Pinto protocol PR 72

Security Review

Cantina Managed review by:

Om Parikh, Security Researcher
T1moh, Associate Security Researcher

May 31, 2025

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Pinto is low volatility money built on Base. Pinto's primary objective is to incentivize independent market participants to regularly cross the price of 1 Pinto over its 1 Dollar peg in a sustainable fashion.

From Apr 16th to May 1st the Cantina team conducted a review of pinto-protocol on commit hash 7d3e7055. The team identified a total of **23** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	3	3	0
Low Risk	5	5	0
Gas Optimizations	0	0	0
Informational	15	14	1
Total	**23**	**22**	**1**

3 Findings

3.1 Medium Risk

3.1.1 `LibSilo._setSortedDepositIds()` doesn't update depositId index

Severity: Medium Risk

Context: LibSilo.sol#L760-L767

Description: There is new function `SiloFacet.updateSortedDepositIds()`, which sets user's deposits in sorted order. There are 2 variables of deposit list:

```
/**
 * @notice This struct stores data for a deposit list for a given token.
 * a mapping from id to index was created to allow for O(1) retrieval of a deposit from the list.
 * @param depositIds An array of depositIds for a given token.
 * @param idIndex A mapping from depositId to index in depositIds.
 */
struct DepositListData {
    uint256[] depositIds;
    mapping(uint256 => uint256) idIndex;
}
```

The problem is that only `depositIds` is updated:

```
function _setSortedDepositIds(
    address account,
    address token,
    uint256[] calldata sortedDepositIds
) internal {
    AppStorage storage s = LibAppStorage.diamondStorage();
    s.accts[account].depositIdList[token].depositIds = sortedDepositIds;
}
```

Suppose the following scenario:

1. User has deposits S2 and S1. `depositIds = [S2, S1]`, entries of `idIndex` are: S2→ 0, S1→ 1.
2. Deposits are sorted. Now `depositIds = [S1, S2]`, but entries of `idIndex` stay the same.
3. User withdraws S2. After calling `removeDepositIDfromAccountList()`, storage is following: `depositIds = [S2]`, idIndex: S1→ 1.

```
/**
 * @notice removes an depositID from an account's depositID list.
 */
function removeDepositIDfromAccountList(
    address account,
    address token,
    uint256 depositId
) internal {
    AppStorage storage s = LibAppStorage.diamondStorage();
    DepositListData storage list = s.accts[account].depositIdList[token];
    uint256 i = findDepositIdForAccount(account, token, depositId);
    list.depositIds[i] = list.depositIds[list.depositIds.length - 1];
    list.idIndex[list.depositIds[i]] = i;
    list.idIndex[depositId] = type(uint256).max;
    list.depositIds.pop();
}

/**
 * @notice given an depositId, find the index of the depositId in the account's deposit list.
 */
function findDepositIdForAccount(
    address account,
    address token,
    uint256 depositId
) internal view returns (uint256 i) {
    AppStorage storage s = LibAppStorage.diamondStorage();
    i = s.accts[account].depositIdList[token].idIndex[depositId];
}
```

4. User withdraws S1. However it reverts because `idIndex[S1]` points to 1, therefore tries to access out of bound index of array `depositIds = [S2]`.

Revert can be escaped in case user does partial withdraw and not full. Therefore Impact is Medium: 1) temporary freeze of funds until root cause is discovered; 2) full withdrawals automated via Tractor will revert, breaking the intention of upgrade.

Recommendation: Update mapping `s.accts[account].depositIdList[token].idIndex` after sorting deposits.

Pinto: This is resolved in PR 80.

Cantina Managed: Fix verified.

3.1.2 `supplyPodDemandScalar` and `initialSoilPodDemandScalar` are never set

Severity: Medium Risk

Context: LibEvaluate.sol#L377-L378, LibEvaluate.sol#L386-L387

Description: PI-9 introduces formula to calculate `soilDemandThreshold`. It's the minimum required to calculate `deltaPodDemand`, otherwise it can be manipulated on low amount:

```
function calcDeltaPodDemand(
    uint256 dsoil
)
    internal
    view
    returns (Decimal.D256 memory deltaPodDemand, uint32 lastSowTime, uint32 thisSowTime)
{
    Weather storage w = LibAppStorage.diamondStorage().sys.weather;

    // get the minimum soil sown threshold needed to calculate delta pod demand.
    uint256 minDemandThreshold = calcMinSoilDemandThreshold();

    // not enough soil sown, consider demand to be decreasing, reset sow times.
    if (dsoil < minDemandThreshold) { // <<<
        return (Decimal.zero(), w.thisSowTime, type(uint32).max);
    }
```

