Tally stGov Review

March 17, 2025

Prepared for DappHero Corp

Conducted by:

Vitor Frasson (hake)

Richie Humphrey (devtooligan)

Kurt Willis (phaze)

Vara Prasad Bandaru (s3v3ru5)

About the Tally stGov Review

Tally provides a governance platform that enables decentralized decision-making for blockchain organizations. The stGov framework builds upon Tally's Staker contract to create a liquid staking token (LST) for governance tokens staked on Staker. This system allows token holders to stake their governance tokens to earn rewards while maintaining liquidity through the LST representation, simultaneously allowing for changes to the delegation of governance voting power. This review examines the stGov implementation, focusing on the security and functionality of the liquid staking token framework.

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 was reviewed at commit 40fcaa5.

The following 14 files were in scope:

- src/FixedGovLst.sol
- src/FixedLstAddressAlias.sol
- src/GovLst.sol
- src/WithdrawGate.sol
- src/WrappedGovLst.sol
- src/auto-delegates/OverwhelmingSupportAutoDelegate.sol
- src/auto-delegates/extensions/AutoDelegateBravoGovernor.sol
- src/auto-delegates/extensions/AutoDelegateOpenZeppelinGovernor.sol
- src/auto-delegates/extensions/BlockNumberClockMode.sol
- src/auto-delegates/extensions/TimestampClockMode.sol
- src/extensions/FixedGovLstOnBehalf.sol
- src/extensions/FixedGovLstPermitAndStake.sol
- src/extensions/GovLstOnBehalf.sol
- src/extensions/GovLstPermitAndStake.sol

The stGov protocol implements liquid staking for governance tokens through a wrapper contract that automates rewards claiming and delegation power management. The system uses a share-based accounting system to track user deposits and distribute rewards, with additional features for auto-delegation and deposit management based on earning power thresholds.

The review identified 3 LOW severity and 7 INFORMATIONAL issues.

Assumptions and Trust Model

Based on discussions with the Tally team, we understand the following design decisions and trust assumptions that informed our analysis:

1. **Earning Power and Balance Relationship**: The earning power is expected to be a scale factor of a deposit's balance, where the scale factor depends on the actions and activity of the chosen delegatee. If the earning power drops below a minimum

threshold relative to the balance, the deposit should be overridden and delegated to the default delegatee.

2. **Earning Power Calculator (EPC)**: While the EPC design allows for flexibility, it's assumed that DAOs will not implement calculators that allow earning power to fluctuate dramatically over short time horizons, as this would be detrimental to all depositors in the underlying Staker contract.

3. **Override Mechanism and Tips**: The override system allows users to claim tips for correcting deposits with low earning power. While there are potential game-theoretic attacks to extract value through self-dealing, the team believes these are mitigated by:
    - The uncertain nature of who will claim the tip (other solvers or subsidized bots might claim it first)
    - The work required to set up such an attack relative to the potential reward

4. **Staker Contract Administration**: The Staker contract (not in scope for this review) grants significant powers to administrators, including the ability to set the earning power calculator and bump tips. The team has confirmed that this administrative role is expected to be controlled by a DAO.

5. **Token Compatibility**: The contracts are primarily designed to work with standard ERC-20 tokens, preferably with 18 decimals. Non-standard tokens (e.g., tokens with fees on transfer, unusual decimals, or non-standard behavior) may cause unexpected behaviors.

Summary of Findings

Identifier	Title	Severity	Fixed
L-01	Unsafe downcasting in the GovLst contract	Low	PR#66
L-02	Max override tip cap is unrealistic for certain valued tokens	Low	PR#78
L-03	Missing validation in constructor for critical parameters	Low	PR#65
I-01	Fee extraction through deposit override manipulation	Info	PR#78

I-02	DepositId overflow risk in HolderState struct	Info	Ack
I-03	Voting window check only enforces minimum timepoint	Info	Ack
I-04	Updates to AutoDelegateBravoGovernor	Info	PR#64
I-05	Potential Out of Gas in claimAndDistributeReward	Info	PR#67
I-06	Inconsistent validation of minimum earning power requirements	Info	Ack. See detail for response from project team.
I-07	Compatibility issues with non-standard tokens	Info	PR#67

Additional Review

As part of the review of the fixes, the project team has also asked us to review some of the fixes for issues identified in the Cantina contest including:

- [Cantina 56] Change alias address calculation
- [Catina #68] Wrong Timepoint Used For Quorum Check In AutoDelegate
- [Cantina 95] Remove tips for caller of override methods

These prs have been reviewed with no additional issues noted.

