# Beanstalk

## Smart Contract Security Audit

Prepared by: **Halborn**

Date of Engagement: **May 9th, 2022 - July 1st, 2022**

Visit: **Halborn.com**

## DOCUMENT REVISION HISTORY

| VERSION | MODIFICATION | DATE | AUTHOR |
|---|---|---|---|
| 0.1 | Document Creation | 05/09/2022 | Roberto Reigada |
| 0.2 | Document Updates | 06/30/2022 | Roberto Reigada |
| 0.3 | Draft Review | 07/01/2022 | Gabi Urrutia |
| 1.0 | Remediation Plan | 07/11/2022 | Roberto Reigada |
| 1.1 | Remediation Plan Review | 07/11/2022 | Gabi Urrutia |

## CONTACTS

| CONTACT | COMPANY | EMAIL |
|---|---|---|
| Rob Behnke | Halborn | Rob.Behnke@halborn.com |
| Steven Walbroehl | Halborn | Steven.Walbroehl@halborn.com |
| Gabi Urrutia | Halborn | Gabi.Urrutia@halborn.com |
| Roberto Reigada | Halborn | Roberto.Reigada@halborn.com |

# EXECUTIVE OVERVIEW

## 1.1 INTRODUCTION

Beanstalk engaged Halborn to conduct a security audit on their smart contracts beginning on May 9th, 2022 and ending on June 30th, 2022. The security assessment was scoped to the smart contracts provided in the GitHub repository BeanstalkFarms/Beanstalk.

## 1.2 AUDIT SUMMARY

The team at Halborn was provided seven weeks for the engagement and assigned a full-time security engineer to audit the security of the smart contract. The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this audit is to:

- Ensure that smart contract functions operate as intended
- Identify potential security issues with the smart contracts

In summary, Halborn identified some security risks that were mostly addressed by the Beanstalk team.

## 1.3 TEST APPROACH & METHODOLOGY

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this audit. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the audit:

- Research into architecture and purpose
- Smart contract manual code review and walkthrough
- Graphing out functionality and contract logic/connectivity/functions (solgraph)
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes
- Manual testing by custom scripts
- Scanning of solidity files for vulnerabilities, security hot-spots or bugs. (MythX)
- Static Analysis of security for scoped contract, and imported functions. (Slither)
- Testnet deployment (Brownie, Remix IDE)

## RISK METHODOLOGY:

Vulnerabilities or issues observed by Halborn are ranked based on the risk assessment methodology by measuring the **LIKELIHOOD** of a security incident and the **IMPACT** should an incident occur. This framework works for communicating the characteristics and impacts of technology vulnerabilities. The quantitative model ensures repeatable and accurate measurement while enabling users to see the underlying vulnerability characteristics that were used to generate the Risk scores. For every vulnerability, a risk level will be calculated on a scale of 5 to 1 with 5 being the highest likelihood or impact.

### RISK SCALE - LIKELIHOOD

5 - Almost certain an incident will occur.
4 - High probability of an incident occurring.
3 - Potential of a security incident in the long term.
2 - Low probability of an incident occurring.
1 - Very unlikely issue will cause an incident.

### RISK SCALE - IMPACT

5 - May cause devastating and unrecoverable impact or loss.
4 - May cause a significant level of impact or loss.

3 - May cause a partial impact or loss to many.
2 - May cause temporary impact or loss.
1 - May cause minimal or un-noticeable impact.

The risk level is then calculated using a sum of these two values, creating a value of 10 to 1 with 10 being the highest level of security risk.

| | | | | |
|---|---|---|---|---|
| CRITICAL | HIGH | MEDIUM | LOW | INFORMATIONAL |

**10** - CRITICAL
**9 - 8** - HIGH
**7 - 6** - MEDIUM
**5 - 4** - LOW
**3 - 1** - VERY LOW AND INFORMATIONAL

## 1.4 SCOPE

IN-SCOPE:
The security assessment was scoped to the following smart contracts:

- MarketplaceFacet.sol
- SeasonFacet.sol
- SiloFacet.sol
- WhitelistFacet.sol
- UnripeFacet.sol
- TokenFacet.sol
- PauseFacet.sol
- OwnershipFacet.sol
- FieldFacet.sol
- FertilizerFacet.sol: Added in Commit ID 2
- FarmFacet.sol
- DiamondLoupeFacet.sol
- DiamondCutFacet.sol
- CurveFacet.sol
- ConvertFacet.sol
- BDVFacet.sol
- FundraiserFacet.sol
- AppStorage.sol
- Diamond.sol
- Bean.sol
- GhostERC20.sol
- Sprout.sol

Commit ID 1:
- 17be0bbf1a17688978dfa551cbfee30d9a200f3e

Commit ID 2:
- 7866e870d4d97f22cc4b92730d5532168edb114c

Changes from Commit ID 1:
BDVFacet:

- Changed the name of a reference to a library for Unripe Beans + Unripe LP.

BarnRaiseFacet:

- Deleted in exchange for Fertilizer Facet.

ConvertFacet:

- Changed BDV of the output of Convert to be the maximum of the BDV of assets being converted from to the BDV of the assets being converted to.
- Combined beanToLP and lpToBean into getAmountOut (View functions).

CurveFacet:

- Fixed HAL-01 issue.

FarmFacet:

- Added a state variable named isFarm. This is set 1 upon deployment (1 = not farm, 2 = farm). Farm is set to 2 when a farm function starts and 1 when it ends. The wrapEth function, and in the future other functions that use Ether, now have a refund operation that checks if the function is a farm function or not. If not, it refunds the Ether. If it is, it doesn't refund the Ether and the farm function returns the Ether at the end of the transaction.

FertilizerFacet:

- Created in accordance with BFP-72

SeasonFacet:

- In accordance with BFP-72, distribute 1/3 Beans mints to those who hold Fertilizer instead of those who hold the Barn Raise tokens.
- Changed Soil based on caseId when p > 1. -> If case < 8, multiple by constant < 1. When case >= 24, multiple by constant > 1.

SiloFacet:

- Added function to update BDV of Unripe token Deposit in accordance with BFP-72.

TokenFacet:

- Added refund option when wrapping Eth.

UnripeFacet:

- Updated Unripe Tokens in association with BFP-72

Fertilizer:

- Added Fertilizer token

Fixed Commit ID:

- 1447fa2c0d42c73345a38edb4f4dad076392f429

## 2. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

| CRITICAL | HIGH | MEDIUM | LOW | INFORMATIONAL |
|---|---|---|---|---|
| 2 | 0 | 2 | 4 | 9 |

LIKELIHOOD

IMPACT

| | | | | |
|---|---|---|---|---|
| 🟨 | 🟨 | 🟥 | 🟥 | 🟪 (HAL-01)<br>(HAL-02) |
| 🟩 | 🟨 | 🟨 | 🟥 | 🟥 |
| 🟩 (HAL-06)<br>(HAL-07)<br>(HAL-08) | 🟩 | 🟨 (HAL-03)<br>(HAL-04) | 🟨 | 🟥 |
| ⬜ | 🟩 | 🟩 (HAL-05) | 🟨 | 🟨 |
| ⬜ (HAL-09)<br>(HAL-10)<br>(HAL-11)<br>(HAL-12)<br>(HAL-13)<br>(HAL-14)<br>(HAL-15)<br>(HAL-16)<br>(HAL-17) | ⬜ | 🟩 | 🟩 | 🟨 |