THe problem is that values `s.sys.extEvaluationParameters.supplyPodDemandScalar` and `s.sys.extEvaluationParameters.initialSoilPodDemandScalar` are never initialized, therefore `calcMinSoilDemand()` returns 0 as before an upgrade:

```
function calcMinSoilDemandThreshold() internal view returns (uint256 minDemandThreshold) {
    AppStorage storage s = LibAppStorage.diamondStorage();
    uint256 beanSupply = BeanstalkERC20(s.sys.bean).totalSupply();

    // calculate the minimum threshold of bean to calculate delta pod demand.
    uint256 calculatedThreshold = (beanSupply * // <<<
        s.sys.extEvaluationParameters.supplyPodDemandScalar) / BEAN_PRECISION;

    // take the maximum of the calculated threshold and the minimum supply bound.
    uint256 beanSupplyThreshold = calculatedThreshold > MIN_BEAN_SUPPLY_BOUND
        ? calculatedThreshold
        : MIN_BEAN_SUPPLY_BOUND;

    // scale s.sys.initialSoil by the initialSoilPodDemandScalar, currently 25%.
    uint256 scaledInitialSoil = (s.sys.initialSoil *
        s.sys.extEvaluationParameters.initialSoilPodDemandScalar) / SOIL_PRECISION; // <<<

    // take the minimum of the scaled initial soil and the bean supply threshold.
    minDemandThreshold = beanSupplyThreshold < scaledInitialSoil
        ? beanSupplyThreshold
        : scaledInitialSoil;
}
```

Recommendation: Set them in new `InitPI10.sol` contract.

Pinto: This is resolved in PR 82.

Cantina Managed: Fix verified.

3.1.3 Mistake in implementation of `convertDown` penalty formula

Severity: Medium Risk

Context: GaugeFacet.sol#L168-L169

Description: PI-7 introduces grownStalk penalty on converts Pinto$\rightarrow$ LP when Pinto price is < 1.025 USD. Penalty is defined by following formula:

> Therefore, we define the percent Grown Stalk Penalty for Converting Pinto to LP Deposits above the value target (g^*) for a given liquidity and time weighted average price over the previous Season (P_{t-1}), P^{upper}, G_t, L2SR over the previous Season (R^W), optimal L2SR (R^{W^*}), and G^{max} as:

$$g^* = \begin{cases} 0 & \text{if} P_{t-1} > P^{upper} \\ max(0, \frac{\min(R^W, R^{W^*})}{R^{W^*}} \times (1 - \frac{\log_2(\mathfrak{G}_t+1)}{\log_2(\mathfrak{G}^{max}+1)})) & \text{otherwise} \end{cases}$$

Function `GaugeFacet.convertDownPenaltyGauge()` implements it. `l2srRatio` reflects first multiplier in formula:

```
function convertDownPenaltyGauge(
    bytes memory value,
    bytes memory systemData,
    bytes memory gaugeData
) external view returns (bytes memory, bytes memory) {
    // ...

    // Scale L2SR by the optimal L2SR.
    uint256 l2srRatio = (1e18 * bs.lpToSupplyRatio.value) / // <<<
        s.sys.evaluationParameters.lpToSupplyRatioOptimal;

    uint256 timeRatio = (1e18 * PRBMathUD60x18.log2(rollingSeasonsAbovePeg * 1e18 + 1e18)) /
        PRBMathUD60x18.log2(rollingSeasonsAbovePegCap * 1e18 + 1e18);

    penaltyRatio = Math.min(1e18, (l2srRatio * (1e18 - timeRatio)) / 1e18);
    return (abi.encode(penaltyRatio, rollingSeasonsAbovePeg), gaugeData);
}
```

The problem is that it misses `min` in numerator, it simply divides `Rw` by `Rw'`. As a result, `l2srRatio` can have value greater than 1 contrary to specification.

Consider following example:

- `lpToSupplyRatio = 0.5e18 (50%); lpToSupplyRatioOptimal = 0.4e18 (40%).`
- It should calculate `min(50%, 40%) / 40% = 100% (1e18).`
- However it calculates `1e18 * 0.5e18 / 0.4e18 = 125% (1.25e18).`

As a result, `penaltyRatio` can have greater value than expected. It results in extra penalty to users.