Code Quality

See Code Quality Recommendations appendix for detailed suggestions.

Detailed Findings

Low Findings

[L-01] Unsafe downcasting in the GovLst contract

The GovLst contract contains several instances of type downcasting without proper safety checks, which could lead to silent truncation and incorrect accounting. This issue primarily affects the handling of token amounts and share calculators. This issue is specifically prominent in tokens with unusually high supply or low value.

The contract performs several numeric type conversions without using the appropriate SafeCast functions. These unsafe conversions can lead to value truncation when handling large amounts, potentially resulting in accounting errors and unexpected behavior.

Key instances of unsafe casting include:

1. In `_transferFeeInShares()`, a `uint160` is unsafely cast to `uint128`:

```
holderStates[_feeReceiver].shares += uint128(_feeShares);
```

2. In `_transfer()`, a `uint256` is unsafely cast to `uint128`:

```
_receiverState.shares += uint128(_shares);
```

3. In `_stake()`, the contract uses direct casting to `uint96` for supply:

```
_totals.supply = _totals.supply + uint96(_amount);
```

4. In `_unstake()`, similar unsafe downcasting occurs:

```
_totals.supply = _totals.supply - uint96(_amount);
```

In these cases, if the calculated values exceed the destination type's maximum value, the values will silently truncate rather than revert, potentially leading to incorrect accounting and loss of tokens.

Recommendation

Replace all unsafe type casts with the SafeCast library functions that are already imported. For example:

```diff
- holderStates[_feeReceiver].shares += uint128(_feeShares);
+ holderStates[_feeReceiver].shares += SafeCast.toUint128(_feeShares);
- _receiverState.shares += uint128(_shares);
+ _receiverState.shares += SafeCast.toUint128(shares);
- _totals.supply = _totals.supply + uint96(_amount);
+ _totals.supply = _totals.supply + SafeCast.toUint96(_amount);
- _totals.supply = _totals.supply - uint96(_amount);
+ _totals.supply = _totals.supply - SafeCast.toUint96(_amount);
```

[L-02] Max override tip cap is unrealistic for certain valued tokens

The `GovLst` contract sets a hardcoded `MAX_OVERRIDE_TIP_CAP` value of 2000e18, which represents the maximum value that can be set for the `maxOverrideTip` parameter. This absolute value approach fails to account for the varying token values for `STAKE_TOKEN`. For example, if token was as valuable as ETH, 2000e18 would be roughly $40M.

```
/// @notice Maximum value to set the max override tip.
uint256 public immutable MAX_OVERRIDE_TIP_CAP = 2000e18;
```

Recommendation

Consider defining the `MAX_OVERRIDE_TIP_CAP` relative to the total supply or based on a percentage of another value in the system, rather than using an absolute token amount. For example adding a variable to track total amount of staked/unstaked tokens and take a percentage from that.

Additionally, allow the contract owner to set this value during initialization based on the specific token's value and the intended economics of the system.

We recommend initializing `MAX_OVERRIDE_TIP_CAP` to zero, forcing the deployer to choose it's own value and diminishing chances of messing up deployment. Also transferring the check to the internal function "_setMaxOverrideTip()". So initially it must be bigger than zero.

```diff
- uint256 public immutable MAX_OVERRIDE_TIP_CAP = 2000e18;
+ uint256 public immutable MAX_OVERRIDE_TIP_CAP;
  constructor(ConstructorParams memory _params)
    Ownable(_params.initialOwner)
    EIP712(_params.rebasingLstName, _params.version)
  {
    // ...
+   MAX_OVERRIDE_TIP_CAP = _params.maxOverrideTipCap;
    // ...
  }
```

This approach would provide flexibility to appropriately set the cap based on the specific token's value and the economics of each deployment.

