Beanstalk BIP-39 Audit Report

Prepared by Cyfrin

Version 1.2

Lead Auditors

Giovanni Di Siena

Carlos Amarante

Assisting Auditors

Dacian

May 2, 2024

Contents

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

Beanstalk is a permissionless algorithmic stablecoin protocol built on Ethereum. The protocol uses a novel dynamic peg maintenance mechanism to have the price of 1 BEAN (the Beanstalk stablecoin) continuously cross its peg value of 1 USD without centralization or collateral requirements.

The BIP-39 upgrade specifically introduces the Seed Gauge System which is designed to dynamically adjust the Grown Stalk distribution in each Season based on deposited BDV per whitelisted LP and Bean. To achieve this, the protocol now considers four parameters when assessing the state of Beanstalk:

- **L2SR**: The USD value of the non-Bean assets trading against Bean that are whitelisted in the Silo. This excludes the locked liquidity corresponding to liquidity backing unripe assets.
- **Bean Price (`deltaB`)**: The deficit/surplus of Bean in its liquidity pools, according to the Curve and Well TWAP oracles. This determines whether the price is above, at, or below peg.
- **Change in Soil Demand**: This is a measure of the ability of Beanstalk to attract a sufficient number of creditors.
- **Pod Rate**: $\frac{\text{minted pods} - \text{paid pods}}{\text{bean supply}}$.

As a result, the protocol now considers 144 cases when updating the Temperature (interest paid for minting Pods) and Gauge Point distribution (which determines how much Grown Stalk accrues to each asset) in each season. A visual summary of the changes BIP-39 makes over the `gm` function can be found here.

Other notable changes introduced in the diff between the two commits referenced in the audit scope include:

- Support for Unripe lambda to lambda Conversions in the Silo has been added by introduction of a new `UNRIPE_TO_RIPE` conversion type. Currently, Farmers can only Chop Unripe assets by forfeiting all their associated Grown Stalk. This new conversion type simply Chops Unripe assets deposited in the Silo and then deposits the corresponding underlying asset, accounting for its penalty and attached Grown Stalk.
- Beanstalk currently changes Temperature on an absolute scale, and does not support the ability to change it on a relative scale. Support for changing Temperature on a relative scale has now been added.

- Currently, the total Deposited BDV stored on-chain does not account for Deposits that have not been migrated to Silo V3. The remaining un-migrated BDV for each token at the BIP-38 migration block has been calculated, incrementing `totalDepositedBDV` by the difference of the un-migrated BDV and the migrated BDV between BIP excecutions. Corresponding modifications have been made to `LibLegacyTokenSilo::_mowAndMigrate` to avoid double-counting this BDV when subsequent migrations to Silo V3 are made by Farmers with legacy deposits.

5 Audit Scope

Cyfrin conducted an audit of Beanstalk based on the code present in the repository commit hash dfb418d, specifically the diff between this commit hash and 7606673 which largely pertains to the BIP-39 upgrade which introduces the Seed Gauge System for dynamic incentivization of deposits for different whitelisted tokens based on the ratio of deposited BDVs. Other changes introduced in the diff between these two commits, since the previous audit performed by Cyfrin, are also considered in the scope of this audit.

The scope also includes a script for the calculation of the BDV of unmigrated deposits at a given block, present in commit 120ae27, and code changes following EBIP remediations merged into original `seed-gauge` branch along with other BIP-39 remediations at commit hash 5142b8f.

6 Executive Summary

Over the course of 26 days, the Cyfrin team conducted an audit on the Beanstalk BIP-39 smart contracts provided by Beanstalk Farms. In this period, a total of 27 issues were found.

This review of the BIP-39 upgrade to Beanstalk yielded a number of bugs that would have affected the core business logic and protocol accounting had they not been identified. Two high-severity findings related to the upgrade process and `InitBipSeedGauge` initialization contract logic have been raised and should be addressed to mitigate errors in Stalk accounting. Two of the medium-severity findings raised related to oversights in the intialization of the Seed Gauge System while another pertains to the incorrect calculation of the locked underlying amount which subsequently affects calculation of the Weather case in the Sunrise mechanism. A number of low-severity issues have also been identified where it is possible for protocol invariants to be broken under certain circumstances. While these do not appear to pose an immediate threat to the functioning of the protocol, it is very often these edge cases, where the system fails to enforce some property that is intended, that give rise to bugs that can be chained together for critical-severity attacks.

It is also important to note that the total deposited BDV migration calculation was provided on request after the formal audit end date and as such has not been subject to a thorough review, although some issues were still identified. Before BIP-38, the migrated BDV for Unripe LP was calculated based on the underlying `BEAN:3CRV`, but after this is `BEAN:ETH` Well liquidity. Given that before BIP-38 there remained a large proportion of unmigrated BDV, it is important to consider how to correctly calculate the new migrated BDV values, how the change of BDV corresponding to the underlying asset in `BDVFacet::unripeLPToBDV` has been considered, and how these affect the outstanding unmigrated BDV values used in BIP initialization.

The main points to take away from this engagement are as follows:

- Care should be taken by the Beanstalk Farms team to always add all modified facets and all facets whose library dependencies have been modified to the relevant upgrade script. It is strongly recommended to develop an upgrade simulation test suite to thoroughly test future upgrades so as to minimize the chance of other similar errors in the upgrade process going unnoticed.
- There is currently insufficient testing of the new Seed Gauge System and BIP-39 upgrade process itself. Extensive unit tests should be written to cover as many protocol states as possible, validating core properties and behavior before, during and after the upgrade has executed.
- The breaking of protocol invariants is a serious issue that could lead to other higher-severity vulnerabilities that have not yet been identified but may well exist if core properties do not hold. We urge the Beanstalk

Farms team to consider fixing any instance where this is the case as soon as possible, in this case prior to or as part of the BIP-39 upgrade.

Summary

Project Name	Beanstalk BIP-39
Repository	Beanstalk
Commit	dfb418d185cd...
Audit Timeline	Oct 16th - Nov 20th
Methods	Manual Review, Unit Testing, Differential Testing

Issues Found

Critical Risk	0
High Risk	2
Medium Risk	4
Low Risk	6
Informational	9
Gas Optimizations	6
Total Issues	27

Summary of Findings

[H-1] Failure to add modified facets and facets with modified dependencies to `bips::bipSeedGauge` breaks the protocol	Open
[H-2] The previous milestone stem should be scaled for use with the new gauge point system which uses untruncated values moving forward	Open
[M-1] Incorrect handling of decimals in `LibLockedUnderlying::getPercentLockedUnderlying` results in an incorrect value being returned, affecting the temperature and Bean to maxLP gaugePoint per BDV ratio updates in each subsequent call to `SeasonFacet::gm` when `unripe asset supply < 10M`	Open
[M-2] Gauge point updates should be made considering the time-weighted average deposited LP BDV rather than instantaneous at the time of Sunrise	Open
[M-3] Gauge point constants in `InitBipSeedGauge` should be scaled by the ratio of deposited BDV	Open
[M-4] Incorrect calculation of unmigrated BDVs for use in `InitBipSeedGauge::init`	Open
[L-1] Missing validation in `LibWhitelist::verifyTokenInLibWhitelistedTokens`	Open

[L-2] Potentially unsafe cast from negative `int96` values	Open
[L-3] Both reserves should be checked in `LibWell::getWellPriceFromTwaReserves`	Open
[L-4] Potential DoS of `SeasonFacet::gm` due to division by zero in `LibGauge::updateGaugePoints`	Open
[L-5] Small unripe token withdrawals don't decrease BDV and Stalk	Open
[L-6] Stalk rewards don't get burned for large partial withdrawals due to unsafe downcast	Open
[I-1] Incorrect storage slot annotation in `Storage::SiloSettings`	Open
[I-2] `LibLockedUnderlying` regression might not be representative of the expected behaviour	Open
[I-3] Outdated Seed Gauge System documentation in PR and inline comments	Open
[I-4] Duplicated code between `LibChop` and `LibUnripe`	Open
[I-5] Outdated reference to urBEAN3CRV Convert	Open
[I-6] Miscellaneous NatSpec and inline comment errors	Open
[I-7] Time-weighted average reserves should be read from the Beanstalk Pump in `LibWell` using a `try/catch` block	Open
[I-8] Use of average grown stalk per BDV is not correctly documented	Open
[I-9] Consolidate unnecessary code duplication in `ConvertFacet::_withdrawTokens`	Open
[G-1] Break out of `LibWhitelist` loops early once the condition is met	Open
[G-2] `LibBytes::packAddressAndStem` calculated twice with the same parameters	Open
[G-3] `LibTokenSilo::stemTipForToken` calculated multiple times with same parameter	Open
[G-4] `SiloFacet::transferDeposits` should only call `LibSiloPermit::_spendDepositAllowance` once	Open
[G-5] Cache updated remaining amount to prevent extra storage read	Open
[G-6] Cache recapitalized amount to prevent extra storage read	Open

7 Findings

7.1 High Risk

7.1.1 Failure to add modified facets and facets with modified dependencies to `bips::bipSeedGauge` breaks the protocol

Description: At the time of a Diamond Proxy upgrade, modified facets are cut by their inclusion in the relevant function within `bips.js`. are Currently, the `bipSeedGauge` function appears to be missing `FieldFacet`, `BDVFacet`, `ConvertFacet`, and `WhitelistFacet` which have all been modified since the previous upgrade. Moreover, the addition of facets with modifications to their libraries has not been taken into account, resulting in multiple issues that break the protocol.