Recommendation:

```
  // Scale L2SR by the optimal L2SR.
- uint256 l2srRatio = (1e18 * bs.lpToSupplyRatio.value) /
+ uint256 l2srRatio = (1e18 * Math.min(bs.lpToSupplyRatio.value,
↪   s.sys.evaluationParameters.lpToSupplyRatioOptimal) /
      s.sys.evaluationParameters.lpToSupplyRatioOptimal;
```

Pinto: This is resolved in PR 81.

Cantina Managed: Fix verified.

3.2 Low Risk

3.2.1 `LibWell.calculateTokenBeanPriceFromReserves()` should ensure there is enough liquidity

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: There is constant `C.WELL_MINIMUM_BEAN_BALANCE = 10e6` which is used to ensure there is enough liquidity in Well during calculation of Bean price. That's important because of how price is calculated:

1. It calculates current LP token supply;
2. Decreases bean reserve by 1 Bean.
3. Calculates the amount of another reserve to reflect LP supply from step 1.
4. Difference of nonBeanTokenReserve is the Token/Bean price.

```
function calculateTokenBeanPriceFromReserves(
    address well,
    uint256 beanIndex,
    uint256 nonBeanIndex,
    uint256[] memory reserves,
    Call memory wellFunction
) internal view returns (uint256 price) {
    // attempt to calculate the LP token Supply.
    try
        IBeanstalkWellFunction(wellFunction.target).calcLpTokenSupply(
            reserves,
            wellFunction.data
        )
    returns (uint256 lpTokenSupply) {
        address nonBeanToken = address(IWell(well).tokens()[nonBeanIndex]);
        uint256 oldReserve = reserves[nonBeanIndex];
        reserves[beanIndex] = reserves[beanIndex] + BEAN_UNIT;

        try
            IBeanstalkWellFunction(wellFunction.target).calcReserve(
                reserves,
                nonBeanIndex,
                lpTokenSupply,
                wellFunction.data
            )
        returns (uint256 newReserve) {
            // Measure the delta of the non bean reserve.
            // Due to the invariant of the well function, old reserve > new reserve.
            uint256 delta = oldReserve - newReserve;
            price = (10 ** (IERC20Decimals(nonBeanToken).decimals() + 6)) / delta;
        } catch {
            return 0;
        }
    } catch {
        return 0;
    }
}
```

This way slippage can result in big price deviation in case Bean liquidity is low, that's why usually it early returns 0 signalising error. Everywhere except this function.

Recommendation: Return 0 price in case bean reserve is less than `C.WELL_MINIMUM_BEAN_BALANCE`.

Pinto: This is resolved in PR 85.

Cantina Managed: Fix verified.

3.2.2 `SeasonGettersFacet.getTotalUsdLiquidity()` returns incorrect value

Severity: Low Risk

Context: LibWell.sol#L478

Description: `getTwaLiquidityFromPump()` returns `usdLiquidity` of Well with precision of `10 ** nonBeanToken.decimals()`:

```
function getTwaLiquidityFromPump(
    address well,
    uint256 tokenUsdPrice
) internal view returns (uint256 usdLiquidity) {
    AppStorage storage s = LibAppStorage.diamondStorage();
    (, uint256 j) = getNonBeanTokenAndIndexFromWell(well);
    Call[] memory pumps = IWell(well).pumps();
    try
        ICumulativePump(pumps[0].target).readTwaReserves(
            well,
            s.sys.wellOracleSnapshots[well],
            uint40(s.sys.season.timestamp),
            pumps[0].data
        )
    returns (uint[] memory twaReserves, bytes memory) {
        usdLiquidity = tokenUsdPrice.mul(twaReserves[j]).div(1e6); // <<<
    } catch {
        // if pump fails to return a value, return 0.
        usdLiquidity = 0;
    }
}
```

Suppose WBTC price is 80_000 USD, and there are 5 WBTC in Well. Then `usdLiquidity = 80_000e6 * 5e8 / 1e6 = 400_000e8`. The problem is that `SeasonGettersFacet.getTotalUsdLiquidity()` sums all values together for all Wells. As a result, liquidity of non-18 decimal tokens in Wells remains unnoticed.

```
function getTotalUsdLiquidity() external view returns (uint256 totalLiquidity) {
    address[] memory wells = LibWhitelistedTokens.getWhitelistedWellLpTokens();
    for (uint i; i < wells.length; i++) {
        totalLiquidity = totalLiquidity.add(getTwaLiquidityForWell(wells[i]));
    }
}

function getTwaLiquidityForWell(address well) public view returns (uint256) {
    (address token, ) = LibWell.getNonBeanTokenAndIndexFromWell(well);
    return LibWell.getTwaLiquidityFromPump(well, LibUsdOracle.getTokenPrice(token));
}
```

Recommendation: UsdLiquidity variables have 1e18 precision in other parts of Pinto protocol, so here should be the same.