[L-03] Missing validation in constructor for critical parameters

The `GovLst` contract fails to validate the `minQualifyingEarningPowerBips` and `maxOverrideTip` parameters in the constructor against their respective maximum caps, while these validations are correctly implemented in the corresponding setter functions. This could allow the contract to be deployed with values exceeding the intended maximum limits.

The `GovLst` contract defines maximum cap values for two critical parameters:

- `MINIMUM_QUALIFYING_EARNING_POWER_BIPS_CAP` (20,000)

- `MAX_OVERRIDE_TIP_CAP` (2000e18)

These caps are enforced in the setter functions (`setMinQualifyingEarningPowerBips` and `setMaxOverrideTip`), which validate the input values against their respective caps before calling the internal implementation functions:

```
function setMinQualifyingEarningPowerBips(uint256 _minQualifyingEarningPowerBips
  _checkOwner();
  if (_minQualifyingEarningPowerBips > MINIMUM_QUALIFYING_EARNING_POWER_BIPS_CAP
    revert GovLst__InvalidParameter();
  }
  _setMinQualifyingEarningPowerBips(minQualifyingEarningPowerBips);
}
```

However, in the constructor, these validations are bypassed as the internal implementation functions are called directly:

```
constructor(ConstructorParams memory _params) {
  // ...
  _setMaxOverrideTip(params.maxOverrideTip);
  _setMinQualifyingEarningPowerBips(params.minQualifyingEarningPowerBips);
  // ...
}
```

The internal functions `_setMaxOverrideTip` and `_setMinQualifyingEarningPowerBips` only emit events and update the state variables without performing any validation:

```
function setMaxOverrideTip(uint256 maxOverrideTip) internal virtual {
  emit MaxOverrideTipSet(maxOverrideTip, _maxOverrideTip);
  maxOverrideTip = _maxOverrideTip;
}
function setMinQualifyingEarningPowerBips(uint256 minQualifyingEarningPowerBips)
  emit MinQualifyingEarningPowerBipsSet(minQualifyingEarningPowerBips, _minQuali
  minQualifyingEarningPowerBips = _minQualifyingEarningPowerBips;
}
```

This inconsistency creates a scenario where the contract could be deployed with values that exceed the defined caps, which would otherwise be rejected by the setter functions after deployment.

Recommendation

Modify the internal setter functions to include the validation, ensuring consistent behavior in both the constructor and external setter functions:

```
function _setMaxOverrideTip(uint256 _maxOverrideTip) internal virtual {
+  if (_maxOverrideTip > MAX_OVERRIDE_TIP_CAP) {
+    revert GovLst__InvalidParameter();
+   }
    emit MaxOverrideTipSet(maxOverrideTip, _maxOverrideTip);
    maxOverrideTip = _maxOverrideTip;
}
function _setMinQualifyingEarningPowerBips(uint256 _minQualifyingEarningPowerBip
+   if (_minQualifyingEarningPowerBips > MINIMUM_QUALIFYING_EARNING_POWER_BIPS_CA
+     revert GovLst__InvalidParameter();
+   }
    emit MinQualifyingEarningPowerBipsSet(minQualifyingEarningPowerBips, _minQuali
    minQualifyingEarningPowerBips = _minQualifyingEarningPowerBips;
}
```

Informational Findings

[I-01] Fee extraction through deposit override manipulation

The GovLst contract allows any user to manipulate the delegatee override system to extract value from the protocol through self-dealing. An attacker can create multiple delegatee deposits and repeatedly trigger override events to collect tips at the expense of all token holders.

The GovLst contract implements mechanisms that allow anyone to override deposits with low earning power, redirecting them to the default delegatee. These mechanisms include functions `enactOverride()` and `revokeOverride()`, which pay the caller a tip for performing these actions.