Impact: At first glance, given that it appears none of these facets or the libraries they use contain significant modifications to the core business logic of Beanstalk, the impact could be considered low. However, given there have been significant alterations to other libraries utilized by multiple facets, this is not the case. One of the more severe issues involves the issuance of significantly increased amounts of Stalk than intended which therefore breaks protocol accounting.

Proof of Concept: A list of all modified facets can be obtained by running the following command:

```
git diff --stat 7606673..dfb418d -- ".sol" ":\!protocol/test/" ":\!protocol/contracts/mocks/*" | grep
↪  "Facet.sol"
```

Output:

```
protocol/contracts/beanstalk/barn/UnripeFacet.sol  | 360 +++++++++------
protocol/contracts/beanstalk/field/FieldFacet.sol  |   4 +-
protocol/contracts/beanstalk/silo/BDVFacet.sol     |  12 +-
protocol/contracts/beanstalk/silo/ConvertFacet.sol |   8 +-
.../contracts/beanstalk/silo/WhitelistFacet.sol    |  79 +++-
.../contracts/beanstalk/sun/GaugePointFacet.sol    |  39 ++
.../beanstalk/sun/SeasonFacet/SeasonFacet.sol      | 120 +++--
.../sun/SeasonFacet/SeasonGettersFacet.sol         | 248 ++++++++++
```

A list of all modified libraries can be obtained by running the following command:

```
git diff --stat 7606673..dfb418d -- "*.sol" ":\!protocol/test/*" ":\!protocol/contracts/mocks/*" | grep
↪  "Lib.*\.sol"
```

Output:

```
.../contracts/libraries/Convert/LibChopConvert.sol |  60 +++
.../contracts/libraries/Convert/LibConvert.sol     |  42 +-
.../contracts/libraries/Convert/LibConvertData.sol |   3 +-
.../libraries/Convert/LibUnripeConvert.sol         |  18 +-
.../contracts/libraries/Convert/LibWellConvert.sol |   3 +-
.../contracts/libraries/Curve/LibBeanMetaCurve.sol |  15 +
.../contracts/libraries/Curve/LibMetaCurve.sol     |  61 ++-
protocol/contracts/libraries/LibCases.sol          | 161 +++++++
protocol/contracts/libraries/LibChop.sol           |  65 +++
protocol/contracts/libraries/LibEvaluate.sol       | 297 ++++++++++++
protocol/contracts/libraries/LibFertilizer.sol     |   2 +
protocol/contracts/libraries/LibGauge.sol          | 330 ++++++++++++
protocol/contracts/libraries/LibIncentive.sol      |  20 +-
.../contracts/libraries/LibLockedUnderlying.sol    | 509 ++++++++++++++++++++
protocol/contracts/libraries/LibUnripe.sol         | 180 ++++++--
.../libraries/Minting/LibCurveMinting.sol          |  26 +-
.../contracts/libraries/Minting/LibWellMinting.sol |  30 +-
.../libraries/Oracle/LibBeanEthWellOracle.sol      |  55 ---
.../contracts/libraries/Oracle/LibEthUsdOracle.sol |   3 +-
.../contracts/libraries/Oracle/LibUsdOracle.sol    |  18 +-
.../libraries/Silo/LibLegacyTokenSilo.sol          |   4 -
protocol/contracts/libraries/Silo/LibTokenSilo.sol |  18 +-
protocol/contracts/libraries/Silo/LibWhitelist.sol | 181 ++++++--
.../libraries/Silo/LibWhitelistedTokens.sol        |  47 ++
protocol/contracts/libraries/Well/LibWell.sol      | 217 ++++++++-
```

The following coded proof of concept has been written to demonstrate the broken Stalk accounting:

```
const { expect } = require('chai');
const { takeSnapshot, revertToSnapshot } = require("../utils/snapshot.js");
const { BEAN, BEAN_3_CURVE, UNRIPE_BEAN, UNRIPE_LP, WETH, BEAN_ETH_WELL, PUBLIUS,
↪  ETH_USD_CHAINLINK_AGGREGATOR } = require('../utils/constants.js');
const { bipSeedGauge } = require('../../scripts/bips.js');
const { getBeanstalk } = require('../../utils/contracts.js');
const { impersonateBeanstalkOwner, impersonateSigner } = require('../../utils/signer.js');
const { ethers } = require('hardhat');

const { impersonateBean, impersonateEthUsdChainlinkAggregator} =
↪  require('../../scripts/impersonate.js');
let silo, siloExit, bean

let grownStalkBeforeUpgrade, grownStalkAfterUpgrade
let snapshotId
let whitelistedTokenSnapshotBeforeUpgrade, whitelistedTokenSnapshotAfterUpgrade

const whitelistedTokens = [BEAN, BEAN_3_CURVE, UNRIPE_BEAN, UNRIPE_LP, BEAN_ETH_WELL]

const whitelistedTokensNames = ["BEAN", "BEAN:3CRV CURVE LP", "urBEAN", "urBEAN:WETH", "BEAN:WETH WELLS
↪  LP"]
const beanHolderAddress = "0xA9Ce5196181c0e1Eb196029FF27d61A45a0C0B2c"
let beanHolder

/**
 * Async function
 * @returns tokenDataSnapshot: Mapping from token address to (name,stemTip)
 */
const getTokenDataSnapshot = async()=>{
  tokenDataSnapshot = new Map()

  for(token of whitelistedTokens){
    tokenDataSnapshot.set(token,{
      name: whitelistedTokensNames[whitelistedTokens.indexOf(token)],
      stemTip: await silo.stemTipForToken(token)
    })
  }
  return tokenDataSnapshot
}



const forkMainnet = async()=>{
  try {
    await network.provider.request({
      method: "hardhat_reset",
      params: [
        {
          forking: {
            jsonRpcUrl: process.env.FORKING_RPC,
            blockNumber: 18619555-1 //a random semi-recent block close to Grown Stalk Per Bdv
            ↪  pre-deployment
          },
        },
      ],
    });
  } catch(error) {
    console.log('forking error in seed Gauge');
    console.log(error);
    return
  }
}

const initializateContractsPointers = async(beanstalkAddress)=>{
  tokenSilo = await ethers.getContractAt('TokenSilo', beanstalkAddress);
  seasonFacet = await ethers.getContractAt('ISeasonFacet', beanstalkAddress);
```

As is shown by the reverting expectations, failure to add `SiloFacet` to the upgrade breaks the milestone stem update and the grown stalk accounting.

If the solution for the issue relating to the previous milestone stem being scaled for use with the new gauge point system (which uses untruncated values moving forward) is implemented without updating the `SiloFacet`, then the previous `LibTokenSilo::stemTipForToken` implementation is used. This allows deposits performed before the upgrade to receive significantly more grown stalk than intended.

Recommended Mitigation: Be sure to always add all modified facets and all facets whose library dependencies have been modified to the upgrade script. It is highly recommended to develop an upgrade simulation test suite to catch other similar errors in the upgrade process in the future.

7.1.2 The previous milestone stem should be scaled for use with the new gauge point system which uses untruncated values moving forward

Description: Within the Beanstalk Silo, the milestone stem for a given token is the cumulative amount of grown stalk per BDV for this token at the last `stalkEarnedPerSeason` update. Previously, the milestone stem was stored in its truncated representation; however, the seed gauge system now stores the value in its untruncated form due to the new granularity of grown stalk and the frequency with which these values are updated.

At the time of upgrade, the previous (truncated) milestone stem for each token should be scaled for use with the gauge point system by multiplying up by a factor of `1e6`. Otherwise, there will be a mismatch in decimals when calculating the stem tip.

```
_stemTipForToken = s.ss[token].milestoneStem +
    int96(s.ss[token].stalkEarnedPerSeason).mul(
        int96(s.season.current).sub(int96(s.ss[token].milestoneSeason))
    );
```

Impact: The mixing of decimals between the old milestone stem (truncated) and the new milestone stem (untruncated, after the first `gm` call following the BIP-39 upgrade) breaks the existing grown stalk accounting, resulting in a loss of grown stalk for depositors.

Proof of Concept: The previous implementation returns the cumulative stalk per BDV with 4 decimals:

```
    function stemTipForToken(address token)
        internal
        view
        returns (int96 _stemTipForToken)
    {
        AppStorage storage s = LibAppStorage.diamondStorage();

        // SafeCast unnecessary because all casted variables are types smaller that int96.
        _stemTipForToken = s.ss[token].milestoneStem +
        int96(s.ss[token].stalkEarnedPerSeason).mul(
            int96(s.season.current).sub(int96(s.ss[token].milestoneSeason))
        ).div(1e6); //round here
    }
```

Which can be mathematically abstracted to: $$StemTip(token) = getMilestonStem(token) + (current \ season - getMilestonStemSeason(token)) \times \frac{stalkEarnedPerSeason(token)}{10^{6}}$$

This division by 10^{6} happens because the stem tip previously had just 4 decimals. This division allows backward compatibility by not considering the final 6 decimals. Therefore, the stem tip **MUST ALWAYS** have 4 decimals.