Pinto: This is resolved in PR 84.

Cantina Managed: Fix verified.

3.2.3 Consider implementing slippage protection in converts

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: Convert functionality allows users to convert Silo deposit into Silo deposit of different token. Convert is one of peg maintenance mechanisms. When users perform unfavourable convert from perspective of peg mechanism, penalty is applied.

There are multiple types of penalty:

1. Penalty based on convert capacity:

```
/**
 * @notice Calculates the stalk penalty for a convert. Updates convert capacity used.
 */
function prepareStalkPenaltyCalculation(
    address inputToken,
    address outputToken,
    LibConvert.DeltaBStorage memory dbs,
    uint256 overallConvertCapacity,
    uint256 fromBdv,
    uint256[] memory initialLpSupply
) public returns (uint256) {
```

2. New penalty on Pinto$\rightarrow$ LP_Tokens convert introduced in PI-7.

There is possibility that user doesn't expect penalty and therefore does convert, however actually penalty is applied. Suppose following scenario:

1. In current situation there is no penalty. UserA sends transaction to perform a convert.
2. UserB also notices there is no penalty. UserB sends similar transaction.
3. But only first transaction won't be charged. Because there is no non-penalty capacity for both.

As a result, either UserA or UserB will be unexpectedly charged.

Recommendation: To prevent this possibility, introduce argument `minExpectedGrownStalk` in `ConvertFacet.convert()` and `PipelineConvertFacet.pipelineConvert()`. Lambda and Anti Lambda converts should not use it.

Pinto: This is resolved in PR 83 via `stalkslippage` as a parameter.

Cantina Managed: Fix verified.

3.2.4 User can't configure slippage to receive bonus

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: With introduction of PR 10, now there is bonus on conversions `WELL_LP` → `BEAN`, i.e. user can receive stalk bonus.

Slippage protection was introduced as a fix to issue `Consider implementing slippage protection in converts`. Now it checks that user receives specified minimum in grown stalk amount:

```
function checkGrownStalkSlippage(
    uint256 newGrownStalk,
    uint256 originalGrownStalk,
    uint256 grownStalkSlippage
) internal view {
    // cap grown stalk slippage at 100%
    if (grownStalkSlippage > MAX_GROWN_STALK_SLIPPAGE)
        grownStalkSlippage = MAX_GROWN_STALK_SLIPPAGE;
    uint256 minimumStalk = originalGrownStalk
        .mul(MAX_GROWN_STALK_SLIPPAGE - grownStalkSlippage)
        .div(MAX_GROWN_STALK_SLIPPAGE);
    require(newGrownStalk >= minimumStalk, "Convert: Stalk slippage");
}
```

`grownStalkSlippage` is e18 coefficient specifying penalty tolerance, max value is `1e18`. However current design doesn't allow to specify "negative" slippage, i.e. user wants Convert to succeed only if there is positive bonus applied.

Suppose following scenario:

1) User has 100 grown Stalk initially.
2) User executes Convert aiming to claim stalk bonus and avoid penalty.
3) Wants Convert to succeed only if there is minimum 105 grown Stalk in the end.
4) Problem is that he can't specify such `grownStalkSlippage`. He can specify to receive at max 100 grown Stalk.

Recommendation: Consider to refactor design to enable "negative" slippage, i.e. to specify desired stalk bonus.

Pinto: Fixed in commit 5779a96.

Cantina Managed: Fix verified.

3.2.5 `stalkPenaltyBdv` is underestimated in Convert

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: Previously it was a requirement to perform a `Convert` without penalty:

```
function checkForValidConvertAndUpdateConvertCapacity(
    PipelineConvertData memory pipeData,
    bytes calldata convertData,
    address fromToken,
    address toToken,
    uint256 fromBdv
) public {
    LibConvertData.ConvertKind kind = convertData.convertKind();
    if (
        kind == LibConvertData.ConvertKind.BEANS_TO_WELL_LP ||
        kind == LibConvertData.ConvertKind.WELL_LP_TO_BEANS
    ) {
        pipeData.overallConvertCapacity = LibConvert.abs(LibDeltaB.overallCappedDeltaB());

        pipeData.stalkPenaltyBdv = prepareStalkPenaltyCalculation(...);

        require(pipeData.stalkPenaltyBdv == 0, "Convert: Non-zero Stalk Penalty"); // <<<
    }
```