| SECURITY ANALYSIS | RISK LEVEL | REMEDIATION DATE |
|---|---|---|
| HAL01 - INTERNAL BALANCE TOKENS CAN BE DRAINED THROUGH THE CURVEFACET.EXCHANGEUNDERLYING FUNCTION | Critical | SOLVED - 07/11/2022 |
| HAL02 - USDC OF THE INTERNAL BALANCE CAN BE DRAINED BY ANY USER THROUGH THE FERTILIZERFACET.MINTFERTILIZER FUNCTION | Critical | SOLVED - 07/11/2022 |
| HAL03 - INCONSISTENT INTERNAL BALANCES WHEN SUPPLYING TRANSFER-ON-FEE OR DEFLATIONARY TOKENS | Medium | SOLVED - 07/11/2022 |
| HAL04 - UNLIMITED FERTILIZER CAN BE BOUGHT THROUGH THE FERTILIZERFACET.MINTFERTILIZER FUNCTION | Medium | SOLVED - 07/11/2022 |
| HAL05 - ACTIVE FERTILIZER WILL BE CLAIMED AUTOMATICALLY BY THE SENDER DURING A SAFETRANSFERFROM CALL | Low | RISK ACCEPTED |
| HAL06 - SEASONFACET.INCENTIVIZE EXPONENTIAL INCENTIVE LOGIC IS NOT WORKING | Low | SOLVED - 07/11/2022 |
| HAL07 - MISSING REQUIRE CHECK IN TOKENFACET.WRAPETH FUNCTION | Low | SOLVED - 07/11/2022 |
| HAL08 - MULTIPLE OVERFLOWS IN MARKETPLACE FACET | Low | SOLVED - 07/11/2022 |
| HAL09 - FERTILIZERPREMINT.BUYANDMINT FUNCTION COULD BE SANDWICHED | Informational | SOLVED - 07/11/2022 |
| HAL10 - POD PRICE IS LIMITED TO 16.7 BEANS | Informational | SOLVED - 07/11/2022 |
| HAL11 - FARMFACET: USE OF DELEGATECALL IN A FOR LOOP | Informational | SOLVED - 07/11/2022 |
| HAL12 - CRITICAL DEPENDENCY ON CURVE METAPOOL FACTORIES | Informational | ACKNOWLEDGED |
| HAL13 - SAFETRANSFER IS NOT USED FOR ALL THE TOKEN TRANSFERS | Informational | SOLVED - 07/11/2022 |

| | | |
|---|---|---|
| HAL14 - REQUIRE STATEMENT TYPOS | Informational | SOLVED - 07/11/2022 |
| HAL15 - INITIALIZE FUNCTION IN FERTILIZER CONTRACT CAN BE REMOVED | Informational | SOLVED - 07/11/2022 |
| HAL16 - UNNEEDED INITIALIZATION OF UINT256 VARIABLES TO 0 | Informational | SOLVED - 07/11/2022 |
| HAL17 - USING POSTFIX OPERATORS IN LOOPS | Informational | SOLVED - 07/11/2022 |

# FINDINGS & TECH DETAILS

## 3.1 (HAL-01) INTERNAL BALANCE TOKENS CAN BE DRAINED THROUGH THE CURVEFACET.EXCHANGEUNDERLYING FUNCTION - CRITICAL

Description:

In the CurveFacet, the exchangeUnderlying() function is used to swap underlying assets from different Curve stable pools:

Listing 1: CurveFacet.sol (Lines 70,72,76,77)

```
66 function exchangeUnderlying(
67     address pool,
68     address fromToken,
69     address toToken,
70     uint256 amountIn,
71     uint256 minAmountOut,
72     LibTransfer.From fromMode,
73     LibTransfer.To toMode
74 ) external payable nonReentrant {
75     (int128 i, int128 j) = getUnderlyingIandJ(fromToken, toToken, pool);
76     IERC20(fromToken).receiveToken(amountIn, msg.sender, fromMode);
77     IERC20(fromToken).approveToken(pool, amountIn);
78 
79     if (toMode == LibTransfer.To.EXTERNAL) {
80         ICurvePoolR(pool).exchange_underlying(
81             i,
82             j,
83             amountIn,
84             minAmountOut,
85             msg.sender
86         );
87     } else {
88         uint256 amountOut = ICurvePool(pool).exchange_underlying(
89             i,
90             j,
91             amountIn,
      minAmountOut
    );
    msg.sender.increaseInternalBalance(IERC20(toToken), amountOut);
  }
}
```

The LibTransfer.From fromMode has 4 different modes:

- EXTERNAL
- INTERNAL
- EXTERNAL_INTERNAL
- INTERNAL_TOLERANT

With the INTERNAL_TOLERANT fromMode tokens will be collected from the user's Internal Balance and the transaction will not fail if there is not enough tokens there.

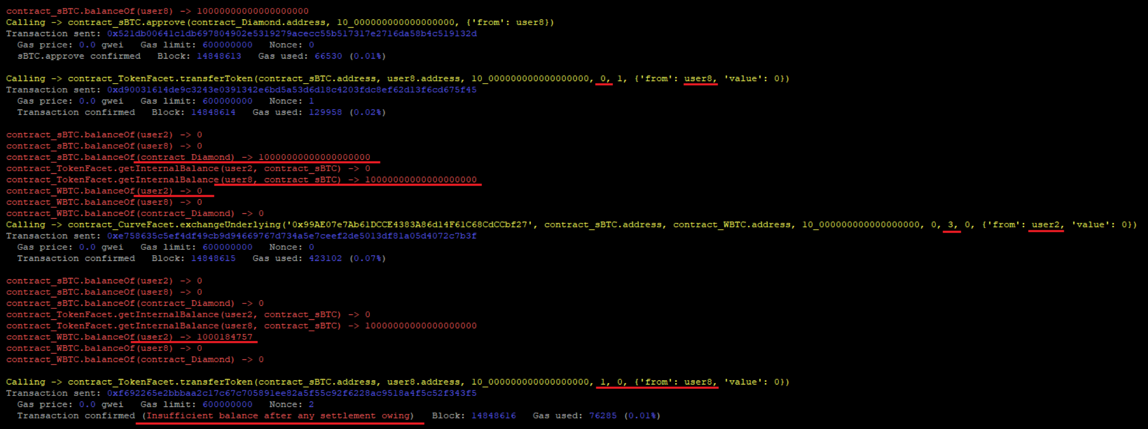
As in the receiveToken() call, users can use the INTERNAL_TOLERANT fromMode and the value returned by receiveToken() is not checked users can abuse this and swap tokens that belong to other users (tokens that are part of other users' internal balance).

Proof of Concept:

Pool: 0x99AE07e7Ab61DCCE4383A86d14F61C68CdCCbf27
Underlying WBTC: 0x2260FAC5E5542a773Aa44fBCfeDf7C193bc2C599
Underlying sBTC: 0xfE18be6b3Bd88A2D2A7f928d00292E7a9963CfC6

1. User8 transfers 10_000000000000000000 sBTC tokens to his internal balance.
2. User2 calls exchangeUnderlying() with an INTERNAL_TOLERANT fromMode, setting as the amountIn 10_000000000000000000 and as fromToken the sBTC token address. These sBTC tokens do belong to user8.
3. User2 successfully swaps for free the sBTC for the WBTC tokens, getting 10_00184757 WBTC in his external balance.

