Tally Staking Review

September 12, 2024

Prepared for DappHero Corp

Conducted by:

Vara Prasad Bandaru (s3v3ru5)

Kurt Willis (phaze)

Richie Humphrey (devtooligan)

About the Tally stUNI Review

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

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

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/contracts folder of the `stUNI` repo was reviewed at commit 7de3a66.

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

- src/contracts/UniLst.sol
- src/contracts/WithdrawGate.sol
- src/contracts/WrappedUniLst.sol

Summary of Findings

Identifier	Title	Severity	Fixed
C-01	Self-transfers allows minting tokens	Critical	Fixed in pr #124
L-01	Unsafe downcast DepositId	Low	Fixed in pr #124
I-01	Fee Amount	Info	Fixed in pr #124
I-02	Incorrect implementation of the _calcStakeForShares function	Info	Fixed in pr #124

Known Issues

- The rewards distribution logic does not account for the holding period so an unfair amount of rewards can be obtained by staking a large amount of Uni (possible through a flashloan), calling `claimAndDistributeReward`, and then immediately unstaking for a profit. This vulnerability is mitigated by setting the delay on the `WithdrawGate` however a similar attack could still be carried out by whale. Limiting the payout amount will also help mitigate this issue since it would reduce the incentive for an attacker.

Rounding Analysis

The Tally team requested a review of how rounding affects the UniLst contract's operations. They had several concerns regarding the precision of share calculations and token transfers. Specifically, they sought answers to the following questions:

1. Can a single share value become so large that it prevents users from transferring small amount of tokens?

2. What is the maximum possible difference between the value a user requests to transfer and the value actually transferred?
3. How do the amount staked and the payout amount for the first reward distribution affect the precision of future share calculations?

This section presents the proved invariants of the UniLst contract and discusses their implications. Suggested code improvements based on these invariants can be found in Appendix A. Detailed proofs can be found in Appendix B.

Invariants

1. Total Tokens to Total Shares Ratio is Always Less Than 1:1
2. The maximum difference between the value a user requests to transfer and the value actually transferred is 1 wei.
3. Rounding-up the calculated shares ensures that the value of shares equals the user's requested amount.
4. The increase in user balance after addition of shares might be 1 wei more than the value of added shares and the decrease in user balance after subtraction of shares might be 1 wei more than the value of deducted shares

Implications

1. The value of a single share remains below 1 wei ensuring that users can always transfer smallest possible amount of tokens.
2. The amount staked and payout amount for the first reward distribution will not impact the precision of shares to the extent that it causes issues.
3. The value of the shares transferred/burned might be less than the value requested by the user with a maximum difference of 1 wei.
    - The receiver might receive 1 wei less than what the sender transferred.
4. The `UniLst.unstake` and the `UniLst.transfer` functions can be updated to round-up the number of shares burned or transferred.
    - This ensures that the value of shares matches the user's requested amount exactly.
    - The extra shares burned from rounding-up will have a value of less than 1 wei.
5. When \(y\) shares are added to a user, the increase in user's balance is either equal to the value of \(y\) shares or 1 wei more.
6. When \(y\) shares are deducted from a user, the decrease in user's balance is either equal to the value of \(y\) shares or 1 wei more.

Code Quality Recommendations

Please find code quality recommendations in Appendix C.

Detailed Findings

Critical Findings

[C-01] Self-transfers allows minting tokens

Anyone is able to transfer tokens to themselves and increase their balance by the transfer amount.

When transferring `UniLst` tokens, the internal `_transfer` function is used to update the sender and receiver share balances. The initial balances are loaded into memory before the share values are modified and updated independently.

```
function _transfer(address _sender, address _receiver, uint256 _value) interna
  // Read required state from storage once.
  Totals memory _totals = totals;
  HolderState memory _senderState = holderStates[_sender];
  HolderState memory _receiverState = holderStates[_receiver];

  // ...

  // Move underlying shares.
  {
    uint256 _shares = _calcSharesForStake(_value, _totals);
    _senderState.shares -= uint128(_shares);
    _receiverState.shares += uint128(_shares);
  }

  // ...


  // Write data back to storage once.
  holderStates[_senderRescoped] = _senderState;
  holderStates[_receiverRescoped] = _receiverState;

  // ...

}
```

The function does not handle the case when both holder states commit to the same storage slot, thereby overwriting any previously deducted balance. If the sender equals the receiver, then the tokens transferred will simply be added to the caller's balance without being subtracted.