The milestone stem is now updated in each `gm` call so long as all LP price oracles pass their respective checks. Notably, the milestone stem is now stored with 10 decimals (untruncated), hence why the second term of the abstraction has omitted the `10^{6}` division in `LibTokenSilo::stemTipForTokenUntruncated`.

However, if the existing milestone stem is not escalated by 10^{6} then the addition performed during the upgrade and in subsequent `gm` calls makes no sense. This is mandatory to be handled within the upgrade otherwise every part of the protocol which calls `LibTokenSilo.stemTipForToken` will receive an incorrect value, except for BEAN:ETH Well LP (given it was created after the Silo v3 upgrade).

Some instances where this function is used include:

- `EnrootFacet::enrootDeposit`
- `EnrootFacet::enrootDeposits`
- `MetaFacet::uri`
- `ConvertFacet::_withdrawTokens`
- `LibSilo::__mow`
- `LibSilo::_removeDepositFromAccount`
- `LibSilo::_removeDepositsFromAccount`
- `Silo::_plant`
- `TokenSilo::_deposit`
- `TokenSilo::_transferDeposits`
- `LibLegacyTokenSilo::_mowAndMigrate`
- `LibTokenSilo::_mowAndMigrate`

As can be observed, critical parts of the protocol are compromised, leading to further cascading issues.

Recommended Mitigation: Scale up the existing milestone stem for each token:

```
for (uint i = 0; i < siloTokens.length; i++) {
+   s.ss[siloTokens[i]].milestoneStem = int96(s.ss[siloTokens[i]].milestoneStem.mul(1e6));
```

7.2 Medium Risk

7.2.1 Incorrect handling of decimals in `LibLockedUnderlying::getPercentLockedUnderlying` results in an incorrect value being returned, affecting the temperature and Bean to maxLP gaugePoint per BDV ratio updates in each subsequent call to `SeasonFacet::gm` when `unripe asset supply < 10M`

Description: Due to the Barn Raise and the associated Beans underlying Unripe assets, the number of tradable Beans does not equal the total Bean supply. Within the calculation of L2SR, the term "locked liquidity" refers to the portion of liquidity in the BEAN:ETH WELL that cannot be retrieved through chopping until the corresponding Fertilizer is paid.

The exchange ratio for the corresponding underlying asset can be summarized in the following formula:

$$\frac{Paid Fertilizer}{Minted Fertilizer} \times \frac{totalUnderlying(urAsset)}{supply(urAsset)}$$

The second factor indicates the amount of the underlying asset backing each unripe asset, while the first indicates the distribution of the underlying asset based on the ratio of Fertilizer that is already paid.

When a user chops an unripe asset, it is burned in exchange for a penalized amount of the underlying asset. The remaining underlying asset is now shared among the remaining unripe asset holders, meaning that if another user tries to chop the same amount of unripe asset at a given recapitalization rate, they will receive a greater amount of underlying asset.

For instance, assume that:

- 50% of the minted Fertilizer is paid
- A current supply of 70M
- An underlying amount of 22M

If Alice chops 1M unripe tokens: $$1,000,000 \times 0.50 \times \frac{22,000,000}{70,000,000} =$$ $$1,000,000 \times 0.50 \times 0.31428 =$$ $$1,000,000 \times 0.50 \times 0.31428 =$$ $$1,000,000 \times 0.15714285 = $$ $$157,142.85$$

If Bob then chops the same amount of tokens: $$1,000,000 \times 0.50 \times \frac{22,000,000-157,142.85}{70,000,000 - 1,000,000} =$$ $$1,000,000 \times 0.50 \times \frac{21,842,857.15}{69,000,000} =$$ $$1,000,000 \times 0.50 \times \frac{21,842,857.15}{69,000,000} =$$ $$1,000,000 \times 0.50 \times 0.3165 =$$ $$158,281.57$$

Given that the assumption of chopping the total unripe asset supply in one step is highly unlikely, the Beanstalk Farms team decided to perform an off-chain regression based on the average unripe asset per unripe asset holder. This yields an approximation for the percentage locked underlying token per asset based on the current unripe asset supply. An on-chain look-up table is used to retrieve the values of this regression; however, the issue with its implementation lies in its failure to account for unripe token decimals when compared with the inline conditional supply constants `1_000_000`, `5_000_000`, and `10_000_000` as the intervals on which the iterative simulation was performed. Given these constants are not a fixed-point representation of the numbers they are intended to represent, comparison with the 6-decimal supply will be incorrect.

Impact: Given that unripe assets have 6 decimals, `LibLockedUnderlying::getPercentLockedUnderlying` will tend to execute this conditional branch, producing an incorrect calculation of locked underlying whenever the supply of the unripe asset is below 10M.

In the given scenario, this error would cascade into an incorrect calculation of L2SR, affecting how the temperature and Bean to maxLP gaugePoint per BDV ratio should be updated in the call to `Weather::calcCaseIdandUpdate` within `SeasonFacet::gm`.

Proof of Concept: A differential test (see Appendix A) was written to demonstrate this issue based on CSV provided by the Beanstalk Farms team. Modifications to the CSV include:

- Adding headers: recapPercentage, urSupply, lockedPercentage
- Generate a CSV without whitespaces
- Round the first column to 3 decimals

- For the third column, delete `e18` and round values to 18 decimals

Recommended Mitigation: Scale each inline constant that is compared against the unripe supply by 6 decimals.

For similar cases in the future, differential testing between the expected and actual outputs is effective in catching bugs of this type which rely on pre-computed off-chain values.

7.2.2 Gauge point updates should be made considering the time-weighted average deposited LP BDV rather than instantaneous at the time of Sunrise

Description: Prior to the introduction of the Seed Gauge System, the Grown Stalk per BDV for whitelisted assets was static and could only be changed via governance. The Seed Gauge System now allows Beanstalk to target an amount of Grown Stalk per BDV that should be issued per Season, with Gauge Points being introduced to determine how the Grown Stalk issued that Season should be distributed between whitelisted LP tokens.

Gauge Points are updated every Season, when `LibGauge::stepGauge` is called within `SeasonFacet::gm`. This Gauge Point update is currently performed by considering the instantaneous total deposited LP BDV at the time of the `gm` call. However, this value can be subject to manipulation so the Seed Gauge System should instead use a time-weighted average deposited LP BDV over the previous Season duration.

Impact: Given the Gauge Points for a given whitelisted LP can only increase/decrease by one point per Season, and the Bean to max LP GP per BDV ratio is capped at 100%, the incentive to perform this attack is relatively low. However, a large deposit immediately before the Sunrise call, and withdrawal immediately after, could nonetheless result in manipulation meaning the Seed Gauge system does not work as intended.

Recommended Mitigation: Consider calculating time-weighted average deposited LP BDVs over the previous Season duration rather than using an instantaneous value. The BDV to include in the calculation at each block should be the one at the end of the previous block to avoid in-block manipulation. These values should be stored and the update should be triggered whenever a function is called which modifies the total deposited BDV in any way.

7.2.3 Gauge point constants in `InitBipSeedGauge` should be scaled by the ratio of deposited BDV

Description: The current initial Gauge Point (GP) distribution is based solely on the grown stalk per season per BDV for each LP, whereas it should be determined by considering the deposited BDV per LP.

Considering the following math which underlies the behavior of the gauge system: $$depositedBDVRatio(LP) = \frac{silo.totalDepositedBDV(LP)}{\sum_{wlpt}^{wlpt \in Whitelisted \ LP \ Tokens} silo.totalDepositedBDV} $$ $GP_{s}(LP) =$

1. $depositedBDVRatio(LP) > LP.optimalDepositedBDVRatio \land GP_{s-1}(LP) \leq 1gp \Rightarrow GP_{s}(LP) = 0$
2. $depositedBDVRatio(LP) > LP.optimalDepositedBDVRatio \land GP_{s-1}(LP) > 1gp \Rightarrow GP_{s}(LP) = GP_{s-1}(LP)- 1gp$
3. $depositedBDVRatio(LP) \leq LP.optimalDepositedBDVRatio \Rightarrow GP_{s}(LP) = GP_{s-1}(LP) + 1gp$

It can be seen that the formula relies on the previous $GP_{s-1}(LP)$, where s indicates the current season number and deposited BDV ratio. Moreover, it is evident that the intention of this mechanism is to incentivize the Beanstalk protocol to have a pre-defined optimal deposited BDV ratio for each LP. Consequently, the initial assignment of GP should consider this intention.

Impact: An incorrect initial GP distribution can result in unintended initial behavior, which can take a significant amount of time to rectify given that gauge points can only increase/decrease by one point per season as defined in `GaugePointFacet::defaultGaugePointFunction`.

Proof of Concept:

```
// InitBipSeedGauge.sol
uint128 beanEthGp = uint128(s.ss[C.BEAN_ETH_WELL].stalkEarnedPerSeason) * 500 * 1e12;
uint128 bean3crvGp = uint128(s.ss[C.CURVE_BEAN_METAPOOL].stalkEarnedPerSeason) * 500 * 1e12
```

As observed, the initial GP assignment is determined by the stalk earned per season before BIP-39, with the following values:

- BEAN:3CRV Curve LP: `3.25e6`
- BEAN:ETH Well LP: `4.5e6`

These values are not correlated with the total BDV deposited per LP. Consequently, the initial assignment of GP is made with incorrect values.