The vulnerability arises from how these tips are distributed and funded. When an override is enacted or revoked:

1. A tip amount is specified by the caller (up to `maxOverrideTip`)
2. The tip is not taken from the affected deposit but is instead created by minting new shares to the tip receiver via `_transferFeeInShares()`
3. These new shares dilute all existing token holders, effectively extracting value from the entire protocol

An attacker can exploit this by:

1. Creating multiple delegatee deposits at once
2. Intentionally allowing them to become inactive (low earning power)
3. Calling `enactOverride()` in bulk to override their own deposits, receiving tips for each
4. Later calling `revokeOverride()` once the deposits meet the minimum earning power threshold
5. Repeating this cycle to continuously extract value

The scale of this attack could be magnified by creating a large number of delegatee deposits initially and then overriding them all simultaneously, maximizing the value extraction in a single transaction or series of transactions.

Recommendation

Since this design choice allows for flexibility and customization, the project may opt not to modify it.

Regardless of the decision, we strongly recommend clearly documenting the risks and related factors.

One idea to address this vulnerability, would be to charge a flat fee from the affected deposit rather than minting new shares or that dilute all holders or alternatively increase the cost of creating delegatee deposits to ensure the cost of attack exceeds potential gains.

[I-02] DepositId overflow risk in HolderState struct

The GovLst contract stores a `depositId` in the `HolderState` struct as a `uint32`, but `Staker.DepositIdentifier` is a `uint256`. An attacker can call `stake()` or `fetchOrInitializeDepositForDelegatee()` repeatedly to create new deposit IDs until they exceed `type(uint32).max`, making it impossible to create or initialize new delegatees, effectively denying service for new delegatee assignments. This attack is only feasible if gas costs are extremely low, making the likelihood of this attack very low.

In the GovLst contract, the `HolderState` struct contains a `depositId` field that is stored as a `uint32`:

```
struct HolderState {
  uint32 depositId;
  uint96 balanceCheckpoint;
  uint128 shares;
}
```

However, the `Staker.DepositIdentifier` type that represents deposit IDs in the Staker contract is a `uint256`. When storing a deposit ID in the `HolderState` struct, the contract converts it using the `_depositIdToUInt32` function:

```
function _depositIdToUInt32(Staker.DepositIdentifier _depositId) internal pure v
  return SafeCast.toUint32(Staker.DepositIdentifier.unwrap(_depositId));
}
```

This creates a vulnerability where an attacker can call `stake()` or `fetchOrInitializeDepositForDelegatee()` repeatedly with different delegatees, generating new deposit IDs until the values exceed `type(uint32).max` (4,294,967,295).

Once this happens, any attempt to create or initialize new delegatees will fail with an overflow error during the `SafeCast.toUint32()` operation.

The vulnerable code paths include:

1. `updateDeposit()` which calls `_depositIdToUInt32()` when setting `holderStates[_account].depositId`.
2. `_updateDeposit()` which assigns deposit IDs through `_holderState.depositId = depositIdToUInt32(newDepositId)`.

The impact of this vulnerability is high. Once triggered, it would prevent any new delegatee assignments in the contract. Users would be unable to delegate their staked tokens to new addresses, significantly limiting functionality and potentially locking voting power distribution. This effectively constitutes a denial of service for a core feature of the governance liquid staking token.

The likelihood of exploitation is very low. While it would require a malicious actor to make a large number of transactions (potentially up to 4 billion) to reach the uint32 maximum, this could be achieved over time, especially if gas costs are low. The attack doesn't require special privileges and could be executed by any user with sufficient funds to pay for transaction fees.

Recommendation

Modify the `HolderState` struct to use a larger data type for the `depositId` field:

```
struct HolderState {
-  uint32 depositId;
+  uint256 depositId;
  uint96 balanceCheckpoint;
  uint128 shares;
}
```

Additionally, remove `_depositIdToUInt32` function to eliminate the type conversion:

```
- function _depositIdToUInt32(Staker.DepositIdentifier _depositId) internal pure
-   return SafeCast.toUint32(Staker.DepositIdentifier.unwrap(_depositId));
- }
```

This change would prevent the potential overflow issue, though it would increase the storage cost of the `HolderState` struct by requiring an additional storage slot.