However now with introduction of `grownStalkSlippage` you allow it. You just apply `stalkPenaltyBdv` like in `PipelineConvert`:

```
function checkForValidConvertAndUpdateConvertCapacity(
    PipelineConvertData memory pipeData,
    bytes calldata convertData,
    address fromToken,
    address toToken,
    uint256 fromBdv,
    uint256 toBdv
) public returns (uint256 grownStalk) {
    LibConvertData.ConvertKind kind = convertData.convertKind();
    if (
        kind == LibConvertData.ConvertKind.BEANS_TO_WELL_LP ||
        kind == LibConvertData.ConvertKind.WELL_LP_TO_BEANS
    ) {
        pipeData.overallConvertCapacity = LibConvert.abs(LibDeltaB.overallCappedDeltaB());

        pipeData.stalkPenaltyBdv = prepareStalkPenaltyCalculation(
            fromToken,
            toToken,
            pipeData.deltaB,
            pipeData.overallConvertCapacity,
            fromBdv,
            pipeData.initialLpSupply
        );

        // apply penalty to grown stalk as a % of bdv converted. See {LibConvert.executePipelineConvert}
        grownStalk = (pipeData.grownStalk * (toBdv - pipeData.stalkPenaltyBdv)) / toBdv;
    } else {
        // apply no penalty to non BEANS_TO_WELL_LP or WELL_LP_TO_BEANS conversions.
        grownStalk = pipeData.grownStalk;
    }
}
```

And very old wrong code now becomes to have impact. Let's first analyze how `stalkPenaltyBdv` is calculated in `PipelineConvert`. Main difference is that it passes `newBdv` instead of `fromBdv` to `prepareStalkPenaltyCalculation()`:

```
function executePipelineConvert(
    address inputToken,
    address outputToken,
    uint256 fromAmount,
    uint256 fromBdv,
    uint256 initialGrownStalk,
    AdvancedPipeCall[] memory advancedPipeCalls
) external returns (uint256 toAmount, uint256 newGrownStalk, uint256 newBdv) {
    // ...

    newBdv = LibTokenSilo.beanDenominatedValue(outputToken, toAmount);

    // Calculate stalk penalty using start/finish deltaB of pools, and the capped deltaB is
    // passed in to setup max convert power.
    pipeData.stalkPenaltyBdv = prepareStalkPenaltyCalculation(
        inputToken,
        outputToken,
        pipeData.deltaB,
        pipeData.overallConvertCapacity,
        newBdv,
        pipeData.initialLpSupply
    );

    // ...
}
```

There is long chain of calls. Finally, this parameter named `bdvConverted` is used only to cap final value of `stalkPenaltyBdv` to 100%:

```
function calculateStalkPenalty(
    DeltaBStorage memory dbs,
    uint256 bdvConverted,
    uint256 overallConvertCapacity,
    address inputToken,
    address outputToken
)
    internal
    view
    returns (
        uint256 stalkPenaltyBdv,
        uint256 overallConvertCapacityUsed,
        uint256 inputTokenAmountUsed,
        uint256 outputTokenAmountUsed
    )
{
    // ...

    // Cap amount of bdv penalized at amount of bdv converted (no penalty should be over 100%)
    stalkPenaltyBdv = min(
        max(spd.higherAmountAgainstPeg, spd.convertCapacityPenalty),
        bdvConverted // <<<
    );

    // ...
}
```

`stalkPenaltyBdv` itself is always applied to `newBdv`, like here:

```
newGrownStalk = (initialGrownStalk * (newBdv - pipeData.stalkPenaltyBdv)) / newBdv;
```

It means that cap value must also be `newBdv`. Convert `WELL_LP` ↔ `BEAN` allows to only increase bdv, it doesn't decrease. As a result, `stalkPenaltyBdv` is capped by lower than intended value, underestimating penalty.

This root cause always existed. Previously it didn't use value of `stalkPenaltyBdv`, instead simply compared to 0, therefore zero impact.

Recommendation:

```
  function checkForValidConvertAndUpdateConvertCapacity(
      PipelineConvertData memory pipeData,
      bytes calldata convertData,
      address fromToken,
      address toToken,
      uint256 fromBdv,
      uint256 toBdv
  ) public returns (uint256 grownStalk) {
      // ...

          pipeData.stalkPenaltyBdv = prepareStalkPenaltyCalculation(
              fromToken,
              toToken,
              pipeData.deltaB,
              pipeData.overallConvertCapacity,
-             fromBdv,
+             toBdv,
              pipeData.initialLpSupply
          );

          // ...
  }
```

Pinto: Updated in commit 5779a96.