4. Now User8 tries to withdraw from his internal balance the 10_000000000000000000 sBTC tokens he had deposited previously, but the transactions fails as the contract does not have those tokens anymore. They were swapped and stolen by user2.



```
contract_sBTC.balanceOf(user8) -> 10000000000000000000
Calling -> contract_sBTC.approve(contract_Diamond.address, 10_000000000000000000, {'from': user8})
Transaction sent: 0x521db00641c1db697804902e5319279acecc55b517317e2716da58b4c519132d
  Gas price: 0.0 gwei   Gas limit: 600000000   Nonce: 0
  sBTC.approve confirmed   Block: 14848613   Gas used: 66530 (0.01%)

Calling -> contract_TokenFacet.transferToken(contract_sBTC.address, user8.address, 10_000000000000000000, 0, 1, {'from': user8, 'value': 0})
Transaction sent: 0xd90031614de9c3243e0391342e6bd5a53d6d18c4203fdc8ef62d13f6cd675f45
  Gas price: 0.0 gwei   Gas limit: 600000000   Nonce: 1
  Transaction confirmed   Block: 14848614   Gas used: 129958 (0.02%)

contract_sBTC.balanceOf(user2) -> 0
contract_sBTC.balanceOf(user8) -> 0
contract_sBTC.balanceOf(contract_Diamond) -> 10000000000000000000
contract_TokenFacet.getInternalBalance(user2, contract_sBTC) -> 0
contract_TokenFacet.getInternalBalance(user8, contract_sBTC) -> 10000000000000000000
contract_WBTC.balanceOf(user2) -> 0
contract_WBTC.balanceOf(user8) -> 0
contract_WBTC.balanceOf(contract_Diamond) -> 0
Calling -> contract_CurveFacet.exchangeUnderlying('0x99AE07e7Ab61DCCE4383A86d14F61C68CdCCbf27', contract_sBTC.address, contract_WBTC.address, 10_000000000000000000, 0, 3, 0, {'from': user2, 'value': 0})
Transaction sent: 0xe758635c5ef4df49cb9d94669767d734a5e7ceef2de5013df81a05d4072c7b3f
  Gas price: 0.0 gwei   Gas limit: 600000000   Nonce: 0
  Transaction confirmed   Block: 14848615   Gas used: 423102 (0.07%)

contract_sBTC.balanceOf(user2) -> 0
contract_sBTC.balanceOf(user8) -> 0
contract_sBTC.balanceOf(contract_Diamond) -> 0
contract_TokenFacet.getInternalBalance(user2, contract_sBTC) -> 0
contract_TokenFacet.getInternalBalance(user8, contract_sBTC) -> 10000000000000000000
contract_WBTC.balanceOf(user2) -> 1000184757
contract_WBTC.balanceOf(user8) -> 0
contract_WBTC.balanceOf(contract_Diamond) -> 0

Calling -> contract_TokenFacet.transferToken(contract_sBTC.address, user8.address, 10_000000000000000000, 1, 0, {'from': user8, 'value': 0})
Transaction sent: 0xf692265e2bbbaa2c17c67c705891ee82a5f55c92f6228ac9518a4f5c52f343f5
  Gas price: 0.0 gwei   Gas limit: 600000000   Nonce: 2
  Transaction confirmed (Insufficient balance after any settlement owing)   Block: 14848616   Gas used: 76285 (0.01%)
```

Risk Level:

**Likelihood - 5**
**Impact - 5**

Recommendation:

It is recommended to save the return value of the receiveToken() call and overwrite the amountIn variable with that return as shown below:

Listing 2: CurveFacet.sol (Line 76)

```
function exchangeUnderlying(
    address pool,
    address fromToken,
    address toToken,
    uint256 amountIn,
    uint256 minAmountOut,
    LibTransfer.From fromMode,
    LibTransfer.To toMode
) external payable nonReentrant {
    (int128 i, int128 j) = getUnderlyingIandJ(fromToken, toToken, pool);
    amountIn = IERC20(fromToken).receiveToken(amountIn, msg.sender, fromMode);
    IERC20(fromToken).approveToken(pool, amountIn);

    if (toMode == LibTransfer.To.EXTERNAL) {
        ICurvePoolR(pool).exchange_underlying(
            i,
            j,
            amountIn,
            minAmountOut,
            msg.sender
        );
    } else {
        uint256 amountOut = ICurvePool(pool).exchange_underlying(
            i,
            j,
            amountIn,
            minAmountOut
        );
        msg.sender.increaseInternalBalance(IERC20(toToken), amountOut);
    }
}
```

### Remediation Plan:

**SOLVED**: The Beanstalk team corrected the issue by overwritting amountIn with the value returned from the receiveToken() call, as suggested.

## 3.2 (HAL-02) USDC OF THE INTERNAL BALANCE CAN BE DRAINED BY ANY USER THROUGH THE FERTILIZERFACET.MINTFERTILIZER FUNCTION - CRITICAL

Description:

In the FertilizerFacet, the mintFertilizer() function is used to buy Fertilizer in exchange for USDC:

**Listing 3: FertilizerFacet.sol (Lines 43-48)**

```
35 function mintFertilizer(
36     uint128 amount,
37     uint256 minLP,
38     LibTransfer.From mode
39 ) external payable {
40     uint256 remaining = LibFertilizer.remainingRecapitalization();
41     uint256 _amount = uint256(amount);
42     if (_amount > remaining) _amount = remaining;
43     LibTransfer.receiveToken(
44         C.usdc(),
45         uint256(amount).mul(1e6),
46         msg.sender,
47         mode
48     );
49     uint128 id = LibFertilizer.addFertilizer(
50         uint128(s.season.current),
51         amount,
52         minLP
53     );
54     C.fertilizer().beanstalkMint(msg.sender, uint256(id), amount,
   ↳ s.bpf);
55 }
```

This function has the same issue that was described in HAL01 - INTERNAL BALANCE TOKENS CAN BE DRAINED THROUGH THE CURVEFACET.EXCHANGEUNDERLYING

FUNCTION as the value returned by receiveToken() is not checked, users can abuse this and buy Fertilizer with the USDC of other users internal balance through the INTERNAL_TOLERANT fromMode.

Risk Level:

**Likelihood - 5**
**Impact - 5**

Recommendation:

It is recommended to save the return value of the receiveToken() call and overwrite the _amount variable with that return as shown below:

**Listing 4: FertilizerFacet.sol (Line 43)**

```
35 function mintFertilizer(
36     uint128 amount,
37     uint256 minLP,
38     LibTransfer.From mode
39 ) external payable {
40     uint256 remaining = LibFertilizer.remainingRecapitalization();
41     uint256 _amount = uint256(amount);
42     if (_amount > remaining) _amount = remaining;
43     _amount = LibTransfer.receiveToken(
44         C.usdc(),
45         uint256(_amount).mul(1e6),
46         msg.sender,
47         mode
48     );
49     uint128 id = LibFertilizer.addFertilizer(
50         uint128(s.season.current),
51         uint128(_amount),
52         minLP
53     );
54     C.fertilizer().beanstalkMint(msg.sender, uint256(id), amount,
   ↳ s.bpf);
55 }
```

Remediation Plan:

**SOLVED**: The Beanstalk team corrected the issue by considering the returned value of the receiveToken() call:

Listing 5: FertilizerFacet.sol (Line 42)

```
35 function mintFertilizer(
36     uint128 amount,
37     uint256 minLP,
38     LibTransfer.From mode
39 ) external payable {
40     uint128 remaining = uint128(LibFertilizer.remainingRecapitalization()); // remaining <= 77_000_000 so downcasting is safe.
41     if (amount > remaining) amount = remaining;
42     amount = uint128(LibTransfer.receiveToken(
43         C.usdc(),
44         uint256(amount).mul(1e6),
45         msg.sender,
46         mode
47     ).div(1e6)); // return value <= amount, so downcasting is safe.
48     uint128 id = LibFertilizer.addFertilizer(
49         uint128(s.season.current),
50         amount,
51         minLP
52     );
53     C.fertilizer().beanstalkMint(msg.sender, uint256(id), amount, s.bpf);
54 }
```

## 3.3 (HAL-03) INCONSISTENT INTERNAL BALANCES WHEN SUPPLYING TRANSFER-ON-FEE OR DEFLATIONARY TOKENS - MEDIUM

Description:

In the library LibTransfer, used by the TokenFacet contract, the transferToken() function assume that the amount of token is transferred to the smart contract after calling token.safeTransferFrom(sender, address(this), amount - receivedAmount); (and thus it updates the states variables accordingly). For example:

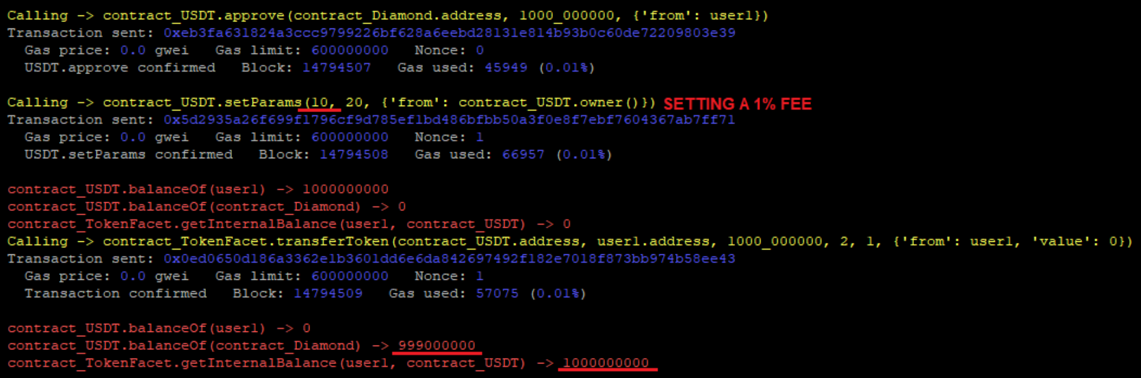
Listing 6: LibTransfer.sol (Lines 37,38,74)

```
29 function transferToken(
30     IERC20 token,
31     address recipient,
32     uint256 amount,
33     From fromMode,
34     To toMode
35 ) internal returns (uint256 transferredAmount) {
36     if (fromMode == From.EXTERNAL && toMode == To.EXTERNAL) {
37         token.transferFrom(msg.sender, recipient, amount);
38         return amount;
39     }
40     amount = receiveToken(token, amount, msg.sender, fromMode);
41     sendToken(token, amount, recipient, toMode);
42     return amount;
43 }
44
45 function receiveToken(
46     IERC20 token,
47     uint256 amount,
48     address sender,
49     From mode
50 ) internal returns (uint256 receivedAmount) {
51     if (amount == 0) return 0;
52     if (mode != From.EXTERNAL) {
        receivedAmount = LibBalance.decreaseInternalBalance(
            sender,
            token,
            amount,
            mode != From.INTERNAL
        );
        if (amount == receivedAmount || mode == From.INTERNAL_TOLERANT)
            return receivedAmount;
    }
    token.safeTransferFrom(sender, address(this), amount - receivedAmount);
    return amount;
}

function sendToken(
    IERC20 token,
    uint256 amount,
    address recipient,
    To mode
) internal {
    if (amount == 0) return;
    if (mode == To.INTERNAL)
        LibBalance.increaseInternalBalance(recipient, token, amount);
    else token.safeTransfer(recipient, amount);
}
```

However, this may not be true if the token is a transfer-on-fee token or a deflationary/rebasing token, causing the received amount to be less than the accounted amount in the different state variables.

## Proof of Concept:



```
Calling -> contract_USDT.approve(contract_Diamond.address, 1000_000000, {'from': user1})
Transaction sent: 0xeb3fa631824a3ccc9799226bf628a6eebd28131e814b93b0c60de72209803e39
  Gas price: 0.0 gwei   Gas limit: 600000000   Nonce: 0
  USDT.approve confirmed   Block: 14794507   Gas used: 45949 (0.01%)

Calling -> contract_USDT.setParams(10, 20, {'from': contract_USDT.owner()}) SETTING A 1% FEE
Transaction sent: 0x5d2935a26f699f1796cf9d785ef1bd486bfbb50a3f0e8f7ebf7604367ab7ff71
  Gas price: 0.0 gwei   Gas limit: 600000000   Nonce: 1
  USDT.setParams confirmed   Block: 14794508   Gas used: 66957 (0.01%)

contract_USDT.balanceOf(user1) -> 1000000000
contract_USDT.balanceOf(contract_Diamond) -> 0
contract_TokenFacet.getInternalBalance(user1, contract_USDT) -> 0
Calling -> contract_TokenFacet.transferToken(contract_USDT.address, user1.address, 1000_000000, 2, 1, {'from': user1, 'value': 0})
Transaction sent: 0x0ed0650d186a3362e1b3601dd6e6da842697492f182e7018f873bb974b58ee43
  Gas price: 0.0 gwei   Gas limit: 600000000   Nonce: 1
  Transaction confirmed   Block: 14794509   Gas used: 57075 (0.01%)

contract_USDT.balanceOf(user1) -> 0
contract_USDT.balanceOf(contract_Diamond) -> 999000000
contract_TokenFacet.getInternalBalance(user1, contract_USDT) -> 1000000000
```

## Risk Level:

**Likelihood - 3**
**Impact - 3**

## Recommendation:

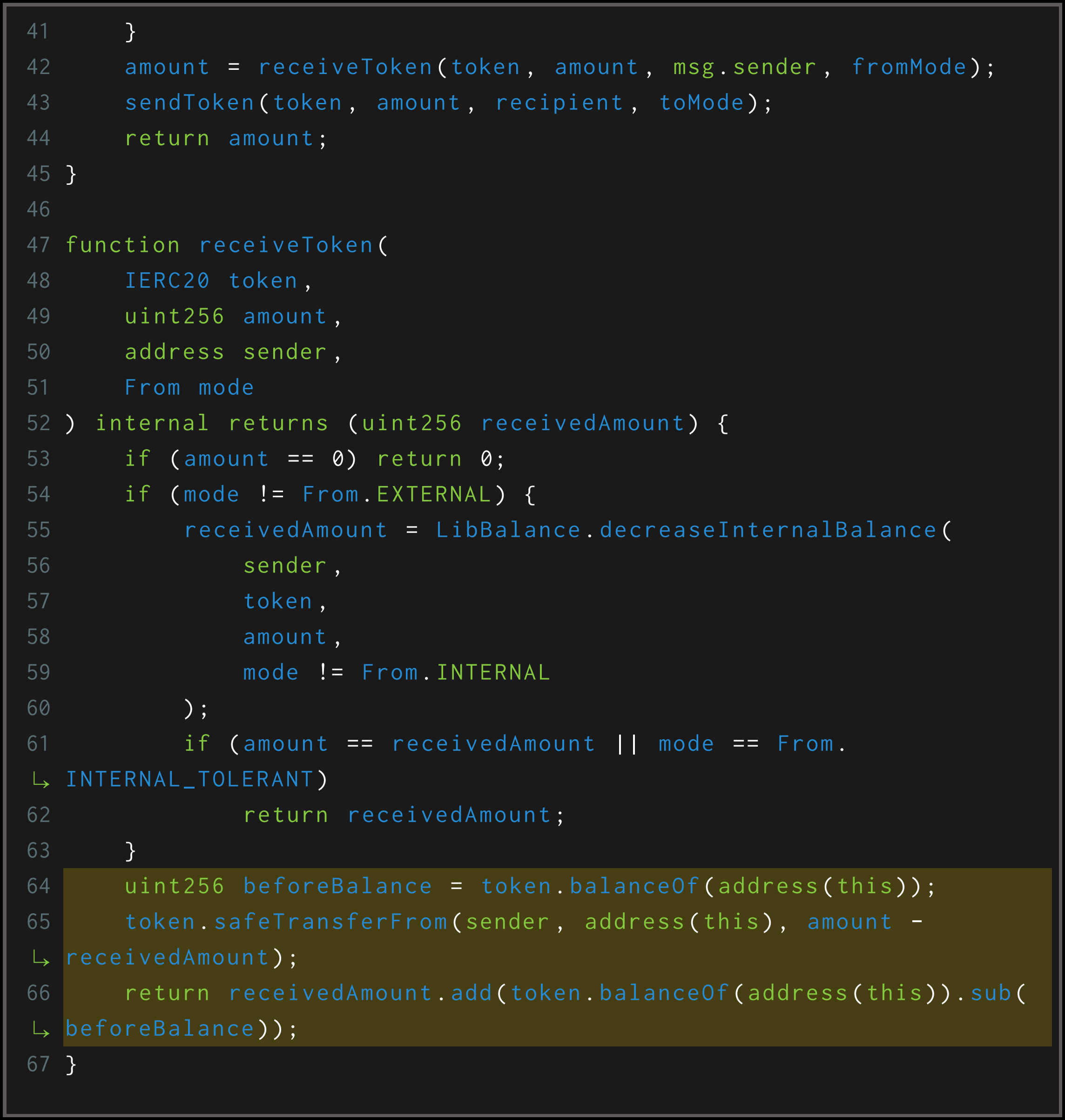
It is recommended to get the actual received token amount by calculating the difference of token balance before and after the transfer.