Recommended Mitigation: Considering that one gauge point is equal to 1e18, the following modification should be made:

```
//  InitBipSeedGauge.sol
+   // BDV has 6 decimals
+   uint256 beanEthBDV = s.siloBalances[C.BEAN_ETH_WELL].depositedBdv
+   uint256 bean3crvBDV = s.siloBalances[C.CURVE_BEAN_METAPOOL].depositedBdv
+   uint256 lpTotalBDV = beanEthBDV + bean3crvGp
-   uint128 beanEthGp = uint128(s.ss[C.BEAN_ETH_WELL].stalkEarnedPerSeason) * 500 * 1e12;
-   uint128 bean3crvGp = uint128(beanEthBDV) * 500 * 1e12
+   // Assume 1 BDV = 1GP for initialization
+   uint128 beanEthGp = uint128(beanEthBDV * 10e6).div(lpTotalBDV) * 1e12;
+   uint128 bean3crvGp = uint128(bean3crvBDV * 10e6).div(lpTotalBDV) * 1e12
```

7.2.4 Incorrect calculation of unmigrated BDVs for use in `InitBipSeedGauge::init`

Description: The current values for the constants in `InitBipSeedGauge::init` are an estimation and not finalized. To correctly calculate the BDV, the Beanstalk Farms team simulates migrating all the remaining unmigrated deposits at the block in which BIP-38 was executed such that the change of BDV corresponding to the underlying asset in `BDVFacet::unripeLPToBDV` is considered and subject to the slippage incurred at the time of liquidity migration. The `deposits.json` file contains a list of outstanding deposits at the Silo V3 deployment block `17671557`, so the script considers all `removeDeposit` events after this point as deposits to be removed from the unmigrated BDV. By filtering from the Enroot fix deployment block `17251905`, if an account has removed its deposit after the Enroot fix but before Silo V3 was deployed, this would improperly assume the deposits have been migrated when they haven't. Additionally, given the script is forking mainnet at the BIP-38 execution block `18392690`, it is not correct to use `18480579` as the end block for event filtering.

The case has also been considered that, given the state changes will already have been applied, and assuming the migration transaction isn't top/bottom of block, it might be desirable to fork/filter up to the block before BIP-38 execution and check whether any migrations occurred before/after the migration transaction that need to be considered manually. After further inspection of the block in which the BIP-38 upgrade took place, it appears this is not necessary as no events were emitted.

An additional discrepancy in the unmigrated Bean BDV value was identified by the Beanstalk Farms team. After Silo V3, the implementation of `Sun::rewardToSilo` increments the BDV by the amount of Bean issued to the Silo, but all previously earned Beans are not considered. Therefore, the value returned by `SiloExit::totalEarnedBeans` at the time of Silo V3 deployment should be added to the total.

Impact: The calculated unmigrated BDVs are incorrect, as shown below. The current implementation returns values that are smaller than they should be, meaning the total deposited BDV will fail to consider some deposits and be lower than intended.

Output of the current implementation:

```
unmigrated:  {
  '0x1BEA0050E63e05FBb5D8BA2f10cf5800B6224449': BigNumber { value: "3209210313166" },
  '0x1BEA3CcD22F4EBd3d37d731BA31Eeca95713716D': BigNumber { value: "6680992571569" },
  '0xBEA0000029AD1c77D3d5D23Ba2D8893dB9d1Efab': BigNumber { value: "304630107407" },
  '0xc9C32cd16Bf7eFB85Ff14e0c8603cc90F6F2eE49': BigNumber { value: "26212521946" }
}
```

Corrected output:

```
unmigrated:  {
  '0x1BEA0050E63e05FBb5D8BA2f10cf5800B6224449': BigNumber { value: "3736196158417" },
  '0x1BEA3CcD22F4EBd3d37d731BA31Eeca95713716D': BigNumber { value: "7119564766493" },
  '0xBEA0000029AD1c77D3d5D23Ba2D8893dB9d1Efab': BigNumber { value: "689428296238" },
  '0xc9C32cd16Bf7eFB85Ff14e0c8603cc90F6F2eE49': BigNumber { value: "26512602424" }
}
```

Recommended Mitigation: Apply the following diff:

```
// L645
- const END_BLOCK = 18480579;
+ const END_BLOCK = BLOCK_NUMBER;

// L811-812
- //get every transaction that emitted the RemoveDeposit event after block 17251905
+ //get every transaction that emitted the RemoveDeposit event after block 17671557
- let events = await queryEvents("RemoveDeposit(address,address,uint32,uint256)",
↪  removeDepositInterface, 17251905); //update this block to latest block when running actual script,
↪  in theory someone could have migrated meanwhile
+ let events = await queryEvents("RemoveDeposit(address,address,uint32,uint256)",
↪  removeDepositInterface, 17671557);
```

Retrieve the amount of Beans previously issued to the Silo:

```
cast call 0xC1E088fC1323b20BCBee9bd1B9fC9546db5624C5 "totalEarnedBeans()" --rpc-url ${FORKING_RPC}
↪  --block "17671557"
```

7.3 Low Risk

7.3.1 Missing validation in `LibWhitelist::verifyTokenInLibWhitelistedTokens`

Description: Prior to the introduction of `LibWhitelistedToken.sol`, Beanstalk did not have a way of iterating through its whitelisted tokens. To mitigate against an upgrade where a new asset is whitelisted, but `LibWhitelistedToken.sol` is not updated, `LibWhitelist::verifyTokenInLibWhitelistedTokens` verifies that the token is both in the correct array(s) and not in invalid arrays.

While `LibWhitelistedTokens::getWhitelistedWellLpTokens` is supposed to return a subset of whitelisted LP tokens, this is not guaranteed. In this case, if the token is either Bean or an Unripe Token, the first `else` block within `LibWhitelist::verifyTokenInLibWhitelistedTokens` should also check that the token is not in the whitelisted Well LP token array.

Recommended Mitigation:

```
} else {
    checkTokenNotInArray(token, LibWhitelistedTokens.getWhitelistedLpTokens());
+   checkTokenNotInArray(token, LibWhitelistedTokens.getWhitelistedWellLpTokens());
}
```

7.3.2 Potentially unsafe cast from negative `int96` values

Description: Where calculations are performed on `int96` values, for example when manipulating stems in `LibSilo::stalkReward`, `LibTokenSilo::grownStalkForDeposit`, and `LibTokenSilo::calculateGrownStalkAndStem`, Beanstalk uses the `LibSafeMathSigned96` library. Based on the invariant that the stem for a new deposit should never exceed the stem tip for a given token, casting these values to `uint256` is fine since the difference between the two stem values should never be negative. However, in the event of a bug that violates this invariant, it could be possible to have a negative `int96` value cast to a very large `uint256` value, potentially resulting in a huge amount of stalk being minted.

This issue is already sufficiently mitigated in `LibTokenSilo::grownStalkForDeposit` and it appears the instance in `LibTokenSilo::calculateGrownStalkAndStem` can never reach this state. Additional logic should similarly be added to `LibSilo::stalkReward` to ensure that the result of subtraction is positive and thus the cast to `uint256` is safe.

Impact: While it appears not currently exploitable, a bug in the calculation of the stem for a given deposit or stem tip for a given token in `LibSilo::stalkReward` could result in the erroneous minting of a large amount of stalk.

Proof of Concept: The following forge test demonstrates this issue:

```
contract TestStemsUnsafeCasting is Test {
    using LibSafeMathSigned96 for int96;

    function stalkReward(int96 startStem, int96 endStem, uint128 bdv)
        internal
        view
        returns (uint256)
    {
        int96 reward = endStem.sub(startStem).mul(int96(bdv));
        console.logInt(reward);
        console.logUint(uint128(reward));

        return uint128(reward);
    }

    function test_stalk_reward() external {
        uint256 reward = stalkReward(1200, 1000, 1337);
        console.logUint(reward);
    }
}
```

Recommended Mitigation: Add additional logic to safely perform the cast from `int96` or otherwise handle the case where the result of stem subtraction could be negative.

7.3.3 Both reserves should be checked in `LibWell::getWellPriceFromTwaReserves`

Description:

```
function getWellPriceFromTwaReserves(address well) internal view returns (uint256 price) {
    AppStorage storage s = LibAppStorage.diamondStorage();
    // s.twaReserve[well] should be set prior to this function being called.
    // 'price' is in terms of reserve0:reserve1.
    if (s.twaReserves[well].reserve0 == 0) {
        price = 0;
    } else {
        price = s.twaReserves[well].reserve0.mul(1e18).div(s.twaReserves[well].reserve1);
    }
}
```

Currently, `LibWell::getWellPriceFromTwaReserves` sets the price to zero if the time-weighted average reserves of the zeroth reserve (for Wells, Bean) is zero. Given the implementation of `LibWell::setTwaReservesForWell`, and that a Pump failure will return an empty reserves array, it does not appear possible to encounter the case where one reserve can be zero without the other except for perhaps an exploit or migration scenario. Therefore, whilst unlikely, it is best to but best to ensure both reserves are non-zero to avoid a potential division by zero `reserve1` when calculating the price as a revert here would result in DoS of `SeasonFacet::gm`.

```
function setTwaReservesForWell(address well, uint256[] memory twaReserves) internal {
    AppStorage storage s = LibAppStorage.diamondStorage();
    // if the length of twaReserves is 0, then return 0.
    // the length of twaReserves should never be 1, but
    // is added for safety.
    if (twaReserves.length < 1) {
        delete s.twaReserves[well].reserve0;
        delete s.twaReserves[well].reserve1;
    } else {
        // safeCast not needed as the reserves are uint128 in the wells.
        s.twaReserves[well].reserve0 = uint128(twaReserves[0]);
        s.twaReserves[well].reserve1 = uint128(twaReserves[1]);
    }
}
```

Additionally, to correctly implement the check identified by the comment in `LibWell::setTwaReservesForWell`, the time-weighted average reserves in storage should be reset if the array length is less-than or equal-to 1.