Cantina Managed: Fix verified.

3.3 Informational

3.3.1 Confusing name of `deltaT`

Severity: Informational

Context: LibGaugeHelpers.sol#L47

Description: PI-10 contains detailed formulas involved into grownStalk bonus calculations during Convert. Formulas use variable `deltaD`. However in code its value is stored in struct `LibGaugeHelpers.ConvertBonusGaugeData` in the field `deltaT`.

Recommendation: Rename in struct and other places.

```
- uint256 deltaT;
+ uint256 deltaD;
```

Pinto: Resolved in PR 98.

Cantina Managed: Fix verified.

3.3.2 Remove `CULTIVATION_FACTOR_PRECISION`

Severity: Informational

Context: Sun.sol#L38

Description: Currently this constant variable is not used anywhere, and additionally it reflects incorrect value. Because actual precision is `100e6 = 100%` and not `1e6`.

Recommendation: Remove `CULTIVATION_FACTOR_PRECISION`.

Pinto: Resolved in PR 97.

Cantina Managed: Fix verified.

3.3.3 Confusing Natspec in `LibEvaluate.calcMinSoilDemandThreshold()`

Severity: Informational

Context: LibEvaluate.sol#L369

Description: In PI-9 you describe formula used to calculate min soil. Actual code and specification use `min` of final values, however Natspec wrongly uses `max`:

```
    // take the minimum of the scaled initial soil and the bean supply threshold.
    minDemandThreshold = beanSupplyThreshold < scaledInitialSoil
        ? beanSupplyThreshold
        : scaledInitialSoil;
}
```

Recommendation:

```
- * @dev if not max(max(0,001% of supply, 10), 25% of soil issued)
+ * @dev if not min(max(0,001% of supply, 10), 25% of soil issued)
```

Pinto: Resolved in PR 96.

Cantina Managed: Fix verified.

3.3.4 Confusing comment in `GaugeDefault.defaultGaugePoints()`

Severity: Informational

Context: GaugeDefault.sol#L21-L23, GaugeDefault.sol#L78

Description: `GaugeDefault.defaultGaugePoints()` calculates new amount of gauge points. If percent of deposited Pdv is within `relativelyClose` range, amount of gauge points per Well stays the same. Previously it was calculated using `UPPER_THRESHOLD` and `LOWER_THRESHOLD`, but now uses `RELATIVELY_CLOSE = 10%`:

```
if (percentOfDepositedBdv > getRelativelyCloseAbove(optimalPercentDepositedBdv)) {
    // ...
} else if (percentOfDepositedBdv < getRelativelyCloseBelow(optimalPercentDepositedBdv)) {
    // ...
} else {
    // If % of deposited BDV is .5% within range of optimal,
    // keep gauge points the same.
    return currentGaugePoints;
}
```

Recommendation: Update comment, and remove unused variables:

```
- uint256 private constant UPPER_THRESHOLD = 10050;
- uint256 private constant LOWER_THRESHOLD = 9950;
- uint256 private constant THRESHOLD_PRECISION = 10000;

- // If % of deposited BDV is .5% within range of optimal,
+ // If % of deposited BDV is 10% within range of optimal,
```

Pinto: Resolved in PR 95.

Cantina Managed: Fix verified.

3.3.5 Incorrect values in `LibCases.sol` description

Severity: Informational

Context: LibCases.sol#L46-L47

Description: Comments in `LibCases.sol` describe the logic behind implementation, and explains hex values, particularly:

```
 * -50: FFFD4A1C50E94E780000 = -50e18
 * +1:  FFFA9438A1D29CF00000 = 1e18
 * +2:  FFF7DE54F2BBF1680000 = 2e18
```

However hex value corresponds to different decimal representation. For example `0xFFFA9438A1D29CF00000 = -1.2e18`.

Recommendation:

```
- * +1:  FFFA9438A1D29CF00000 = 1e18
- * +2:  FFF7DE54F2BBF1680000 = 2e18
+ * +1:  00000DE0B6B3A7640000 = 1e18
+ * +2:  00001BC16D674EC80000 = 2e18
```

Pinto: Resolved in PR 94.

Cantina Managed: Fix verified.