## Remediation Plan:

**SOLVED**: The Beanstalk team addressed the issue and now supports transfer-on-fee tokens:

Listing 7: LibTransfer.sol (Lines 38,39,40,64,65,66,)

```
30 function transferToken(
31     IERC20 token,
32     address recipient,
33     uint256 amount,
34     From fromMode,
35     To toMode
36 ) internal returns (uint256 transferredAmount) {
37     if (fromMode == From.EXTERNAL && toMode == To.EXTERNAL) {
38         uint256 beforeBalance = token.balanceOf(recipient);
39         token.safeTransferFrom(msg.sender, recipient, amount);
40         return token.balanceOf(recipient).sub(beforeBalance);
```



```
    }
    amount = receiveToken(token, amount, msg.sender, fromMode);
    sendToken(token, amount, recipient, toMode);
    return amount;
}

function receiveToken(
    IERC20 token,
    uint256 amount,
    address sender,
    From mode
) internal returns (uint256 receivedAmount) {
    if (amount == 0) return 0;
    if (mode != From.EXTERNAL) {
        receivedAmount = LibBalance.decreaseInternalBalance(
            sender,
            token,
            amount,
            mode != From.INTERNAL
        );
        if (amount == receivedAmount || mode == From.INTERNAL_TOLERANT)
            return receivedAmount;
    }
    uint256 beforeBalance = token.balanceOf(address(this));
    token.safeTransferFrom(sender, address(this), amount - receivedAmount);
    return receivedAmount.add(token.balanceOf(address(this)).sub(beforeBalance));
}
```

## 3.4 (HAL-04) UNLIMITED FERTILIZER CAN BE BOUGHT THROUGH THE FERTILIZERFACET.MINTFERTILIZER FUNCTION - MEDIUM

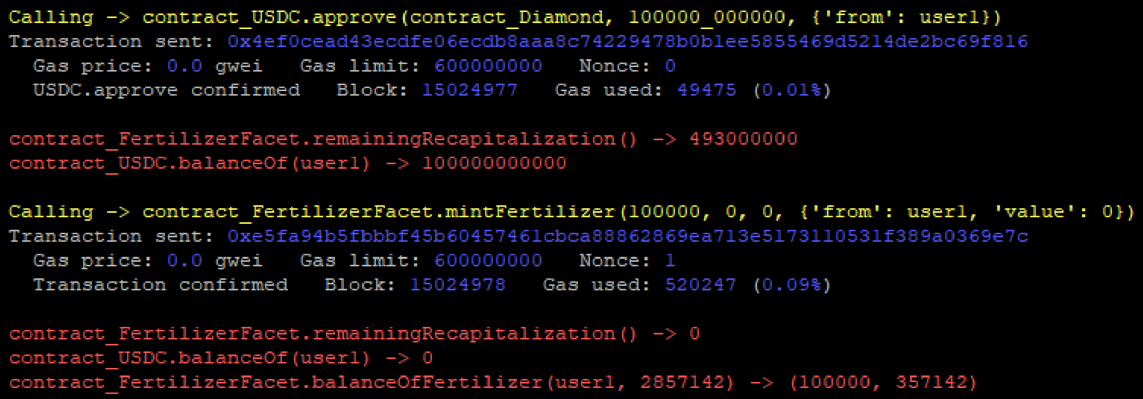
Description:

In the FertilizerFacet contract, the mintFertilizer() function checks if the amount provided by the user is higher than the remaining amount of Fertilizer and if that is the case, _amount is overwritten with the remaining Fertilizer preventing users to buy more Fertilizer than what is remaining:

Listing 8: FertilizerFacet.sol (Lines 42,45,51)

```
35 function mintFertilizer(
36     uint128 amount,
37     uint256 minLP,
38     LibTransfer.From mode
39 ) external payable {
40     uint256 remaining = LibFertilizer.remainingRecapitalization();
41     uint256 _amount = uint256(amount);
42     if (_amount > remaining) _amount = remaining;
43     LibTransfer.receiveToken(
44         C.usdc(),
45         uint256(amount).mul(1e6),
46         msg.sender,
47         mode
48     );
49     uint128 id = LibFertilizer.addFertilizer(
50         uint128(s.season.current),
51         amount,
52         minLP
53     );
54     C.fertilizer().beanstalkMint(msg.sender, uint256(id), amount,
   ↳ s.bpf);
55 }
```

Although, the contract wrongly uses the amount variable instead of _amount allowing users to mint more Fertilizer than what is remaining:



```
Calling -> contract_USDC.approve(contract_Diamond, 100000_000000, {'from': user1})
Transaction sent: 0x4ef0cead43ecdfe06ecdb8aaa8c74229478b0b1ee5855469d5214de2bc69f816
  Gas price: 0.0 gwei   Gas limit: 600000000   Nonce: 0
  USDC.approve confirmed   Block: 15024977   Gas used: 49475 (0.01%)

contract_FertilizerFacet.remainingRecapitalization() -> 493000000
contract_USDC.balanceOf(user1) -> 100000000000

Calling -> contract_FertilizerFacet.mintFertilizer(100000, 0, 0, {'from': user1, 'value': 0})
Transaction sent: 0xe5fa94b5fbbbf45b60457461cbca88862869ea713e5173110531f389a0369e7c
  Gas price: 0.0 gwei   Gas limit: 600000000   Nonce: 1
  Transaction confirmed   Block: 15024978   Gas used: 520247 (0.09%)

contract_FertilizerFacet.remainingRecapitalization() -> 0
contract_USDC.balanceOf(user1) -> 0
contract_FertilizerFacet.balanceOfFertilizer(user1, 2857142) -> (100000, 357142)
```

## Risk Level:

**Likelihood - 3**  
**Impact - 3**

## Recommendation:

It is recommended to use the _amount variable instead of amount for the receiveToken(), addFertilizer() and beanstalkMint() calls in the FertilizerFacet.mintFertilizer() function.