Exploit scenario

The issue is seen when Alice transfers `1 ether` to herself.

```
function test_transferSenderEqualsReceiver() public {
  uint256 _stakeAmount = 1 ether;
  address _sender = address(0xa11ce);
  address _receiver = _sender;

  _stakeAmount = _boundToReasonableStakeTokenAmount(_stakeAmount);

  _mintAndStake(_sender, _stakeAmount);

  uint256 _originalSenderBalance = lst.balanceOf(_sender);

  vm.prank(_sender);
  lst.transfer(_receiver, _stakeAmount);

  assertEq(lst.balanceOf(_sender), _originalSenderBalance);
}
```

Her initial balance has doubled.

```
Failing tests:
Encountered 1 failing test in test/UniLst.t.sol:TransferAndReturnBalanceDiffs
[FAIL. Reason: assertion failed: 2000000000000000000 != 1000000000000000000] tes

Encountered a total of 1 failing tests, 0 tests succeeded
```

Recommendation

Include an early return in the `_transfer` function when the `sender` equals `receiver` while still emitting the `Transfer` event.

Further, make sure to include this scenario as test cases. Currently, these cases are discarded via the `_assumeSafeHolders` function.

```
function _assumeSafeHolders(address _holder1, address _holder2) internal view
  _assumeSafeHolder(_holder1);
  _assumeSafeHolder(_holder2);
  vm.assume(_holder1 != _holder2);
}
```

Low Findings

[L-01] Unsafe downcast DepositId

When the `depositId` on `UniStaker` exceeds 2^32, the UniLst logic truncates the id which leads to loss of funds.

NOTE: The team is aware of this issue and has stated the following.

> Creating a new deposit for an existing delegate on UniStaker costs about 221,000 gas. At 1 gwei and current prices, this works out to a little more than $0.50 to create a deposit. uint32.max is > $4.2 Billion, so gas spent creating deposits would have to be ~$2.2 Billion to get to this number.

> Assuming a 40 million per block gas limit, ~200 deposits could be created per block if the whole block was creating UniStaker deposits. If every block was full, it would take ~21.5 million blocks to reach uint32 max. At 12 seconds/block it would take 8.1 years to reach this point if every block was fully utilized to create UniStaker deposits.

We agree with this reasoning and assess this is as an unlikely scenario which is why the severity has been assessed at low.

Exploit scenario

Assumptions:

- UniStaker.nextDepositId = 4294967297 (uint32.max + 3). When unsafely downcasted to a uint32, this becomes 2.
- Assume UniStaker.depositId 2 is NOT owned by UniLst.

Steps:

1. Alice calls fetchOrInitializeDepositForDelegatee
2. Alice calls stake and deposits funds. This defaults to the DEFAULT_DELEGATE
3. Alice calls updateDeposit passing the new id returned by fetchOr.. in step 1
4. Alice's deposited funds are now stuck, she cannot call unstake to withdraw nor can she call updateDeposit to change her delegate since in both cases UniStaker reverts with Unauthorized

POC

```
    function test_depositId_unsafe_downcast() public {
        // unsafe downcast wraps around uint32.max to depositId NOT owned by LST

        address user1 = address(0x123);

        // depositId's 0 and 1 are used
        // This creates a stake in UniStaker that is NOT owned by LST with depos
        lst.STAKER().stake(0, address(0xdead));

        uint depositId_Uint32Max_plus_3 = uint256(type(uint32).max) + 3;

        // set next deposit id to 3 over uint32.max:
        // @dev We overwrote the UniStaker contract with a new setter function
        ModifiedUniStaker(address(lst.STAKER())).setNextDepositId(depositId_Uint

        // fetch depositId for user1
        lst.fetchOrInitializeDepositForDelegatee(user1);
        // this correctly stores the proper uint256 value for the depositId in st


        // call stake first
        // NOTE: If you call update first then you can't mint because UniStaker.s
        _mintAndStake(user1, 1e18);
        // staked with default delegatee

        // call update deposit with new depositId        lst.updateDeposit(IUniSt
        // holderStates[_account].depositId = uint32(IUniStaker.DepositIdentifie
        // this incorrectly updates the depositId on the holderState with id wit
        // 0x02 is the depositId created above that is NOT owned by LST

        // UNSTAKE reverts
        vm.prank(user1);
        vm.expectRevert(); // UniStaker.withdraw reverts with "unauthorized"
        lst.unstake(0.5e18);

        // UPDATE DEPOSIT reverts -- can't change the depositId
        vm.prank(user1);
        vm.expectRevert();
        lst.updateDeposit(IUniStaker.DepositIdentifier.wrap(0));
    }
```