Recommended Mitigation:

```
// LibWell::getWellPriceFromTwaReserves`
- if (s.twaReserves[well].reserve0 == 0) {
+ if (s.twaReserves[well].reserve0 == 0 || s.twaReserves[well].reserve1 == 0) {
        price = 0;
} else {

// LibWell::setTwaReservesForWell
- if (twaReserves.length < 1) {
+ if (twaReserves.length <= 1) {
    delete s.twaReserves[well].reserve0;
    delete s.twaReserves[well].reserve1;
} else {
```

7.3.4 Potential DoS of `SeasonFacet::gm` due to division by zero in `LibGauge::updateGaugePoints`

There currently exists an edge case in `LibGauge::updateGaugePoints` where it is possible to unintentionally DoS `SeasonFacet::gm` due to a potential division by zero. If there is only one newly whitelisted LP token in the Beanstalk protocol which therefore has no deposited BDV, execution will revert, thus preventing Beanstalk from advancing to the next Season. While it is unlikely that Beanstalk will encounter this issue so long as the existing whitelisted LP tokens remain, there is a small possibility that this could be an issue in the event of some future liquidity migration and so it should be handled accordingly.

```
...
// if there is only one pool, there is no need to update the gauge points.
if (whitelistedLpTokens.length == 1) {
    // Assumes that only Wells use USD price oracles.
    if (LibWell.isWell(whitelistedLpTokens[0]) && s.usdTokenPrice[whitelistedLpTokens[0]] == 0) {
        return (maxLpGpPerBdv, lpGpData, totalGaugePoints, type(uint256).max);
    }
    uint256 gaugePoints = s.ss[whitelistedLpTokens[0]].gaugePoints;
+   if (s.siloBalances[whitelistedLpTokens[0]].depositedBdv != 0) {
        lpGpData[0].gpPerBdv = gaugePoints.mul(BDV_PRECISION).div(
            s.siloBalances[whitelistedLpTokens[0]].depositedBdv
        );
+   }
    return (
        lpGpData[0].gpPerBdv,
        lpGpData,
        gaugePoints,
        s.siloBalances[whitelistedLpTokens[0]].depositedBdv
    );
}
...
```

7.3.5 Small unripe token withdrawals don't decrease BDV and Stalk

Description: For any whitelisted token where `bdvCalc(amountDeposited) < amountDeposited`, a user can deposit that token and then withdraw in small amounts to avoid decreasing BDV and Stalk. This is achieved by exploiting a rounding down to zero precision loss in `LibTokenSilo::removeDepositFromAccount`:

```
// @audit small unripe bean withdrawals don't decrease BDV and Stalk
// due to rounding down to zero precision loss. Every token where
// `bdvCalc(amountDeposited) < amountDeposited` is vulnerable
uint256 removedBDV = amount.mul(crateBDV).div(crateAmount);
```

Impact: An attacker can withdraw deposited assets without decreasing BDV and Stalk. While the cost to perform this attack is likely more than the value an attacker would stand to gain, the potential impact should definitely be explored more closely especially considering the introduction of the Unripe Chop Convert in BIP-39 as this could have other unintended consequences in relation to this bug (given that the inflated BDV of an Unripe Token will persist once deposit is converted to its ripe counterpart, potentially allowing value to be extracted that way depending on how this BDV is used/manipulated elsewhere).

The other primary consideration for this bug is that it breaks the mechanism that Stalk is supposed to be lost when withdrawing deposited assets and keeps the `totalDepositedBdv` artificially high, violating the invariant that the `totalDepositedBdv` value for a token should be the sum of the BDV value of all the individual deposits.

Proof of Concept: Add this PoC to `SiloToken.test.js` under the section `describe("1 deposit, some", async function () {`:

```
it('audit small unripe bean withdrawals dont decrease BDV and Stalks', async function () {
    let initialUnripeBeanDeposited    = to6('10');
    let initialUnripeBeanDepositedBdv = '2355646';
    let initialTotalStalk = pruneToStalk(initialUnripeBeanDeposited).add(toStalk('0.5'));

    // verify initial state
    expect(await this.silo.getTotalDeposited(UNRIPE_BEAN)).to.eq(initialUnripeBeanDeposited);
    expect(await this.silo.getTotalDepositedBdv(UNRIPE_BEAN)).to.eq(initialUnripeBeanDepositedBdv);
    expect(await this.silo.totalStalk()).to.eq(initialTotalStalk);

    // snapshot EVM state as we want to restore it after testing the normal
    // case works as expected
    let snapshotId = await network.provider.send('evm_snapshot');

    // normal case: withdrawing total UNRIPE_BEAN correctly decreases BDV & removes stalks
    const stem = await this.silo.seasonToStem(UNRIPE_BEAN, '10');
    await this.silo.connect(user).withdrawDeposit(UNRIPE_BEAN, stem, initialUnripeBeanDeposited,
    ↪  EXTERNAL);

    // verify UNRIPE_BEAN totalDeposited == 0
    expect(await this.silo.getTotalDeposited(UNRIPE_BEAN)).to.eq('0');
    // verify UNRIPE_BEAN totalDepositedBDV == 0
    expect(await this.silo.getTotalDepositedBdv(UNRIPE_BEAN)).to.eq('0');
    // verify silo.totalStalk() == 0
    expect(await this.silo.totalStalk()).to.eq('0');

    // restore EVM state to snapshot prior to testing normal case
    await network.provider.send("evm_revert", [snapshotId]);

    // re-verify initial state
    expect(await this.silo.getTotalDeposited(UNRIPE_BEAN)).to.eq(initialUnripeBeanDeposited);
    expect(await this.silo.getTotalDepositedBdv(UNRIPE_BEAN)).to.eq(initialUnripeBeanDepositedBdv);
    expect(await this.silo.totalStalk()).to.eq(initialTotalStalk);

    // attacker case: withdrawing small amounts of UNRIPE_BEAN doesn't decrease
    // BDV and doesn't remove stalks. This lets an attacker withdraw their deposits
    // without losing Stalks & breaks the invariant that the totalDepositedBDV should
    // equal the sum of the BDV of all individual deposits
    let smallWithdrawAmount = '4';
    await this.silo.connect(user).withdrawDeposit(UNRIPE_BEAN, stem, smallWithdrawAmount, EXTERNAL);

    // verify UNRIPE_BEAN totalDeposited has been correctly decreased
    expect(await this.silo.getTotalDeposited(UNRIPE_BEAN)).to.eq(initialUnripeBeanDeposited.sub(smallWi
    ↪  thdrawAmount));
    // verify UNRIPE_BEAN totalDepositedBDV remains unchanged!
    expect(await this.silo.getTotalDepositedBdv(UNRIPE_BEAN)).to.eq(initialUnripeBeanDepositedBdv);
    // verify silo.totalStalk() remains unchanged!
    expect(await this.silo.totalStalk()).to.eq(initialTotalStalk);
});
```

Run with: `npx hardhat test --grep "audit small unripe bean withdrawals dont decrease BDV and Stalks"`.

Additional Mainnet fork tests have been written to demonstrate the presence of this bug in the current and post-BIP-39 deployments of Beanstalk (see Appendix B).

Recommended Mitigation: `LibTokenSilo::removeDepositFromAccount` should revert if `removedBDV == 0`. A similar check already exists in `LibTokenSilo::depositWithBDV` but is missing in `removeDepositFromAccount()` when calculating `removedBDV` for partial withdrawals.

The breaking of protocol invariants could lead to other serious issues that have not yet been identified but may well exist if core properties do not hold. We would urge the team to consider fixing this bug as soon as possible, prior to or as part of the BIP-39 upgrade.

7.3.6 Stalk rewards don't get burned for large partial withdrawals due to unsafe downcast

Description: When calling `SiloFacet::withdrawDeposit`, it is possible that Stalk rewards are not burned for large partial withdrawals as `LibSilo::stalkReward` will return 0 due to an unsafe downcast of `removedBDV` from `uint128 -> int96`.

Impact: Stalk rewards don't get burned for large partial withdrawals.