[I-03] Voting window check only enforces minimum timepoint

The `_isWithinVotingWindow()` function only verifies that the current timepoint is after the start of the voting window (calculated as `_proposalDeadline - votingWindow`), but does not check that the current timepoint is before the proposal deadline. This means votes could technically be cast after the proposal deadline has passed.

Additionally, if the `votingWindow` value is larger than the `_proposalDeadline` value, this calculation could underflow, especially if the clock is based on a non-standard genesis. This would result in an unexpectedly large start time value.

Recommendation:
Update the function to properly check both the lower and upper bounds of the voting window:

```
function _isWithinVotingWindow(uint256 _proposalDeadline) internal view virtual r
-     return clock() >= (_proposalDeadline - votingWindow);
+     uint256 start = votingWindow > _proposalDeadline ? 0 : _proposalDeadline - v
+     return clock() >= start && clock() <= _proposalDeadline;
}
```

This change ensures that votes can only be cast within the actual voting window period and handles the case where `votingWindow` might be larger than `_proposalDeadline`.

[I-04] Updates to AutoDelegateBravoGovernor

The `AutoDelegateBravoGovernor` contract has two issues related to its integration with Compound's `GovernorBravoDelegate`:

1. **Clock Mode Mismatch**: Compound's governance system uses block numbers for its clock mode, but the `AutoDelegateBravoGovernor` contract does not explicitly inherit from or implement a specific clock mode. This could lead to inconsistencies when calculating voting windows and timepoints.
2. **Voting Period Constraints**: The contract does not enforce or document Compound's specific voting period constraints. Compound's governance has fixed constants for voting periods:

```
// In GovernorBravoDelegate.sol
uint public constant MIN_VOTING_PERIOD = 5760; // About 24 hours
uint public constant MAX_VOTING_PERIOD = 80640; // About 2 weeks
```

Without this information, users might configure voting windows that are incompatible with Compound's governance system.

Recommendation:

1. Update the `AutoDelegateBravoGovernor` contract to inherit from `BlockNumberClockMode` to align with Compound's use of block numbers for timepoints:

```
-abstract contract AutoDelegateBravoGovernor {
+abstract contract AutoDelegateBravoGovernor is BlockNumberClockMode {
```

2. Either enforce Compound's voting period constraints in the contract or add documentation to guide users:

```
/// @title AutoDelegateBravoGovernor
/// @author [ScopeLift](https://scopelift.co)
/// @notice Extension for the OverwhelmingSupportAutoDelegate that integrates wi
/// @dev This contract provides implementations for the abstract functions in Ove
/// that are specific to Governor Bravo contracts.
+/// @dev Note: Compound's Governor Bravo enforces MIN_VOTING_PERIOD = 5760 (abou
+/// MAX_VOTING_PERIOD = 80640 (about 2 weeks). When deploying for Compound gove
+/// the MIN_VOTING_WINDOW and MAX_VOTING_WINDOW are set to match these values.
```

[I-05] Potential Out of Gas in `claimAndDistributeReward`

The `claimAndDistributeReward` function in the GovLst contract allows a caller to claim rewards from multiple deposit IDs in a single transaction. However, as the number of deposit IDs increases, there's a risk that the transaction could exceed block gas limits, preventing the collection of all rewards simultaneously.

In the implementation:

```
function claimAndDistributeReward(
  address _recipient,
  uint256 _minExpectedReward,
  Staker.DepositIdentifier[] calldata _depositIds
) external virtual {
  // ...
  // Claim the reward tokens earned by the LST for each deposit
  uint256 _rewards;
  for (uint256 _index = 0; _index < _depositIds.length; _index++) {
      _rewards += STAKER.claimReward(_depositIds[_index]);
  }
  // ...
}
```

If the LST manages a large number of deposits, a caller might only be able to process a subset of deposit IDs in a single transaction. Since rewards for all deposit IDs cannot be collected at once, this creates a timing difference between when rewards are available to direct stakers versus when they become available to LST holders. This lag can temporarily reduce the value of liquid staking tokens compared to direct staking, as direct stakers have more immediate access to their earned rewards.