Recommendation

The recommended solution is to use safe casting and revert if the id is larger than 32 bits using.

Informational Findings

[I-01] Fee Amount

The `feeAmount` is the amount (in wei) of stUNI value that will be issued to the `feeRecipient` when `claimAndDistributeReward` is called.

```
  function setFeeParameters(uint256 _newFeeAmount, address _newFeeCollector) exte
    _checkOwner();
    if (_newFeeAmount > payoutAmount) {
      revert UniLst__InvalidFeeParameters();
    }
    if (_newFeeCollector == address(0)) {
      revert UniLst__InvalidFeeParameters();
    }
```

There is a check in `setFeeParameters` that ensures the fee does not get set higher than the `payoutAmount`. However, even setting the fee to 100% of the payout amount wouldn't be fair to the stakers as they would not receive any rewards.

Additionally, due to the relationship between fee and payout, when the payout is changed, the fee will also likely have to be changed as well. This increases chance of mistakes, adds overhead to the process, and costs gas.

Recommendation

Change the `feeAmount` variable to be a percentage (in basis points) of the `payoutAmount` instead of a nominal value.

Cap the `feeAmount` be implementing a `MAX_FEE` amount constant or immutable which will ensure the fee cannot become unreasonable.

[I-02] Incorrect implementation of the _calcStakeForShares function

The `_calcStakeForShares` function is incorrect in the following ways:

- It checks `_totals.supply` to avoid divison by zero, but it should be checking `_totals.shares` instead, as it is the denominator in the divsion. Although the current implementation does not create a state where `_totals.supply > 0` and `_totals.shares == 0`, this could lead to DoS if the contract is updated to allow such state.
- It returns `0` when `_totals.supply` is zero instead of returning `_amount / SHARE_SCALE_FACTOR`.

```
  function _calcStakeForShares(uint256 _amount, Totals memory _totals) inte
    if (_totals.supply == 0) {
      return 0;
    }

    return (_amount * _totals.supply) / _totals.shares;
  }
```

Exploit scenario

Alice, a developer of UniLst contract, updates the contract. The updates allow for the contract to reach a state where `_totals.supply > 0` and `_totals.shares == 0`. The `_transfer` function which uses the `_calcStakeForShares` function fails with divison-by-zero error disallowing users to transfer stUNI.

Recommendation

Update the if condition in the `_calcStakeForShares` function to check for `_totals.shares` and to return `_amount / SHARE_SCALE_FACTOR` if it is zero.

Appendix A - Proposed Code Improvements for Rounding

The UniLst contract uses specific design patterns to handle rounding issues and prevent positions from becoming insolvent. By applying the proven invariants, the code can be simplified, reducing code complexity and improving readability.

The following section outlines each function and details the suggested changes based on the invariants.

General

- Add a `_calcSharesForStakeUp(_tokens, _totals)` internal function which rounds-up the amount of shares.
- Remove the condition that sets the user's `balanceCheckpoint` to the minimum of the current `balanceCheckpoint` and the user's balance, as the following improvements ensure that `balanceCheckpoint` is always less than or equal to the user's balance.
- Use the `_calcSharesForStakeUp` function in the `sharesForStake` to calculate the shares required for the `_amount`.

