// UniLst.updateDeposit(newDepositId) emits the following event:
DepositUpdated(USER ADDRESS, oldDepositId, newDepositId);
```

Compared with calling `updateDeposit` on the new `FixedLst` contract:

```
// UPDATE DEPOSIT ID FIXED

// FixedUniLst.updateDeposit(newDepositId) emits the following event:
DepositUpdated(USER ADDRESS, newDepositId);

// UniLst - No events emitted by UniLst.updateFixedDeposit
```

The main difference between the events emitted by `UniLst` and `FixedUniLst` is that no event is emitted by `UniLst`. This is appropriate since, in the `UniLst` contract, the deposit id is updated on the aliased address which may not be desirable to expose. However, this creates an inconsistency for indexers whereby the sum of the delegated amounts will not equal the sum of the amounts staked and the non-delegated amount will appear overstated.

Additionally, the event on the `FixedUniLst` contract does not log the `oldDepositId` as does its `UniLst` counterpart.

Recommendations

- Add the `oldDepositId` argument to the `FixedUniLst.DepositUpdated` event. This should be an easy fix since the `UniLst.updateFixedDeposit` already returns the `oldDepositId`.

- Add documentation instructing indexers to combine the `DepositUpdated` events emitted by both `UniLst` and `FixedUniLst` for an accurate picture.

Convert to Fixed and Convert to Rebasing

The following events are emitted when a user calls `convertToFixed` and `convertToRebasing` on the new `FixedUniLst` contract. (excluding events emitted by UniStaker and Stake Token contracts):

```
// CONVERT TO FIXED

// FixedUniLst.convertToFixed(liquidStakeTokens) emits the following events:
Fixed(USER ADDRESS, liquidStakeTokens)
Transfer(address(0), USER ADDRESS, _fixedTokens)

// UniLst.convertToFixed(USER ADDRESS, liquidStakeTokens) emits the following eve
Transfer(USER ADDRESS, ADDRESS OF FIXED LST, liquidStakeTokens)

// CONVERT TO REBASING


// FixedUniLst.convertToRebasing(fixedTokens) emits the following events:
Fixed(USER ADDRESS, liquidStakeTokens)
Transfer(address(0), USER ADDRESS, _fixedTokens)
```

```
// UniLst.convertToRebasing(USER ADDRESS, shares) emits the following event:
Transfer(ADDRESS OF FIXED LST, USER ADDRESS, liquidStakeTokens)
```

The `UniLst` correctly emits a `Transfer` event showing a transfer to and from USER ADDRESS to ADDRESS OF FIXED LST. There are no `Staked` or `Unstaked` events needed because, from the `UniLst` perspective the tokens remained staked.

However we can see on the `FixedUniLst` side, it does not emit an `Unstaked` or `Staked` event.

Let's consider how these conversions would be accomplished if the helper functions were not available. For example, `convertToRebasing` could be accomplished by calling `unstake` on `FixedUniLst` and `stake` on `UniLst`:

1. Call `FixedUniLst.unstake(stakeTokens)`
    - `UniLst.convertToRebasingAndUnstake` emits the following events:
        - `Transfer(ALIASED USER ADDRESS, address(0), stakeTokens)`
        - `Unstaked(ADDRESS OF FIXED LST, stakeTokens)`
    - `FixedUniLst.unstake` emits the following events:
        - `Transfer(USER ADDRESS, address(0), fixedTokens)`
        - `Unstaked(USER ADDRESS, stakeTokens)`
2. Call `UniLst.stake(stakeTokens)`
    - `UniLst.stake` emits the following events:
        - `Transfer(address(0), USER ADDRESS, stakeTokens)`
        - `Staked(USER ADDRESS, stakeTokens)`

As shown above, when we call `unstake` on `FixedUniLst` first, followed by `stake` on `UniLst` we end up with two `Transfer` events:

- `Transfer(ALIASED USER ADDRESS, address(0), stakeTokens)`
- `Transfer(address(0), USER ADDRESS, stakeTokens)`

These are equivalent to:

- `Transfer(ALIASED USER ADDRESS, USER ADDRESS, stakeTokens)`

But instead, when we call `convertToRebasing` on `FixedUniLst`, the `UniLst` contract emits:

- `Transfer(ADDRESS OF FIXED LST, USER ADDRESS, stakeTokens)`

One way uses the ALIASED USER ADDRESS and the other uses the ADDRESS OF FIXED LST.

Continuing to look at the differences, it is correct that offsetting `Staked`/`Unstaked` are not emitted by the `UniLst` contract when calling `FixedUniLst.convertToRebasing`. This is effectively a transfer and `UniLst` does not emit `Staked`/`Unstaked` events for transfers.

However, it is problematic that the `FixedUniLst` does not emit an `Unfixed` event when `convertToRebasing` is called.

Recommendations

- Assuming the mitigation reccomended above is followed and the ADDRESS OF FIXED LST is used in the `UniLst.Transfer` event for `stake` / `unstake` then no change is needed in the `UniLst` contract. However, if this change is not implemented, the `UniLst._convertToRebasing` and `UniLst.convertToFixed` functions should be updated so that the `Transfer` event emitted uses the ALIASED USER ADDRESS instead of the ADDRESS OF FIXED LST.

- The `FixedUniLst.convertToFixed` and `convertToRebasing` functions should emit `Fixed` and `Unfixed` events respectively. Note: These event names may be changed per the recommendation above

Rescue

*Note: I-01 recommends changing the behavior of the `rescue` function. The following is based on the current code without adopting the recommendation. If the recommendation in **I-01** is followed then this section can be disregarded*

The following event is emitted when `rescue` is called on the `FixedUniLst` contract:

```
// RESCUE

// UniLst.rescue(USER ADDRESS) emits the following event:
Transfer(address(0), USER ADDRESS, _fixedTokens);
Rescued(USER ADDRESS, _stakeTokens);
```

This would come after LST tokens were transferred to an ALIASED USER ADDRESS. A transfer on the `UniLst` contract would have previously emitted the following event:

```
Transfer(ADDRESS SENDER, ALIASED USER ADDRESS, amount)
```

This rescue process presumably happens when a user sends their aliased address some LST tokens instead of using the `convertToFixed` function on the `FixedUniLst` contract. Had they properly called `convertToFixed` the following events would have been emitted:

```
// UniLst.convertToFixed emits the following events:
Transfer(address(0), ALIASED USER ADDRESS, stakeTokens)
Staked(ADDRESS OF FIXED LST, stakeTokens)
```

One difference between calling `convertToFixed` versus transferring and calling `rescue` is that the `Rescued` event is emitted instead of `Fixed`.

There is another potential difference depending if previously suggested recommedndations are implemented which is discussed below.

Recommendations

- The `rescue` function should cause the `FixedUniLst` contract to emit `Fixed`.

- In the Stake / Unstake section above, it was recommended that the `Transfer` event emitted by `UniLst` should reference the ADDRESS OF FIXED LST. If this recommendation is not followed, then no additional change is needed.

- However, if that change is not enacted, the `Transfer` events emitted by `UniLst` become out of sync. To address this, the `rescue` function can call a new function on `UniLst` which will emit a corrective event:

```
// UniLst.sol

// New function to correct event logs after rescue
function rescue(account, amount) external {
    emit Transfer(account.fixedAlias(), ADDRESS OF FIXED LST, amount)

}
```