Recommendation:
Document this constraint in protocol documentation so users can plan the `payoutAmount`

and earnings power calculations accordingly, understanding that rewards may not be immediately and fully reflected in the LST value.

[I-06] Inconsistent validation of minimum earning power requirements

The `GovLst` contract implements a mechanism to prevent users from updating their deposits to ones with insufficient earning power (below `minQualifyingEarningPowerBips`), but this validation is inconsistently applied across different functions. The validation is missing from `_stake()` and some conditions within `_updateDeposit()`, allowing users to stake tokens into or interact with deposits that don't meet the minimum earning power requirements.

The `GovLst` contract uses `_revertIfInvalidDeposit()` to verify that a deposit's earning power meets the minimum threshold. However, this validation is not uniformly applied across all functions that interact with deposits.

The validation is missing from the following key functions:

1. `_stake()` - When a user calls `stake()`, tokens are added to their current deposit without validating whether the deposit meets the minimum earning power requirements.

2. In `_updateDeposit()`, the `_revertIfInvalidDeposit()` check is only applied in certain conditions, but it's not called in the condition where `_isSameDepositId(_oldDepositId, _newDepositId)`, which means users can continue staking to the same deposit even if it no longer meets the requirements.

Project Response

The project team responded to this finding with the following statement.
tldr; There is no rule stating that customers may never stake more tokens when their selected delegate has been overridden, and enforcing such a rule would reduce customizability and degrade the UX.

From the project team:

> If multiple users want to hold LST tokens and delegate their voting weight to the same delegatee, they accomplish this by calling updateDeposit with the same depositId. The LST owns one deposit per delegatee, and all stake intended to delegate to that delegatee goes in the same depositId.

> If a given deposit is no longer earning sufficient rewards (presumably because of the inaction of the delegatee associated with that deposit), the enactOverride method is

> used to move that deposit's delegatee temporarily over the default delegatee. The documented assumption here is that the default delegatee will always earn sufficient rewards for the LST. It's the job of the LST owner to use the override methods to keep deposits up to date. If the LST owner fails to do this, their product will offer substandard yield and be unsuccessful.

> When a user stakes more tokens, or receives them via transfer, those tokens are placed into the deposit they have specified via updateDeposit, or go to the default deposit if no deposit has been chosen. We do not think it's appropriate to enforce in the contracts that the user's chosen deposit is earning sufficient rewards at this point.

> The user would have to resolve this by updating their deposit, effectively choosing a new delegatee, which could be bad UX. If a user is, for example, trying to transfer tokens to an address with an invalid deposit via a normal wallet, they would see the transaction reverting with no clear cause, and no obvious way to fix this. We think this would make bad UX.

> It may be appropriate for the client to force users, or strongly encourage them, to update their deposit before (or while) staking more tokens. This is a client-side decision, and the UX can be appropriately designed at that level. Since users may interact with the contracts via many clients, including normal wallets that support ERC20, enforcing it at the contract level seems inappropriate.

> We *do* prevent users from intentionally choosing a deposit that is invalid via updateDeposit. If the user is in the process of choosing a new delegatee, it feels appropriate to enforce they choose one whose actions are resulting in sufficient earning power, i.e. an active delegatee.

> More importantly, if users could choose invalid deposits at any time, they could grief the LST owner by constantly creating new deposits (that would not qualify for sufficient rewards) and updating to those deposits, forcing the LST owner to play a continuous cat and mouse game of overriding deposits. That is why we *do* enforce valid deposits in the updateDeposit method.