UniLst._stake

A snippet of `UniLst._stake` function:

```
    uint256 _balanceDiff = _calcBalanceOf(_holderState, _totals) - _initialBalan
    if (!_isSameDepositId(_calcDepositId(_holderState), DEFAULT_DEPOSIT_ID)) {
      _holderState.balanceCheckpoint = _holderState.balanceCheckpoint + uint96(_
    }
```

Proposed improvement of the above snippet:

```
    if (!_isSameDepositId(_calcDepositId(_holderState), DEFAULT_DEPOSIT_ID)) {
      _holderState.balanceCheckpoint = min(_holderState.balanceCheckpoint + uint
    }
```

The proposed updates ensures solvency of the user deposit for the following reasons:

1. The `balanceCheckpoint` is incremented by the same amount, `_amount`, that is deposited into the user's deposit.
2. Since shares are rounded-down, the user's balance might increase by `_amount - 1`. The `min(balanceCheckpoint + _amount, _calcBalanceOf(_holderState, _totals))` ensures that user's `balanceCheckpoint` remains less than or equal to the user's balance.

UniLst._unstake

A snippet of `UniLst._unstake` function:

```
     // Decreases the holder's balance by the amount being withdrawn
    uint256 _sharesDestroyed = _calcSharesForStake(_amount, _totals);
    _holderState.shares -= uint128(_sharesDestroyed);

    // By re-calculating amount as the difference between the initial and cu
    // amount unstaked is reflective of the actual change in balance. This r
    // less than the user requested by a small amount.
    _amount = _initialBalanceOf - _calcBalanceOf(_holderState, _totals);

    // cast is safe because we've validated user has sufficient balance
    _totals.supply = _totals.supply - uint96(_amount);
```

```

     // Write updated states back to storage.
     totals = _totals;
     holderStates[_account] = _holderState;
```

Proposed improvement of above snippet of code:

```
     // Decreases the holder's balance by the amount being withdrawn
    uint256 _sharesDestroyed = _calcSharesForStakeUp(_amount, _totals);
    _holderState.shares -= uint128(_sharesDestroyed);

    // cast is safe because we've validated user has sufficient balance
    _totals.supply = _totals.supply - uint96(_amount);
```

```

    _holderState.balanceCheckpoint = min(_holderState.balanceCheckpoint, _c
    // Write updated states back to storage.
    totals = _totals;
    holderStates[_account] = _holderState;
```

The proposed updates ensures solvency of the user deposit for the following reasons:

1. Rounding up the shares ensures that the value of destroyed shares is at least `_amount`, removing the need for recalculation.
2. The user's balance might decrease by `_amount + 1`. The `min(_holderState.balanceCheckpoint, _calcBalanceOf(_holderState, _totals))` ensures that user's `balanceCheckpoint` is less than or equal to their balance.
    - The extra 1-wei gets subtracted from the user's undelegated balance, resulting in a 1-wei addition to the default deposit.

UniLst._transfer

A snippet of `UniLst._unstake` function:

```
    // Move underlying shares.
    {
      uint256 _shares = _calcSharesForStake(_value, _totals);
      _senderState.shares -= uint128(_shares);
      _receiverState.shares += uint128(_shares);
    }

    // Due to truncation, it is possible for the amount which the sender'
    // amount by which the receiver's balance increases.
    uint256 _senderBalanceDecrease = _senderInitBalance - _calcBalanceOf(
    uint256 _receiverBalanceIncrease = _calcBalanceOf(_receiverState, _to

    // To protect the solvency of each underlying Staker deposit, we want
    // decreases by at least as much as the receiver's increases. Therefo
    // from the sender until such point as it is.
    while (_receiverBalanceIncrease > _senderBalanceDecrease) {
      _senderState.shares -= 1;
      _senderBalanceDecrease = _senderInitBalance - _calcBalanceOf(_sende
    }

    // Knowing the sender's balance has decreased by at least as much as
    // calculation of how much to move between deposits on the greater nu
    // when we update the receiver's balance checkpoint, we use the small
    // As a result, extra wei may be lost, i.e. no longer controlled by e
    // but are instead stuck permanently in the receiver's deposit. This
    // we've ensured the solvency of each underlying Staker deposit.

    if (!_isSameDepositId(_calcDepositId(_receiverState), DEFAULT_DEPOSIT_
      _receiverState.balanceCheckpoint += uint96(_receiverBalanceIncrease
    }
```

```

    // Write data back to storage once.
    holderStates[_senderRescoped] = _senderState;
    holderStates[_receiverRescoped] = _receiverState;
```