Proof of Concept: Add to `SiloToken.test.js` under the section `describe("deposit", function () {`:

```
describe("audit withdrawing deposited asset for large BDV value", function () {
  // values found via fuzz testing
  let beanDeposit       = "79228162514264337593543950337";
  let problemRemovedBdv = "79228162514264337593543950336";

  beforeEach(async function () {
    await this.season.teleportSunrise(10);
    this.season.deployStemsUpgrade();

    await this.siloToken.connect(user).approve(this.silo.address, beanDeposit);
    await this.siloToken.mint(userAddress, beanDeposit);
    await this.silo.connect(user).deposit(this.siloToken.address, beanDeposit, EXTERNAL);
  });

  it("audit stalk rewards not burned when withdrawing deposited asset for large BDV value", async
  ↪  function () {
    let initialTotalStalk = beanDeposit + "0000";

    // verify initial state
    expect(await this.silo.getTotalDeposited(this.siloToken.address)).to.eq(beanDeposit);
    // siloToken has 1:1 BDV calc
    expect(await this.silo.getTotalDepositedBdv(this.siloToken.address)).to.eq(beanDeposit);
    expect(await this.silo.totalStalk()).to.eq(initialTotalStalk);

    // fast forward to build up some stalk rewards
    await this.season.teleportSunrise(20);

    // snapshot EVM state as we want to restore it after testing the normal
    // case works as expected
    let snapshotId = await network.provider.send("evm_snapshot");

    // normal case: withdraw the entire deposited amount
    const stem = await this.silo.seasonToStem(this.siloToken.address, "10");
    await this.silo.connect(user).withdrawDeposit(this.siloToken.address, stem, beanDeposit,
    ↪  EXTERNAL);

    // verify token.totalDeposited == 0
    expect(await this.silo.getTotalDeposited(this.siloToken.address)).to.eq("0");
    // verify token.totalDepositedBDV == 0
    expect(await this.silo.getTotalDepositedBdv(this.siloToken.address)).to.eq("0");
    // verify totalStalk == 0; both the initial stalk & stalk rewards were burned
    expect(await this.silo.totalStalk()).to.eq("0");

    // restore EVM state to snapshot prior to testing normal case
    await network.provider.send("evm_revert", [snapshotId]);

    // re-verify initial state
    expect(await this.silo.getTotalDeposited(this.siloToken.address)).to.eq(beanDeposit);
    // siloToken has 1:1 BDV calc
    expect(await this.silo.getTotalDepositedBdv(this.siloToken.address)).to.eq(beanDeposit);
    expect(await this.silo.totalStalk()).to.eq(initialTotalStalk);

    // problem case: partial withdraw a precise amount causing
    // by LibTokenSilo::removeDepositFromAccount() to calculate & return
    // `removedBDV` to a known exploitable value. This causes LibSilo::stalkReward()
    // to return 0 due to an unsafe downcast of `removedBDV` from uint128 -> int96
    // meaning stalk rewards are not burned when the withdrawal occurs
    await this.silo.connect(user).withdrawDeposit(this.siloToken.address, stem, problemRemovedBdv,
    ↪  EXTERNAL);

    // verify token.totalDeposited has been correcly decremented
    expect(await this.silo.getTotalDeposited(this.siloToken.address)).to.eq("1");
    // verify token.totalDepositedBDV == 1 as siloToken has 1:1 BDV calc
    expect(await this.silo.getTotalDepositedBdv(this.siloToken.address)).to.eq("1");

    // verify totalStalk == 10000 which fails and instead 10010 is returned.
```

Recommended Mitigation: The withdrawal should revert if the result of the downcast overflows such that no Stalk is burned. This could be achieved by performing a safe downcast and/or validating that a non-zero Stalk amount is burned when withdrawing a non-zero BDV. `LibTokenSilo::toInt96` is used in that contract for validating inputs and casting.

7.4 Informational

7.4.1 Incorrect storage slot annotation in `Storage::SiloSettings`

While it appears that the order struct members in storage have not changed, the storage slot annotation of `Storage::SiloSettings` in `AppStorage.sol` is incorrect and should be updated as follows:

```
struct SiloSettings {
    bytes4 selector; //  4
-   uint32 stalkEarnedPerSeason; //   4  (16)
+   uint32 stalkEarnedPerSeason; //   4  (8)
-   uint32 stalkIssuedPerBdv; //      4  (8)
+   uint32 stalkIssuedPerBdv; //      4  (12)
-   uint32 milestoneSeason; //        4  (12)
+   uint32 milestoneSeason; //        4  (16)
    int96 milestoneStem; //           12 (28)
    bytes1 encodeType; //  1  (29)
    // 3 bytes are left here.
    uint128 gaugePoints; //   -------- 16
    bytes4 gpSelector; //                    4   (20)
    uint96 optimalPercentDepositedBdv; //  12  (32)
}
```

7.4.2 `LibLockedUnderlying` regression might not be representative of the expected behaviour

The percentage of locked liquidity, used in determining the L2SR in `LibEvaluate`, is obtained through the implementation of an on-chain look-up table based on an off-chain linear regression. The assumption considered acceptable for both Unripe Bean and Unripe LP is that 46,659 Unripe Tokens are chopped at each step. This number is calculated by dividing the number of Unripe Tokens by the number of Unripe Token Holders, resulting in an average of 46,659 Unripe Tokens held per Farmer with a non-zero balance. 2000 is used as a slight overestimation of the number of holders for both Unripe Bean and Unripe LP. An overestimation is acceptable because it results in a more conservative L2SR.

However, this average might not accurately represent what is expected in each Chop. For example, consider a scenario where 9 users each have 100,000 Unripe Tokens, and one user has 5.1 million Unripe Tokens.

$$\frac{Number\ of \ Unripe; Tokens}{Number\ of \ Unripe\ Token \ Holders}= \frac{9 \times 100.000 + 5.100.000}{10} =$$

$$\frac{900.000 + 5.100.000}{10} = \frac{6.000.000}{10}=600.000$$

In this case, the regression would consider that `600,000` Unripe Tokens are Chopped in each step, which can actually be done by just one single user. Therefore, here it would be better to use the mode or median values rather than the mean.

7.4.3 Outdated Seed Gauge System documentation in PR and inline comments

There are multiple instances in both the PR and inline comments where documentation of the Seed Gauge System is outdated. For example:

- `LibWhitelist::updateGaugeForToken` does not allow the gauge points to be changed. This is contrary to the comment in `WhitelistFacet::updateGaugeForToken`.
- The behavior of `LibGauge::getBeanToMaxLpGpPerBdvRatioScaled` is incorrectly documented as the reverse of its actual behavior, which is `f(0) = MIN_BEAN_MAX_LPGP_RATIO` and `f(100e18) = MAX_BEAN_MAX_LPGP_RATIO`.
- Gauge Points are not normalized to `100e18`, as stated in the PR.
- The `MIN_BEAN_MAX_LP_GP_PER_BDV_RATIO` constant is actually `50e18`, not `25e18` as stated in the PR.

- `gpPerBdv` and `beanToMaxLpGpPerBdvRatio` both have 18 decimal precision, but there are multiple comments which incorrectly state that these variables have 6 decimal precision.

7.4.4 Duplicated code between `LibChop` and `LibUnripe`

Duplicate versions of `LibUnripe::_getPenalizedUnderlying` and `LibUnripe::isUnripe` have been added to `LibChop`. This is not necessary as the logic that executes is identical, so these duplicate versions can be removed in favor of code reuse.

7.4.5 Outdated reference to urBEAN3CRV Convert

There are multiple instances in `LibConvert::getMaxAmountIn` and `LibConvert::getAmountOut` that reference conversion to urBEAN3CRV and BEAN:3CRV. Since the underlying liquidity has now been migrated to the BEAN:ETH Well, it is no longer possible to convert Unripe Bean or Unripe LP to BEAN:3CRV. Therefore, the following diff should be applied:

```
function getMaxAmountIn(address tokenIn, address tokenOut)
    internal
    view
    returns (uint256)
{
    /// BEAN:3CRV LP -> BEAN
    if (tokenIn == C.CURVE_BEAN_METAPOOL && tokenOut == C.BEAN)
        return LibCurveConvert.lpToPeg(C.CURVE_BEAN_METAPOOL);

    /// BEAN -> BEAN:3CRV LP
    if (tokenIn == C.BEAN && tokenOut == C.CURVE_BEAN_METAPOOL)
        return LibCurveConvert.beansToPeg(C.CURVE_BEAN_METAPOOL);

    // Lambda -> Lambda
    if (tokenIn == tokenOut)
        return type(uint256).max;

    // Bean -> Well LP Token
    if (tokenIn == C.BEAN && tokenOut.isWell())
        return LibWellConvert.beansToPeg(tokenOut);

    // Well LP Token -> Bean
    if (tokenIn.isWell() && tokenOut == C.BEAN)
        return LibWellConvert.lpToPeg(tokenIn);

-   // urBEAN3CRV Convert
+   // urBEAN:ETH Convert
    if (tokenIn == C.UNRIPE_LP){
-       // urBEAN:3CRV -> urBEAN
+       // urBEAN:ETH -> urBEAN
        if(tokenOut == C.UNRIPE_BEAN)
            return LibUnripeConvert.lpToPeg();
-       // UrBEAN:3CRV -> BEAN:3CRV
+       // UrBEAN:ETH -> BEAN:ETH
-       if(tokenOut == C.CURVE_BEAN_METAPOOL)
+       if(tokenOut == C.BEAN_ETH_WELL)
        return type(uint256).max;
    }

    // urBEAN Convert
    if (tokenIn == C.UNRIPE_BEAN){
-       // urBEAN -> urBEAN:3CRV LP
+       // urBEAN -> urBEAN:ETH LP
        if(tokenOut == C.UNRIPE_LP)
            return LibUnripeConvert.beansToPeg();
        // UrBEAN -> BEAN
        if(tokenOut == C.BEAN)
            return type(uint256).max;
    }

    revert("Convert: Tokens not supported");
}

function getAmountOut(address tokenIn, address tokenOut, uint256 amountIn)
    internal
    view
    returns (uint256)
{
    /// BEAN:3CRV LP -> BEAN
    if (tokenIn == C.CURVE_BEAN_METAPOOL && tokenOut == C.BEAN)
        return LibCurveConvert.getBeanAmountOut(C.CURVE_BEAN_METAPOOL, amountIn);

    /// BEAN -> BEAN:3CRV LP
    if (tokenIn == C.BEAN && tokenOut == C.CURVE_BEAN_METAPOOL)
        return LibCurveConvert.getLPAmountOut(C.CURVE_BEAN_METAPOOL, amountIn);
```