Recommendation

Add consistent validation across all relevant functions:

1. Add validation to `_stake()`:

```
function _stake(address _account, uint256 _amount) internal virtual returns (uin
    // Read required state from storage once.
    Totals memory _totals = totals;
    HolderState memory _holderState = holderStates[account];
+   revertIfInvalidDeposit(calcDepositId(_holderState));
}
```

2. Ensure validation is applied consistently in `_updateDeposit()` by adding it to the missing condition:

```
} else if (_isSameDepositId(_oldDepositId, _newDepositId)) {
    _holderState.balanceCheckpoint = uint96(balanceOf);
    STAKER.withdraw(DEFAULT_DEPOSIT_ID, uint96(_undelegatedBalance));
    STAKER.stakeMore(_newDepositId, uint96(_undelegatedBalance));
+   revertIfInvalidDeposit(newDepositId);
}
```

[I-07] Compatibility issues with non-standard tokens

The GovLst and Staker contracts assume standard ERC20 behavior for the STAKE_TOKEN. However, several token variants exist that could cause issues with these contracts, including tokens with non-standard decimals (both more and less than 18) and tokens with special transfer logic like fee-on-transfer mechanisms. This informational finding explores the potential issues that could arise when using these non-standard tokens with the GovLst and Staker systems.

Tokens with Fewer Than 18 Decimals

GovLst Issues

- **Hardcoded 18 Decimals**: The GovLst contract's `decimals()` function returns a hardcoded 18, regardless of the underlying STAKE_TOKEN's decimals:

  ```
  function decimals() external pure virtual override returns (uint8) {
      return 18;
  }
  ```

- **User Experience**: Users might expect 1 STAKE_TOKEN to map to 1 LST unit, but the decimal mismatch would make the LST balance appear different than expected.

Staker Issues

- **Less Susceptible**: Since Staker operates in raw token units without hardcoded decimal assumptions, it's less prone to issues from decimal mismatches than GovLst.

- **Earning Power Calculation**: If the external `earningPowerCalculator` assumes STAKE_TOKEN has 18 decimals, rewards could be miscalculated when using tokens with fewer decimals.

Tokens with More Than 18 Decimals

GovLst Issues

- **Overflow Risks**: The contract uses `SafeCast` to downcast values to limited types:

  ```
  _totals.supply = _totals.supply + uint96(_amount);
  _holderState.shares = _holderState.shares + _newShares.toUint128();
  ```

  If STAKE_TOKEN has 24 decimals, casting to `uint96` risks overflow for larger token amounts.

- **Share Calculation Issues**: With `SHARE_SCALE_FACTOR = 1e10`, a token value with 24 decimals would result in shares with 34 decimals, potentially causing precision problems in subsequent divisions.

- **Reward Parameters**: Values like `payoutAmount` stored in `uint80` could overflow for tokens with high decimals:

  ```
  RewardParameters {
    uint80 payoutAmount;
    uint16 feeBips;
    address feeCollector;
  }
  ```

Staker Issues

- **Type Constraints**: Deposits store balances as `uint96`:

  ```
  struct Deposit {
    uint96 balance;
    // ...
    uint96 earningPower;
    // ...
  }
  ```

  If STAKE_TOKEN has 24 decimals, large stakes could overflow these fields, causing reverts.

- **Scaling Factor**: The contract uses `SCALE_FACTOR = 1e36` for reward calculations, which is already quite large. With high-decimal tokens, reward calculations might approach the limits of `uint256`.

Tokens with Transfer Fees

Incorrect Accounting

- **Supply Discrepancy**: In GovLst, if a user transfers 100 tokens with a 1% fee, the contract receives 99 but records 100 in `totals.supply`, creating a permanent discrepancy.

- **Deposit Balance Inconsistency**: When GovLst stakes the received amount to Staker, it transfers 99 tokens but records 100 in the deposit balance, potentially allowing users to withdraw more than is available.

- **Unstaking Issues**: During unstaking, if the contract attempts to withdraw 100 tokens but only 99 are available, operations might fail or users might receive less than expected.

Operational Disruptions

- **Transfer Failures**: Fee-on-transfer mechanisms might cause transfers to fail unexpectedly if contracts don't account for the reduced amounts received.