Proposed improvement of above snippet of code:

```
    // Move underlying shares.
    {
      uint256 _shares = _calcSharesForStakeUp(_value, _totals);
      _senderState.shares -= uint128(_shares);
      _receiverState.shares += uint128(_shares);
    }

    uint256 _senderBalanceDecrease = _value;
    if (!_isSameDepositId(_calcDepositId(_receiverState), DEFAULT_DEPOSIT_
      _receiverState.balanceCheckpoint += uint96(_value);
    }
    _senderState.balanceCheckpoint = min(_senderState.balanceCheckpoint, _
    // Write data back to storage once.
    holderStates[_senderRescoped] = _senderState;
    holderStates[_receiverRescoped] = _receiverState;
```

The proposed updates ensures solvency of the user deposit for the following reasons:

1. Rounding up the `_shares` transferred ensures the transferred value is at least `_value`.
2. The `_receiverState.balanceCheckpoint` is incremented by `_value`, which matches the amount moved from the sender's deposit to receiver's deposit.
3. The `min(_senderState.balanceCheckpoint, _calcBalanceOf(_senderState, _totals)` ensures the sender's balanceCheckpoint is less than or equal to the balance in case sender's balance decreases by `_value + 1`.
4. If the receiver's balance increases by `_value + 1`, the extra 1 wei is counted as undelegated balance and becomes a loss for the default deposit. However, since the Tally team plans to subsidize default deposit, it will remain solvent.

Appendix B. UniLst Contract Invariants Proofs

The proofs use the following variables.

- \(Ts\) = Total shares
- \(Ta\) = Total tokens
- scale_factor = SCALE_FACTOR used by the UniLst contract

1. Total Tokens to Total Shares Ratio is Always Less Than 1:1

The protocol uses a `SCALE_FACTOR` and sets the initial `Ta:Ts` ratio to `1:SCALE_FACTOR`. This means the UniLst contract mints `SCALE_FACTOR * tokens` shares for the first depositor. As a result, if the `Ta:Ts` ratio ever reaches \(1:1\), the initial depositor would make a profit of `SCALE_FACTOR - 1` times their initial investment.

Consider the following scenario:

1. At time \(0\), User Alice stakes \(10\) UNI in UniLst contract and receives \(10 * 1e18 * scale_factor\)
    - \(Ta = 10 * 1e18\), \(Ts = 10 * 1e18 * scale_factor\)
2. After time \(t\), the \(Ta:Ts\) ratio became \(1:1\) due to rewards.
3. Alice redeems her shares \(10 * 1e18 * scale_factor\) and receives \(10 * scale_factor\) UNI (\(10 * 1e18 * scale_factor\) wei).

Alice earns \((scale_factor - 1)\) times the initial amount as the profit after time \(t\).

Since the UniLst contract sets $scale\_factor$ to $1e10$, the time required for the initial depositor to achieve a profit of $(1e10 - 1)$ times their investment is extremely long. Thus, the $Ta:Ts$ ratio is unlikely to reach $1:1$ over the protocol's lifetime.

2. The maximum difference between the value a user requests to transfer and the value actually transferred is 1

Let

- a = Amount requested by the user (`unstake(a)`, `transfer(sender, receiver, a)`)
- y = Shares deducted/added to the user
- b = Amount transferred to the user (value of the y shares)

The `UniLst.unstake` and the `UniLst.transfer` functions round-down the amount of shares burned/transferred from the user.

$$y = \lfloor \frac{a * Ts}{Ta} \rfloor$$

$$b = \lfloor \frac{y * Ta}{Ts} \rfloor$$

Rouding down can be defined as

$$\frac{p}{q} = \lfloor \frac{p}{q} \rfloor + \frac{remainder}{q}$$

The $\frac{remainder}{q}$ is the value lost because of rounding down.