3.3.6 Inconsistent structure of "if ladder" in `LibIncentive.fracExp()`

Severity: Informational

Context: LibIncentive.sol#L393-L396

Description: `LibIncentive.fracExp()` uses ladder of "if" cases to return hardcoded values. It has following structure:

- "rounded" number in big if: 30, 60, 90, etc...
- Last "little" if has rounded number: 30, 60, 90 etc...
- First "little" if starts with numbers: 32, 62, 92 etc...

For example:

```
} else if (secondsLate <= 90) {
    if (secondsLate <= 62) {
        return _scaleReward(beans, 1_853_212);
    }
    ...
    if (secondsLate <= 90) {
        return _scaleReward(beans, 2_448_633);
    }
} else if (secondsLate <= 120) {
```

However structure is broken on the range `[240; 270]`.

Recommendation:

```
- } else if (secondsLate <= 238) {
+ } else if (secondsLate <= 240) {
```

```
      if (secondsLate <= 238) {
          return _scaleReward(beans, 10_677_927);
      }
+     if (secondsLate <= 240) {
+         return _scaleReward(beans, 10_892_553);
+     }
  } else if (secondsLate <= 270) {
-     if (secondsLate <= 240) {
-         return _scaleReward(beans, 10_892_553);
-     }
```

Pinto: Resolved in PR 93.

Cantina Managed: Fix verified.

3.3.7 Outdated Natspec in `LibFlood.handleRain()`

Severity: Informational

Context: LibFlood.sol#L85

Description: PI-3 updates value from 5% to 3%:

> Lower the Excessively Low Pod Rate threshold from 5% to 3%.

Recommendation:

```
- // cases % 36  3-8 represent the case where the pod rate is less than 5% and P > 1.
+ // cases % 36  3-8 represent the case where the pod rate is less than 3% and P > 1.
```

Pinto: Resolved in PR 91.

Cantina Managed: Fix verified.

3.3.8 `LibSilo.checkForEarnedBeans()` is not used anymore

Severity: Informational

Context: LibSilo.sol#L705-L717

Description: Function `LibSilo.checkForEarnedBeans()` was previously used to determine whether deposit has germinating earned beans to account it correctly during transfer and withdrawal. However now earned beans are no more germinating, and this function is not used anywhere.

Recommendation: Consider removing it.

Pinto: Resolved in PR 90.

Cantina Managed: Fix verified.

3.3.9 Array length access in Assembly can be simplified

Severity: Informational

Context: LibEvaluate.sol#L361, LibGauge.sol#L249, LibUsdOracle.sol#L184, LibTokenSilo.sol#L416

Description: There are multiple places across codebase where you perform extra add 0 while accessing array length. For example:

```
assembly {
    tokenPrice := mload(add(data, add(0x20, 0)))
}
```

It doesn't do anything, better to keep code clean.

Recommendation: Use `mload(add(<array>, 0x20))` instead in all referred places.

Pinto: Resolved in PR 89.

Cantina Managed: Fix verified.

3.3.10 Incorrect Natspec in `LibTokenSilo.incrementTotalDeposited()`

Severity: Informational

Context: LibTokenSilo.sol#L180

Description: Simple typo, current way it doesn't make sense:

```
/**
    * @dev Increment the total amount and bdv of `token` deposited in the Silo.
    * @dev `IncrementTotalDeposited` should be used when removing deposits that are // <<<
    * >= 2 seasons old (ex. when a user converts).
    */
function incrementTotalDeposited(address token, uint256 amount, uint256 bdv) internal {
    AppStorage storage s = LibAppStorage.diamondStorage();
    s.sys.silo.balances[token].deposited = s.sys.silo.balances[token].deposited.add(
        amount.toUint128()
    );
    s.sys.silo.balances[token].depositedBdv = s.sys.silo.balances[token].depositedBdv.add(
        bdv.toUint128()
    );
}
```

Recommendation:

```
- * @dev `IncrementTotalDeposited` should be used when removing deposits that are
+ * @dev `IncrementTotalDeposited` should be used when adding deposits that are
```

Pinto: Resolved in PR 88.

Cantina Managed: Fix verified.