- **Reward Calculation Errors**: The reward distribution relies on accurate accounting of staked tokens. Transfer fees create discrepancies in these calculations, potentially unfairly distributing rewards.

ERC-777 and Tokens with Unorthodox Logic

This contract is designed to work with compliant ERC-20 tokens. We did not fully investigate the potential effects of all types of non-compliant ERC-20 tokens, but we do not recommend using them with this system as they could introduce unexpected behaviors or security risks.

Recommendations

1. **Document Token Compatibility Requirements**:
    - Clearly document that these contracts are designed to work only with compliant standard ERC-20 tokens
    - Explicitly warn that using non-standard tokens may lead to unexpected behavior or severe security implications

2. **Token Compatibility Improvements**:
    - Adapt to STAKE_TOKEN's actual decimals rather than hardcoding values

- Use larger integer types where appropriate to prevent overflow with higher-decimal tokens
- Implement pre/post balance checks to detect fee-on-transfer tokens
- Add clear documentation that only standard ERC20 tokens with 18 decimals should be used

3. **Testing and Verification**:

   - Test against various non-standard token implementations to identify compatibility issues
   - Whitelist compatible tokens or token implementations to prevent unexpected behavior

Code Quality Recommendations

Parameter naming inconsistency can lead to confusion in stake and unstake functions

In the `GovLst` contract, the `_amount` parameter has different semantic meanings between the `stakeForShares()` and `sharesForStake()` functions, which could lead to confusion and potential implementation errors.

In `sharesForStake()`, the `_amount` parameter represents the amount of stake tokens (the input token to be staked), while in `stakeForShares()`, the `_amount` parameter represents the amount of shares to be withdrawn (the output token received after unstaking). This inconsistency in parameter semantics while using the same parameter name makes the code confusing and increases the risk of errors during development, maintenance, or when integrating with other parts of the system.

Recommendation

Rename parameters to clearly indicate what they represent in each function:

```
-  function stakeForShares(uint256 _amount) public view virtual returns (uint256
+ function stakeForShares(uint256 _shares) public view virtual returns (uint256)
     Totals memory _totals = totals;
-    return _calcStakeForShares(_amount, _totals);
+   return _calcStakeForShares(_shares, _totals);
  }
....
-  function _calcStakeForShares(uint256 _amount, Totals memory _totals) internal
+  function _calcStakeForShares(uint256 _shares, Totals memory _totals) internal
     if (_totals.shares == 0) {
-      return _amount / SHARE_SCALE_FACTOR;
+     return _shares / SHARE_SCALE_FACTOR;
     }
-    return (_amount * _totals.supply) / _totals.shares;
```

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

_calcDepositId gas optimization in _stake function

```
+ uint depositId = _calcDepositId(_holderState)
-   if (!_isSameDepositId(_calcDepositId(_holderState), DEFAULT_DEPOSIT_ID)) {
+ if (!_isSameDepositId(depositId, DEFAULT_DEPOSIT_ID)) {
      _holderState.balanceCheckpoint =
        _min(_holderState.balanceCheckpoint + uint96(_amount), uint96(_calcBalan
    }
...
- STAKER.stakeMore(_calcDepositId(_holderState), uint96(_amount));
+ STAKER.stakeMore(depositId, uint96(_amount));
```

Inconsistent nomenclature

We recommend changing `depositId` to `newDepositId` so it's congruent with GovLstOnBehalf.sol terminology.

```
abstract contract FixedGovLstOnBehalf is FixedGovLst {
  /// @notice Type hash used when encoding data for `updateDepositOnBehalf` call
- bytes32 public constant UPDATE_DEPOSIT_TYPEHASH = keccak256("UpdateDeposit(add
+ bytes32 public constant UPDATE_DEPOSIT_TYPEHASH = keccak256("UpdateDeposit(add
```

Domain Separator spelled incorrectly in tests

"Seperator" -> Separator in GovLst.t.sol and WithdrawGate.t.sol