Then

$$y = \lfloor \frac{a * Ts}{Ta} \rfloor = \frac{a * Ts}{Ta} - \frac{k1}{Ta}$$

where $k1$ is the remainder, $k1 < Ta$

And

$$b = \lfloor \frac{y * Ta}{Ts} \rfloor = \frac{y * Ta}{Ts} - \frac{k2}{Ts}$$

where $k2$ is the remainder, $k2 < Ts$

substituting y

$$b = ((\frac{a * Ts}{Ta} - \frac{k1}{Ta}) * Ta) * \frac{1}{Ts} - \frac{k2}{Ts}$$

$$= \frac{a * Ts - k1}{Ts} - \frac{k2}{Ts}$$

$$= a - \frac{k1}{Ts} - \frac{k2}{Ts}$$

$$b = a - \frac{k1 + k2}{Ts}$$

$$a - b = \frac{k1 + k2}{Ts}$$

excluding decimals

$$ a - b = \lfloor \frac{k1 + k2}{Ts} \rfloor $$

Given $k1 < Ta$ and $k2 < Ts$, the difference can be estimated based on the $Ta:Ts$ ratio

Writing $k1$ in terms of Ts

$$ k1 = n * Ts + k3 $$

where $k3$ is the remainder, n is the quotient ($n \in \mathbb{N}$).

$$ \frac{k1 + k2}{Ts} = \frac{n * Ts + k3 + k2}{Ts} = n + \frac{k2 + k3}{Ts}$$

Using the constraints

$$ k2 < Ts,\ k3 < Ts \implies 0 <= k2 + k3 < 2*Ts $$

and

$$ k1 < Ta \implies n < \frac{Ta}{Ts} \implies n <= \lfloor \frac{Ta - 1}{Ts} \rfloor $$

bounds the difference $a - b$ to

$$ 0 <= a - b <= n + 1 \implies 0 <= a - b <= \lfloor \frac{Ta - 1}{Ts} \rfloor + 1 $$

Since the $\frac{Ta}{Ts}$ is always less than 1 for the UniLst contract,

$$ 0 <= a - b <= 1 \implies a - b == 0\ or\ 1 $$

As a result, the value of the shares transferred might be less than the requested value by at-most 1.

3. Rounding-up the calculated shares ensures that the value of shares equals the user's requested amount.

The UniLst contract can round-up the division used to calculate the amount of shares transferred(y) to ensure that the transferred value is always at-least the requested value.

Because the $\frac{Ta}{Ts}$ is always less than 1 for the UniLst contract, the worth of y shares will be exactly equal to requested amount.

The proof can be derived by defining the ceil divison as

$$ \lceil \frac{p}{q} \rceil = \frac{p}{q} + \frac{m}{q} $$

where

$$ m = \left\{ \begin{array}{ll} 0 & \mbox{if } p\ \%\ q = 0 \\ q - (p\ \%\ q) & \mbox{if } p\ \%\ q > 0 \end{array} \right. $$

and $m < q$

If the `UniLst.unstake` and `UniLst.transfer` functions are updated to round-up the shares transferred. Then,

$$ y = \lceil \frac{a * Ts}{Ta} \rceil = \frac{a * Ts}{Ta} + \frac{m}{Ta}$$

where $m < Ta$

Above derivations can be used with $m = -k1$ to calculate the bounds for $b - a$:

$$ 0 <= b - a <= \lfloor \frac{Ta - 1}{Ts} \rfloor$$

and

$$ \frac{Ta}{Ts} < 1 \implies b - a = 0$$

4. The increase in user balance after addition of shares might be 1 more than the value of added shares

Let

- y = shares added
- Ri = User(receiver) initial shares
- Rc = Ri + y = User shares after transfer.

Then

$$ \lfloor \frac{Rc * Ta}{Ts} \rfloor - \lfloor \frac{Ri * Ta}{Ts} \rfloor == \lfloor \frac{y * Ta}{Ts} \rfloor \ or\ == \lfloor \frac{y * Ta}{Ts} \rfloor + 1 $$