## Remediation Plan:

**SOLVED**: The Beanstalk team corrected the issue:

Listing 9: FertilizerFacet.sol (Line 41)

```
35 function mintFertilizer(
36     uint128 amount,
37     uint256 minLP,
38     LibTransfer.From mode
39 ) external payable {
40     uint128 remaining = uint128(LibFertilizer.
 ↳ remainingRecapitalization()); // remaining <= 77_000_000 so
 ↳ downcasting is safe.
41     if (amount > remaining) amount = remaining;
```

```
    amount = uint128(LibTransfer.receiveToken(
        C.usdc(),
        uint256(amount).mul(1e6),
        msg.sender,
        mode
    ).div(1e6)); // return value <= amount, so downcasting is safe.
    uint128 id = LibFertilizer.addFertilizer(
        uint128(s.season.current),
        amount,
        minLP
    );
    C.fertilizer().beanstalkMint(msg.sender, uint256(id), amount, s.bpf);
}
```

## 3.5 (HAL-05) ACTIVE FERTILIZER WILL BE CLAIMED AUTOMATICALLY BY THE SENDER DURING A SAFETRANSFERFROM CALL - LOW

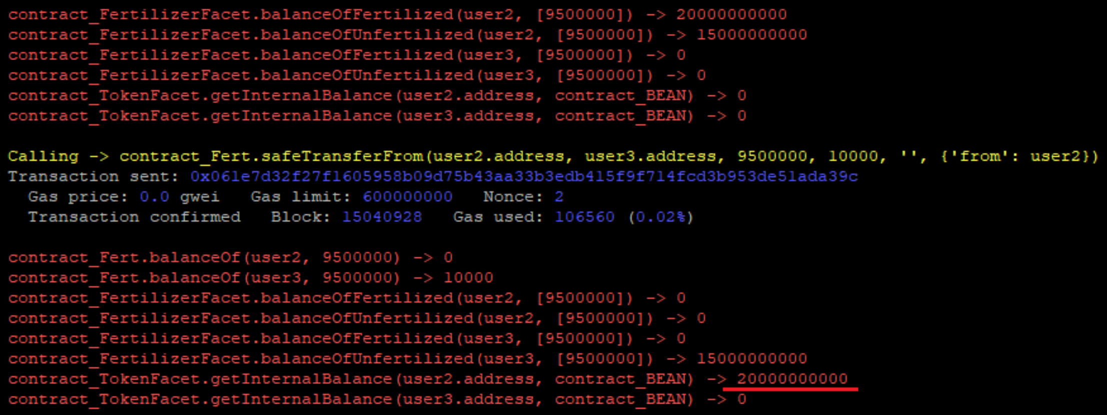
Description:

The Fertilizer contract contains the following _beforeTokenTransfer() hook:

**Listing 10: Fertilizer.sol (Lines 59,60)**

```
50 function _beforeTokenTransfer(
51     address, // operator,
52     address from,
53     address to,
54     uint256[] memory ids,
55     uint256[] memory, // amounts
56     bytes memory // data
57 ) internal virtual override {
58     uint256 bpf = uint256(IBS(owner()).beansPerFertilizer());
59     if (from != address(0)) _update(from, ids, bpf);
60     _update(to, ids, bpf);
61 }
```

This hook will be called with every safeTransferFrom() or safeBatchTransferFrom() call and will claim the fertilizer claimable amount automatically on behalf of the sender:

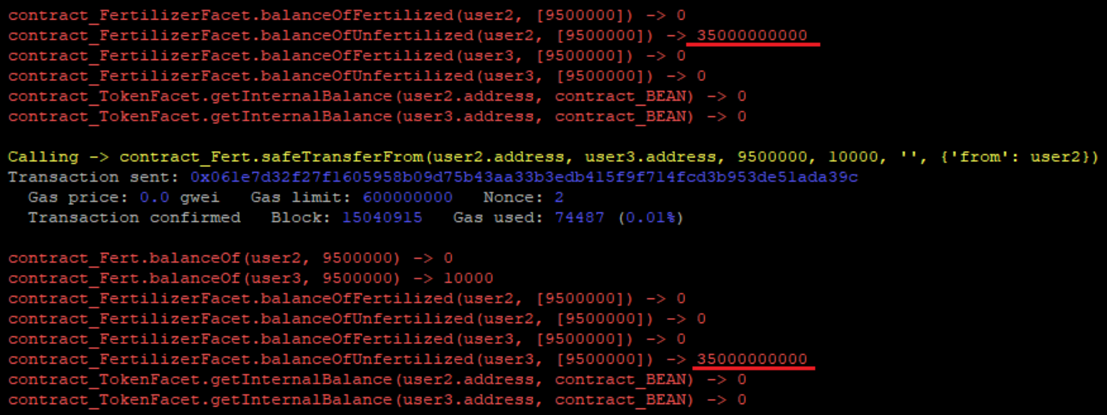


```
contract_FertilizerFacet.balanceOfFertilized(user2, [9500000]) -> 20000000000
contract_FertilizerFacet.balanceOfUnfertilized(user2, [9500000]) -> 15000000000
contract_FertilizerFacet.balanceOfFertilized(user3, [9500000]) -> 0
contract_FertilizerFacet.balanceOfUnfertilized(user3, [9500000]) -> 0
contract_TokenFacet.getInternalBalance(user2.address, contract_BEAN) -> 0
contract_TokenFacet.getInternalBalance(user3.address, contract_BEAN) -> 0

Calling -> contract_Fert.safeTransferFrom(user2.address, user3.address, 9500000, 10000, '', {'from': user2})
Transaction sent: 0x061e7d32f27f1605958b09d75b43aa33b3edb415f9f714fcd3b953de51ada39c
  Gas price: 0.0 gwei   Gas limit: 600000000   Nonce: 2
  Transaction confirmed   Block: 15040928   Gas used: 106560 (0.02%)

contract_Fert.balanceOf(user2, 9500000) -> 0
contract_Fert.balanceOf(user3, 9500000) -> 10000
contract_FertilizerFacet.balanceOfFertilized(user2, [9500000]) -> 0
contract_FertilizerFacet.balanceOfUnfertilized(user2, [9500000]) -> 0
contract_FertilizerFacet.balanceOfFertilized(user3, [9500000]) -> 0
contract_FertilizerFacet.balanceOfUnfertilized(user3, [9500000]) -> 15000000000
contract_TokenFacet.getInternalBalance(user2.address, contract_BEAN) -> 20000000000
contract_TokenFacet.getInternalBalance(user3.address, contract_BEAN) -> 0
```

If the amount of claimable fertilizer is zero, the receiver will get the full unfertilized amount as expected:



```
contract_FertilizerFacet.balanceOfFertilized(user2, [9500000]) -> 0
contract_FertilizerFacet.balanceOfUnfertilized(user2, [9500000]) -> 35000000000
contract_FertilizerFacet.balanceOfFertilized(user3, [9500000]) -> 0
contract_FertilizerFacet.balanceOfUnfertilized(user3, [9500000]) -> 0
contract_TokenFacet.getInternalBalance(user2.address, contract_BEAN) -> 0
contract_TokenFacet.getInternalBalance(user3.address, contract_BEAN) -> 0

Calling -> contract_Fert.safeTransferFrom(user2.address, user3.address, 9500000, 10000, '', {'from': user2})
Transaction sent: 0x061e7d32f27f1605958b09d75b43aa33b3edb415f9f714fcd3b953de51ada39c
  Gas price: 0.0 gwei   Gas limit: 600000000   Nonce: 2
  Transaction confirmed   Block: 15040915   Gas used: 74487 (0.01%)

contract_Fert.balanceOf(user2, 9500000) -> 0
contract_Fert.balanceOf(user3, 9500000) -> 10000
contract_FertilizerFacet.balanceOfFertilized(user2, [9500000]) -> 0
contract_FertilizerFacet.balanceOfUnfertilized(user2, [9500000]) -> 0
contract_FertilizerFacet.balanceOfFertilized(user3, [9500000]) -> 0
contract_FertilizerFacet.balanceOfUnfertilized(user3, [9500000]) -> 35000000000
contract_TokenFacet.getInternalBalance(user2.address, contract_BEAN) -> 0
contract_TokenFacet.getInternalBalance(user3.address, contract_BEAN) -> 0
```

This could allow the following scenario:

1. By making use of a third-party marketplace, user1 puts for sale his Fertilizer at a low price. That fertilizer id can be fully claimed at that time.
2. User2 buys the fertilizer planning to claim it afterwards and make some profit, but the fertilizer is claimed automatically on behalf of user1 during the safeTransferFrom() call and the user2 just receives an already claimed fertilizer.