3.3.11 Legacy code block can be removed from `LibFlood.calculateSopPerWell()`

Severity: Informational

Context: LibFlood.sol#L455-L460

Description: You can see that condition of 2nd `if` exists in 1st `if`. So 2nd block is never executed:

```
if (totalPositiveDeltaB < totalNegativeDeltaB || positiveDeltaBCount == 0) { // <<<
    // The less than conditional can occur if the twaDeltaB is positive, but the instanteous deltaB is negative
    ↪  or 0
    // In that case, no reductions are needed.
    // If there are no positive values, no well flooding is needed, return zeros
    for (uint256 i = 0; i < positiveDeltaBCount; i++) {
        wellDeltaBs[i].deltaB = 0;
    }
    return wellDeltaBs;
}

if (totalPositiveDeltaB < totalNegativeDeltaB) { // <<<
    for (uint256 i = 0; i < positiveDeltaBCount; i++) {
        wellDeltaBs[i].deltaB = 0;
    }
    return wellDeltaBs;
}
```

Recommendation: Consider removing it.

Pinto: Resolved in PR 87.

Cantina Managed: Fix verified.

3.3.12 Outdated Natspec in `LibDibbler.morningTemperature()`

Severity: Informational

Context: LibDibbler.sol#L247-L255

Description: This description is stale from the times Beanstalk was migrated to Arbitrum. It describes how L2 blocks are processed to L1 blocks and so on, but that doesn't reflect intention of why logic is following, and makes things hard to understand.

```
/**
    * @dev Returns the temperature `s.sys.weather.temp` scaled down based on the block delta.
    * Precision level 1e6, as soil has 1e6 precision (1% = 1e6)
    * the formula `log3.5(A * MAX_BLOCK_ELAPSED + 1)` is applied, where:
    * `A = 0.1`
    * `MAX_BLOCK_ELAPSED = 25`
    * @dev L2 block times are signifncatly shorter than L1. To adjust for this,
    * `delta` is scaled down by the ratio of L2 block time to L1 block time.
    */
function morningTemperature() internal view returns (uint256) {
    AppStorage storage s = LibAppStorage.diamondStorage();
    uint256 delta = block
        .number
        .sub(s.sys.season.sunriseBlock)
        .mul(L2_BLOCK_TIME)
        .div(L1_BLOCK_TIME)
        .div(2); // dividing by 2 increases the morning auction time from 25 to 50 L1 blocks (5 min -> 10 min)
```

Recommendation: Make description consistent with actual logic. Basically there are 25 chunks of 24 seconds each - they form 10 minutes. And applied formula is `log3.5(0.1 * CHUNKS_ELAPSED + 1)`.

Pinto: Resolved in PR 86.

Cantina Managed: Fix verified.

3.3.13 Incorrect Natspec in `LibWell.getTwaReservesForWell()`

Severity: Informational

Context: LibWell.sol#L305-L308

Description: Actually it reads `twaReserves` and not prices:

```
/**
    * @notice Returns the TKN / USD price stored in {AppStorage.usdTokenPrice}.
    * @dev assumes TKN has 18 decimals.
    */
function getTwaReservesForWell(
    address well
) internal view returns (uint256[] memory twaReserves) {
    AppStorage storage s = LibAppStorage.diamondStorage();
    twaReserves = new uint256[](2);
    twaReserves[0] = s.sys.twaReserves[well].reserve0;
    twaReserves[1] = s.sys.twaReserves[well].reserve1;
}
```

Recommendation: Update it.

Pinto: Resolved in PR 92.

Cantina Managed: Fix verified.

3.3.14 Incorrect Natspec for `LibWell.getUsdTokenPriceForWell()`

Severity: Informational

Context: LibWell.sol#L228-L231

Description: Actually it returns TKN / USD price. For example if WBTC = 80_000 USD, then it will return `1 / 80_000 * 1e8`.

```
/**
    * @notice Returns the USD / TKN price stored in {AppStorage.usdTokenPrice}.
    * @dev assumes TKN has 18 decimals.
    */
function getUsdTokenPriceForWell(address well) internal view returns (uint tokenUsd) {
    tokenUsd = LibAppStorage.diamondStorage().sys.usdTokenPrice[well];
}
```

That's because price cached to `sys.usdTokenPrice[well]` was quoted via `LibUsdOracle.getUsdPrice()`.

Recommendation: Update Natspec.

Pinto: Resolved in PR 92.

Cantina Managed: Fix verified.

3.3.15 Deprecated version of `PRBMath` is used for `LibPRBMathRoundable`

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description:

```
"@prb/math": "v2.5.0"
```

`LibPRBMathRoundable` uses `PRBMath` v.25 which is no longer maintained. It was originally used with lower compiler versions and project should check the compatibility thoroughly for used functions.

Recommendation: Try to upgrade to newer version of the library if possible.

Pinto: Acknowledged, at first glance a number of functions would need to be resolved, and we are very thorough in our analysis of the compatibility, and thus think this is ok.

Cantina Managed: Acknowledged.