7.4.6 Miscellaneous NatSpec and inline comment errors

The following NatSpec errors were identified:

- `UnripeFacet::balanceOfPenalizedUnderlying` NatSpec is incorrectly copied from `UnripeFacet::balanceOfUnderlying` and should be modified for this particular function.
- The NatSpec of `LibWell::getTwaReservesForWell` is incorrect. It states that this function returns the `USD / TKN` price stored in `{AppStorage.usdTokenPrice}`; however, this is actually the `TKN / USD` price and should be updated accordingly.
- A comment explaining the implementation of`Weather::updateTemperature` incorrectly references `uint32(-change)` where it should instead be `uint256(-change)`.
- A comment in `LibCases` explaining the behavior of the constants incorrectly states `Bean2maxLpGpPerBdv set to 10% of current value` when it should be 50% of the current value.
- A comment in `InitBipNewSilo` has been changed to state the `stemStartSeason` is stored as a `uint32` when it is actually `uint16`.
- The NatSpec for the `int8[32] cases` member of `AppStorage` is outdated and along with the member itself should be marked as deprecated in favor of `bytes32[144] casesV2`.
- There is a slight error in the case of `TwaReserves` in the NatSpec of `AppStorage` which should instead be `twaReserves`.
- `deprecated_beanEthPrice` does not currently exist in the NatSpec of `AppStorage`. It should be added along with an explanation of why this member is deprecated.

7.4.7 Time-weighted average reserves should be read from the Beanstalk Pump in `LibWell` using a `try/catch` block

There are instances in `LibWell::getTwaReservesFromBeanstalkPump` and `LibWell::getTwaLiquidityFromBeanstalkPump` where the time-weighted average reserves are read directly from the Beanstalk Pump. Unlike the implementation in `LibWellMinting::twaDeltaB`, these functions do not wrap the call in a `try/catch` block. This should not affect the Beanstalk Sunrise mechanism as the execution of `LibWell::getTwaReservesFromStorageOrBeanstalkPump` will not reach the invocation of `LibWell::getTwaReservesFromBeanstalkPump`, since here the reserves are already set in storage, but consider handling Pump failure gracefully so that `LibEvaluate::calcLPToSupplyRatio` and `SeasonGettersFacet::getBeanEthTwaUsdLiquidity` (which is also used in `SeasonGettersFacet::getTotalUsdLiquidity`) do not revert if there is an issue.

7.4.8 Use of average grown stalk per BDV is not correctly documented

The nature of the Gauge Point system is to distribute new stalk among whitelisted LP and BEAN deposits based on their Bean-denominated value (BDV). The average grown stalk per BDV also takes into account the BDV of unripe assets, linked to their respective underlying asset based on the following ratio:

$$\frac{paidFertilizer}{mintedFertilzier} \times \frac{totalUnderlying(urAsset)}{supply(urAsset)}$$

While considering the BDV of unripe assets for the average grown stalk per BDV is mathematically correct, this metric lacks practical sense due to a portion of the average grown stalk per BDV never being issued, causing it to lose its semantic meaning.

```
// LibGauge::updateGrownStalkEarnedPerSeason
uint256 totalBdv = totalLpBdv.add(beanDepositedBdv);
...
uint256 newGrownStalk = uint256(s.seedGauge.averageGrownStalkPerBdvPerSeason)
    .mul(totalBdv) // This BDV does not include unripe asset BDV
    .div(BDV_PRECISION);
```

As can be seen, the clear intention of this calculation is to issue `newGrownStalk` for a season but only take into account the BDV corresponding to whitelisted LPs and BEAN. Given that it is never issued, the rest of the grown

stalk per BDV could be considered implicitly burned. This design decision should be better documented, making it clear how the unissued grown stalk is considered.

7.4.9 Consolidate unnecessary code duplication in `ConvertFacet::_withdrawTokens`

`ConvertFacet::_withdrawTokens` duplicates the following code in L119-132 then again in L137-151:

```
if (a.tokensRemoved.add(amounts[i]) < maxTokens) {
    //keeping track of stalk removed must happen before we actually remove the deposit
    //this is because LibTokenSilo.grownStalkForDeposit() uses the current deposit info
    // @audit start duplicated code
    depositBDV = LibTokenSilo.removeDepositFromAccount(
        msg.sender,
        token,
        stems[i],
        amounts[i]
    );
    bdvsRemoved[i] = depositBDV;
    a.stalkRemoved = a.stalkRemoved.add(
        LibSilo.stalkReward(
            stems[i],
            LibTokenSilo.stemTipForToken(token),
            depositBDV.toUint128()
        )
    );
    // @audit end duplicated code

} else {
    amounts[i] = maxTokens.sub(a.tokensRemoved);

    // @audit start duplicated code
    depositBDV = LibTokenSilo.removeDepositFromAccount(
        msg.sender,
        token,
        stems[i],
        amounts[i]
    );

    bdvsRemoved[i] = depositBDV;
    a.stalkRemoved = a.stalkRemoved.add(
        LibSilo.stalkReward(
            stems[i],
        LibTokenSilo.stemTipForToken(token),
            depositBDV.toUint128()
        )
    );
    // @audit end duplicated code
}
```

Consider refactoring to remove the duplicated code by changing the `if` condition to only update `amounts[i]` when required then perform the same processing that is currently on each `if/else` branch:

```
while ((i < stems.length) && (a.tokensRemoved < maxTokens)) {
    if (a.tokensRemoved.add(amounts[i]) >= maxTokens) {
        amounts[i] = maxTokens.sub(a.tokensRemoved);
    }

    //keeping track of stalk removed must happen before we actually remove the deposit
    //this is because LibTokenSilo.grownStalkForDeposit() uses the current deposit info
    depositBDV = LibTokenSilo.removeDepositFromAccount(
        msg.sender,
        token,
        stems[i],
        amounts[i]
    );
    bdvsRemoved[i] = depositBDV;
    a.stalkRemoved = a.stalkRemoved.add(
        LibSilo.stalkReward(
            stems[i],
            LibTokenSilo.stemTipForToken(token),
            depositBDV.toUint128()
        )
    );

    a.tokensRemoved = a.tokensRemoved.add(amounts[i]);
    a.bdvRemoved = a.bdvRemoved.add(depositBDV);

    depositIds[i] = uint256(LibBytes.packAddressAndStem(token, stems[i]));
    i++;
}
```

7.5 Gas Optimization

7.5.1 Break out of `LibWhitelist` loops early once the condition is met

Once the given address is found in the array passed to `LibWhitelist::checkTokenInArray` or `LibWhitelist::checkTokenNotInArray`, these functions could break early to avoid potentially unnecessary additional loop iterations.

```
/**
 * @notice Checks whether a token is in an array.
 */
function checkTokenInArray(address token, address[] memory array) private pure {
    // verify that the token is in the array.
    bool success;
    for (uint i; i < array.length; i++) {
-       if (token == array[i]) success = true;
+       if (token == array[i]) {
+           success = true;
+           break;
+       }
    }
    require(success, "Whitelist: Token not in whitelisted token array");
}

/**
 * @notice Checks whether a token is in an array.
 */
function checkTokenNotInArray(address token, address[] memory array) private pure {
    // verify that the token is not in the array.
    bool success = true;
    for (uint i; i < array.length; i++) {
-       if (token == array[i]) success = false;
+       if (token == array[i]) {
+           success = false;
+           break;
+       }
    }    require(success, "Whitelist: Token in incorrect whitelisted token array");
}
```

7.5.2 `LibBytes::packAddressAndStem` calculated twice with the same parameters

`LibSilo::_removeDepositsFromAccount` calls `LibBytes::packAddressAndStem` after `LibTokenSilo::removeDepositFromAccount` has already called the same function with the same parameters.

Consider refactoring to calculate `LibBytes::packAddressAndStem` once for each loop iteration in `LibSilo::_removeDepositsFromAccount`, then pass the result as a parameter in the call to `LibTokenSilo::removeDepositFromAccount`.

7.5.3 `LibTokenSilo::stemTipForToken` calculated multiple times with same parameter

`LibTokenSilo::stemTipForToken` is calculated multiple times with the same parameter in `LibSilo::_removeDepositsFromAccount`. This wastes gas since `LibTokenSilo::stemTipForToken` is always called with the same `token` parameter during bulk withdrawals, performing 4 SLOAD operations on storage that does not change.