Risk Level:

**Likelihood - 3**
**Impact - 2**

### Recommendation:

It is recommended to consider removing the _beforeTokenTransfer() hook so these claims are not done automatically, preventing the scenario mentioned.

### Remediation Plan:

**RISK ACCEPTED**: The Beanstalk team accepts this risk.

## 3.6 (HAL-06) SEASONFACET.INCENTIVIZE EXPONENTIAL INCENTIVE LOGIC IS NOT WORKING - LOW

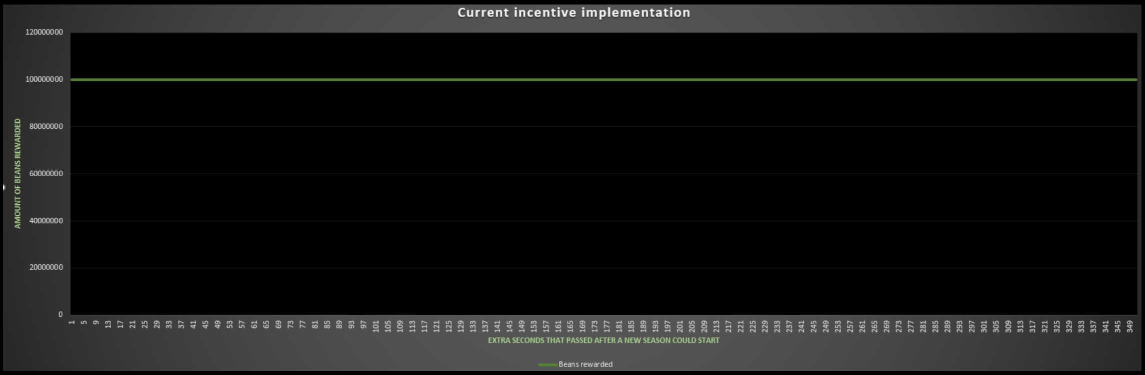
Description:

In the SeasonFacet contract, the incentivize() function is used to send some Beans to the user that successfully called sunrise() to start a new season:

Listing 11: SeasonFacet.sol (Lines 75,76)

```
70 function incentivize(address account, uint256 amount) private {
71     uint256 timestamp = block.timestamp.sub(
72         s.season.start.add(s.season.period.mul(season()))
73     );
74     if (timestamp > 300) timestamp = 300;
75     uint256 incentive = LibIncentive.fracExp(amount, 100,
   ↳ timestamp, 1);
76     C.bean().mint(account, amount);
77     emit Incentivization(account, incentive);
78 }
```

As we can see, the rewards/timestamp is capped at a maximum of 300 seconds and makes use of exponential rewards. But then, in the mint call, the amount parameter is incorrectly used instead of incentive, which means that the caller will always receive a fixed amount of beans (100):



Risk Level:

**Likelihood - 1**
**Impact - 3**

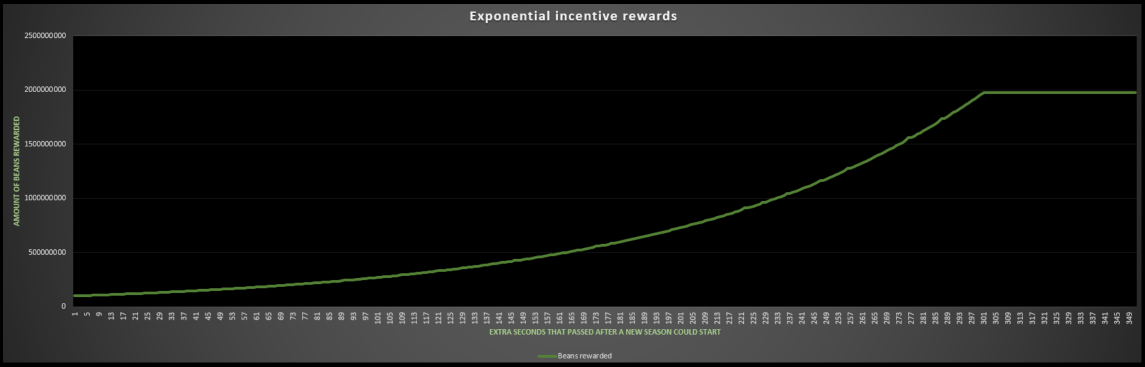
Recommendation:

It is recommended to update the incentivize() function as shown below so the exponential rewards implementation is used:

Listing 12: SeasonFacet.sol (Line 76)

```
70 function incentivize(address account, uint256 amount) private {
71     uint256 timestamp = block.timestamp.sub(
72         s.season.start.add(s.season.period.mul(season()))
73     );
74     if (timestamp > 300) timestamp = 300;
75     uint256 incentive = LibIncentive.fracExp(amount, 100,
   ↳ timestamp, 1);
76     C.bean().mint(account, incentive);
77     emit Incentivization(account, incentive);
78 }
```

This would be the rewarded beans with the suggested/corrected implementation:



Remediation Plan:

**SOLVED**: The Beanstalk team corrected the issue and updated the code as suggested.

## 3.7 (HAL-07) MISSING REQUIRE CHECK IN TOKENFACET.WRAPETH FUNCTION - LOW

### Description:

In the TokenFacet contract, the wrapEth(uint256 amount, LibTransfer.To mode) function wraps the amountof Ether into WETH and sends it to the user internal/external balance:

Listing 13: TokenFacet.sol

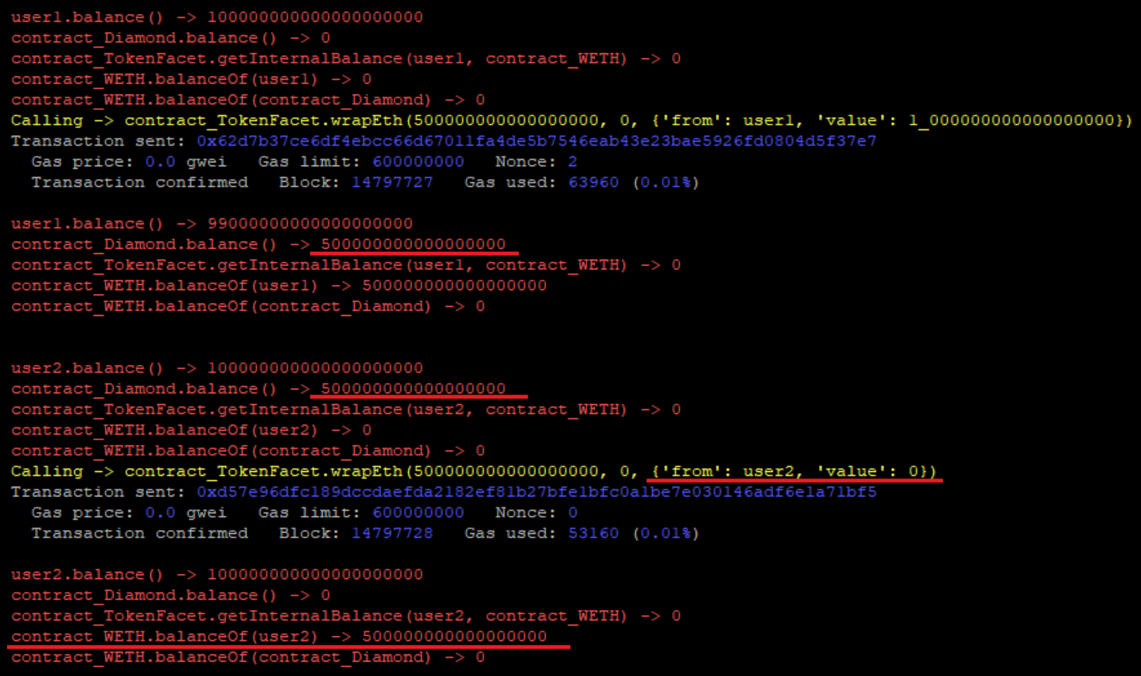
```
52 function wrapEth(uint256 amount, LibTransfer.To mode) external
↳ payable {
53     LibWeth.wrap(amount, mode);
54 }
```