Proof:

$$ receiverInitialBalance = \lfloor \frac{Ri * Ta}{Ts} \rfloor = \frac{Ri * Ta}{Ts} - \frac{l1}{Ts} $$

$$ receiverCurrentBalance = \lfloor \frac{Rc * Ta}{Ts} \rfloor = \frac{Rc * Ta}{Ts} - \frac{l2}{Ts} $$

$$ receiverBalanceIncrease = receiverCurrentBalance - receiverInitialBalance $$

$$ = (\frac{Rc * Ta}{Ts} - \frac{l2}{Ts}) - (\frac{Ri * Ta}{Ts} - \frac{l1}{Ts}) $$

$$ = \frac{(Rc - Ri) * Ta}{Ts} + \frac{l1 - l2}{Ts} $$

$$ = \frac{y * Ta}{Ts} + \frac{l1 - l2}{Ts} $$

\[= (\lfloor \frac{y * Ta}{Ts} \rfloor + \frac{l3}{Ts}) + \frac{l1 - k2}{Ts} \]

\[= \lfloor \frac{y * Ta}{Ts} \rfloor + \frac{l3 + l1 - l2}{Ts} \]

The above values are in infinite precision, upon converting to finite precision

\[receiverBalanceIncrease = \lfloor \frac{y * Ta}{Ts} \rfloor + \lfloor \frac{l3 + l1 - l2}{Ts} \rfloor \]

- l1 = remainder of \(Ri * Ta\) divided by \(Ts\)
- l2 = remainder of \(Rc * Ta\) divided by \(Ts\)
- l3 = remainder of \(y * Ta\) divided by \(Ts\)

Each of \(l1\), \(l2\), \(l3\) is less than \(Ts\) resulting in

\[0 <= l3 + l1 - l2 < 2*Ts \]

\[\lfloor \frac{k3 + l1 - l2}{Ts} \rfloor = 0\ or\ = 1 \]

\[receiverBalanceIncrease = \lfloor \frac{y * Ta}{Ts} \rfloor \ or = \lfloor \frac{y * Ta}{Ts} \rfloor + 1\]

A similar proof can show that when a user transfers \(y\) shares, the decrease in their balance might be 1 wei more than the value of \(y\) shares.

Let

- \(Si\) = User(sender) initial shares
- \(Sc\) = \(Si - y\); \(y\) shares are deducted from user balance

Then

\[\lfloor \frac{Si * Ta}{Ts} \rfloor - \lfloor \frac{Sc * Ta}{Ts} \rfloor == \lfloor \frac{y * Ta}{Ts} \rfloor \ or\ == \lfloor \frac{y * Ta}{Ts} \rfloor + 1 \]

\[senderBalanceDecrease = senderInitialBalance - senderCurrentBalance\]

\[== \lfloor \frac{y * Ta}{Ts} \rfloor \ or\ == \lfloor \frac{y * Ta}{Ts} \rfloor + 1 \]

Appendix C. Code Quality Recommendations

- **Name on the ERC-20 token metadata differs from name used for EIP712**
  This is not a security risk since EIP712 uses the contract address in the `DOMAIN_SEPARATOR`. Consider using the same name in both places for consistency and to avoid confusion.

- **Unexpected behavior from `depositForDelegatee`**
  It is possible that new default delegate to have a previous deposits i.e a deposit id. The function would return deposit-id corresponding to the delegatee till the update to default delegatee and would return default deposit id after the update.. Add documentation around this issue to inform integrators.

- **Add sanity check for amount > 0 to `unstake`**
  This function kicks off a series of events, especially if WithdrawGate.delay > 0. Mistaken users will waste gas and it also opens up a griefing attack via spamming events.

- **Revert in the `_updateDeposit` function**
  If the owner of `newDepositId` is not `UniLst` This avoids unnecessary gas costs and provides a clearer error message, as the function will eventually revert during the `UniStaker.stakeMore` call.

- **Update the `_updateDeposit` function**
  Return early if both `_oldDepositId` and `_newDepositId` are `DEFAULT_DEPOSIT_ID`. In this scenario, the function is a no-op, and returning early will save gas.

- **Rename the `_amount` parameter** of the `_calcStakeForShares` function to `_shares` to improve readability.

- **Update the `WrappedUniLst.wrap` function to use `UniLst.sharesForStake` to determine number of shares transferred**
  Currently, the `wrap` function calculates this by computing the difference in shares before and after the transfer. Using the `sharesForStake` reduces external calls by one, thereby saving gas.

- **Remove unused error**
  The `WithdrawalCompleted` error in `WithdrawGate` is not used and may be removed.

- **Public functions not used internally could be marked external**
  Two functions apply the public modifier but they are not called internally so they may be changed to external. `WrappedUniLst.setDelegatee` and `UniLst.nonces`