Consider calculating the stem tip once before entering the loop then pass the result as a parameter to `stalkReward()`.

The same issue also occurs in `ConvertFacet::_withdrawTokens`.

7.5.4 `SiloFacet::transferDeposits` should only call `LibSiloPermit::_spendDepositAllowance` once

`SiloFacet::transferDeposits` currently loops through the input `amounts` array and calls `LibSiloPermit::_spendDepositAllowance` once for each `amounts[i]`.

Instead, consider having a `totalAmount` stack variable that is incremented for each `amounts[i]` when looping through the inputs. Then, after the initial loop is complete, call `LibSiloPermit::_spendDepositAllowance` with `totalAmount` to save a significant number of storage reads & writes.

Consider this simplified example using Foundry:

```
uint256 s_allowance = 10;

function _spendAllowance(uint256 amount) private {s_allowance-=amount;}

function testBulkTransfer1() public {
    // prepare input
    uint256[10] memory amounts;
    for(uint256 i=0; i<10; i++){amounts[i] = 1;}

    // function implementation; update storage 1-by-1
    for (uint256 i = 0; i < amounts.length; ++i) {
        _spendAllowance(amounts[i]);
    }

    assert(s_allowance == 0);
}

function testBulkTransfer2() public {
    // prepare input
    uint256[10] memory amounts;
    for(uint256 i=0; i<10; i++){amounts[i] = 1;}

    // function implementation; cache total amount, update storage once
    uint256 totalSpend;
    for (uint256 i = 0; i < amounts.length; ++i) {
        totalSpend += amounts[i];
    }

    _spendAllowance(totalSpend);

    assert(s_allowance == 0);
}

[PASS] testBulkTransfer1() (gas: 5494)
[PASS] testBulkTransfer2() (gas: 3435)
```

7.5.5 Cache updated remaining amount to prevent extra storage read

`FundraiserFacet::fund` should save the calculated `remaining - amount` then use it to set storage in L125 and to check for completion in L128; this prevents re-reading storage again in L128. One easy solution is to reuse the existing `remaining` stack variable:

```
remaining = remaining - amount; // Note: SafeMath is redundant here.
s.fundraisers[id].remaining = remaining;
emit FundFundraiser(msg.sender, id, amount);

// If completed, transfer tokens to payee and emit an event
if (remaining == 0) {
    _completeFundraiser(id);
}
```

Consider this simplified example using Foundry:

```
uint256 private s_remainingDebt = 10;

function _onDebtRepayment() private {}

function testRemaining1() public {
    uint256 repaymentAmount = 10;

    // update storage
    s_remainingDebt -= repaymentAmount;

    // use storage read for check
    if(s_remainingDebt == 0) {
        _onDebtRepayment();
    }

    assert(s_remainingDebt == 0);
}

function testRemaining2() public {
    uint256 repaymentAmount = 10;

    // cache remaining debt
    uint256 remainingDebt = s_remainingDebt - repaymentAmount;
    // update storage
    s_remainingDebt = remainingDebt;

    // use cache for check
    if(remainingDebt == 0) {
        _onDebtRepayment();
    }

    assert(s_remainingDebt == 0);
}

[PASS] testRemaining1() (gas: 621)
[PASS] testRemaining2() (gas: 563)
```

7.5.6 Cache recapitalized amount to prevent extra storage read

`LibFertilizer::remainingRecapitalization` should cache `s.recapitalized` then use the cached stack variable in L166-167 to prevent reading the same value a second twice from storage.

8 Appendix

8.1 Appendix A. Locked Underlying Differential Test

Locked underlying CSV:

Run test with `yarn hardhat test --grep 'Unripe supply > 1.000.000'`:

```
const { expect } = require('chai');

const { ethers } = require('hardhat');
const csv = require('csv-parser')
const fs = require('fs')

const RECAP_PERCENTAGE_DECIMALS = 6
const UNRIPE_TOKENS_DECIMALS = 6
const LOCKED_PERCENTAGE_DECIMALS = 18
let expectedLockedRatios = new Map()
let libLockedUnderlyingFacet

/**
 * CSV
 *      * MUST BE in protocol/test/BIP39/DifferentialTestingData/getLockedUnderlying.csv
 *      * CSV headers MUST BE recapPercentage,urSupply,lockedPercentage
 *      * recapPercentage MUST BE with just 6 decimals or less
 *      * urSupply MUST BE with just 6 decimals or less
 *      * lockedPercentage MUST BE with 18 decimals or less
 * @returns expectedValues Mapping (unripeSupply, recapPercentage)=>ExpectedlockedRation
 */

const getExpectedOutputsMappingFromCSV = async()=>{
    //const expectedValues = new Map()
    await new Promise((resolve,reject)=>{
        fs.createReadStream("./test/BIP39/DifferentialTestingData/getLockedUnderlying.csv")
        .pipe(csv())
        .on('data',(data)=>{
            let recapPercentage =
            ↪  ethers.utils.parseUnits(data.recapPercentage,RECAP_PERCENTAGE_DECIMALS)
            let unripeSupply = ethers.utils.parseUnits(data.urSupply,UNRIPE_TOKENS_DECIMALS)
            let lockedPercentage =
            ↪  ethers.utils.parseUnits(data.lockedPercentage,LOCKED_PERCENTAGE_DECIMALS)
            expectedLockedRatios.set({
                unripeSupply: unripeSupply,
                recapPercentage: recapPercentage
            },lockedPercentage)
        }).on('end',resolve).on('error',reject)
    })

}

const getPercentageLockedUnderlying = async(unripeSupply, recapPercentage)=>{
    let lockedLiquidityPercentageTx = await
    ↪  libLockedUnderlyingFacet.getPercentLockedUnderlying(unripeSupply,recapPercentage)
    let lockedLiquidityPercentageReceipt = await lockedLiquidityPercentageTx.wait()
    const [_,percentageEvent] = lockedLiquidityPercentageReceipt.events
    const {percentage} = percentageEvent.args
    return percentage
}


describe('LibLockedUnderlying.getPercentLockedUnderlying', async function () {

    before(async()=>{
        // Deploy library
        const LibLockedUnderlyingFactory = await ethers.getContractFactory("LibLockedUnderlying")

        const libLockedUnderlying = await LibLockedUnderlyingFactory.deploy()
        await libLockedUnderlying.deployed()

        // Deploy mock

        LibLockedUnderlyingFacetFactory = await ethers.getContractFactory(
            "MockLibLockedUnderlyingFacet",{
                libraries:{
```

8.2 Appendix B. Mainnet Rounding Error Tests

Run all tests with `yarn hardhat test --grep 'SiloToken: Mainnet Rounding Error'`:

```
const { expect } = require('chai');
const { mine } = require("@nomicfoundation/hardhat-network-helpers");
const { takeSnapshot, revertToSnapshot } = require("../utils/snapshot.js");
const { BEAN, BEAN_3_CURVE, UNRIPE_BEAN, UNRIPE_LP, WETH, BEAN_ETH_WELL, PUBLIUS,
↪  ETH_USD_CHAINLINK_AGGREGATOR } = require('../utils/constants.js');
const { to6 } = require('../utils/helpers.js');
const { bipSeedGauge } = require('../../scripts/bips.js');
const { getBeanstalk } = require('../../utils/contracts.js');
const { impersonateBeanstalkOwner, impersonateSigner } = require('../../utils/signer.js');
const { ethers } = require('hardhat');
const { impersonateBean, impersonateEthUsdChainlinkAggregator} =
↪  require('../../scripts/impersonate.js');
const { EXTERNAL } = require('../utils/balances.js');
//import { diamondFacetData, attachNewCodeToFacet } from '../utils/diamondFacet.js';

let seasonFacet, silo, siloFacet, tokenSilo, admin, well, weth, bean, beanEth, beanEthToken, unripeLp,
↪  unripeBean, beanMetapool, chainlink, migrationFacet, diamondFacet, siloExit, convertFacet, bdvFacet

let snapshotId

const FORKING_BLOCK = 18577183 - 1 // Block before this migration:
↪  https://etherscan.io/tx/0xd056dff2e0ec7609aa32de780a81ed27bcb7a0de6f6d8814a005a2889c2b804a

let legacyUnripeHolderAddress = "0xf84f39554247723c757066b8fd7789462ac25894"
let unripeBeanHolderAddress = "0xa82240Bb0291A8Ef6e46a4f6B8ABF4737B0b5257"

let inputData = {
    "account": "0xf84f39554247723c757066b8fd7789462ac25894",
    "tokens": [
        "0x1bea0050e63e05fbb5d8ba2f10cf5800b6224449",
        "0x1bea3ccd22f4ebd3d37d731ba31eeca95713716d",
        "0xbea0000029ad1c77d3d5d23ba2d8893db9d1efab"
    ],
    "seasons": [
      [
        1064,
        1281,
        1282,
        1766,
        1767,
        2454,
        2626,
        3367,
        5122,
        6022,
        6074
      ],
      [1044],
      [10795]
    ],
    "amounts": [
      [
        89681039,
        23603649,
        2260255,
        17652540,
        33361024,
        241605139,
        200065458,
        859438810,
        47598426,
        23059093917,
        1728409109
      ],
      [1232458169],
      [286296869]
```

Note that the issue only applies for new deposits of unripe tokens.