Listing 14: LibWeth.sol (Lines 20,21)

```
19 function wrap(uint256 amount, LibTransfer.To mode) internal {
20     deposit(amount);
21     LibTransfer.sendToken(IERC20(WETH), amount, msg.sender, mode);
22 }
```

As the msg.value is never compared to the amount parameter, if the msg.value sent by the user was higher than the amount the difference would be taken by the contract and any other user would be able to steal it.

## Proof of Concept:



```
user1.balance() -> 100000000000000000000
contract_Diamond.balance() -> 0
contract_TokenFacet.getInternalBalance(user1, contract_WETH) -> 0
contract_WETH.balanceOf(user1) -> 0
contract_WETH.balanceOf(contract_Diamond) -> 0
Calling -> contract_TokenFacet.wrapEth(500000000000000000, 0, {'from': user1, 'value': 1_000000000000000000})
Transaction sent: 0x62d7b37ce6df4ebcc66d67011fa4de5b7546eab43e23bae5926fd0804d5f37e7
  Gas price: 0.0 gwei   Gas limit: 600000000   Nonce: 2
  Transaction confirmed   Block: 14797727   Gas used: 63960 (0.01%)

user1.balance() -> 99000000000000000000
contract_Diamond.balance() -> 500000000000000000
contract_TokenFacet.getInternalBalance(user1, contract_WETH) -> 0
contract_WETH.balanceOf(user1) -> 500000000000000000
contract_WETH.balanceOf(contract_Diamond) -> 0


user2.balance() -> 100000000000000000000
contract_Diamond.balance() -> 500000000000000000
contract_TokenFacet.getInternalBalance(user2, contract_WETH) -> 0
contract_WETH.balanceOf(user2) -> 0
contract_WETH.balanceOf(contract_Diamond) -> 0
Calling -> contract_TokenFacet.wrapEth(500000000000000000, 0, {'from': user2, 'value': 0})
Transaction sent: 0xd57e96dfc189dccdaefda2182ef81b27bfe1bfc0a1be7e030146adf6e1a71bf5
  Gas price: 0.0 gwei   Gas limit: 600000000   Nonce: 0
  Transaction confirmed   Block: 14797728   Gas used: 53160 (0.01%)

user2.balance() -> 100000000000000000000
contract_Diamond.balance() -> 0
contract_TokenFacet.getInternalBalance(user2, contract_WETH) -> 0
contract_WETH.balanceOf(user2) -> 500000000000000000
contract_WETH.balanceOf(contract_Diamond) -> 0
```

## Risk Level:

**Likelihood - 1**
**Impact - 3**

## Recommendation:

It is recommended to add a require statement that checks that msg.value is equal to the amount parameter set in the wrapEth() call.

## Remediation Plan:

**SOLVED**: The Beanstalk team corrected the issue. Ether refunds were added instead of a require check. If there is leftover Ether in the contract, then it will be refunded.

## 3.8 (HAL-08) MULTIPLE OVERFLOWS IN MARKETPLACEFACET - LOW

### Description:

In the MarketplaceFacet there are multiple overflows that can cause some inconsistencies.

One of them is located in the _createPodListing() function:

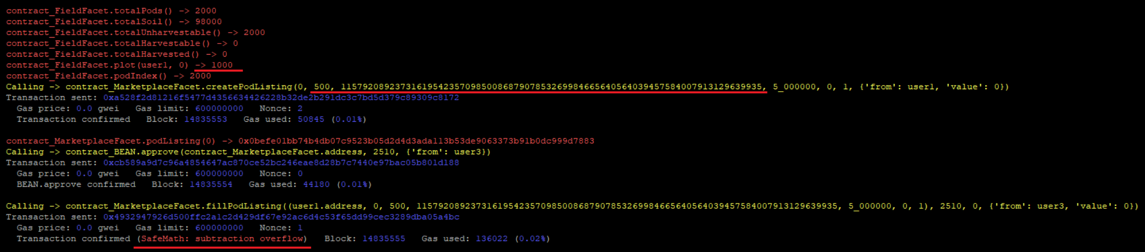
**Listing 15: Listing.sol (Line 60)**

```
50 function _createPodListing(
51     uint256 index,
52     uint256 start,
53     uint256 amount,
54     uint24 pricePerPod,
55     uint256 maxHarvestableIndex,
56     LibTransfer.To mode
57 ) internal {
58     uint256 plotSize = s.a[msg.sender].field.plots[index];
59     require(
60         plotSize >= (start + amount) && amount > 0,
61         "Marketplace: Invalid Plot/Amount."
62     );
63     require(
64         0 < pricePerPod,
65         "Marketplace: Pod price must be greater than 0."
66     );
67     require(
68         s.f.harvestable <= maxHarvestableIndex,
69         "Marketplace: Expired."
70     );
71
72     if (s.podListings[index] != bytes32(0)) _cancelPodListing(
   ↳ index);
73
74     s.podListings[index] = hashListing(
75         start,
76         amount,
77         pricePerPod,
78         maxHarvestableIndex,
79          mode
80      );
81
82      emit PodListingCreated(
83          msg.sender,
84          index,
85          start,
86          amount,
87          pricePerPod,
88          maxHarvestableIndex,
89          mode
90      );
91  }
```

The require(plotSize >= (start + amount)&& amount > 0, "Marketplace: Invalid Plot/Amount."); overflow allows users to create PodListings of very high amounts, although this can not be exploited since when removing the Plots from the seller through the removePlot() function SafeMath is used and the transaction reverts:

Listing 16: PodTransfer.sol (Line 82)

```
72  function removePlot(
73      address account,
74      uint256 id,
75      uint256 start,
76      uint256 end
77  ) internal {
78      uint256 amount = s.a[account].field.plots[id];
79      if (start == 0) delete s.a[account].field.plots[id];
80      else s.a[account].field.plots[id] = start;
81      if (end != amount)
82          s.a[account].field.plots[id.add(end)] = amount.sub(end);
83  }
```



```
contract_FieldFacet.totalPods() -> 2000
contract_FieldFacet.totalSoil() -> 98000
contract_FieldFacet.totalUnharvestable() -> 2000
contract_FieldFacet.totalHarvestable() -> 0
contract_FieldFacet.totalHarvested() -> 0
contract_FieldFacet.plot(user1, 0) -> 1000
contract_FieldFacet.podIndex() -> 2000
Calling -> contract_MarketplaceFacet.createPodListing(0, 500, 115792089237316195423570985008687907853269984665640564039457584007913129639935, 5_000000, 0, 1, {'from': user1, 'value': 0})
Transaction sent: 0xa528f2d81216f5477d4356634426228b32de2b291dc3c7bd5d379c89309c8172
  Gas price: 0.0 gwei   Gas limit: 600000000   Nonce: 2
  Transaction confirmed   Block: 14835553   Gas used: 50845 (0.01%)

contract_MarketplaceFacet.podListing(0) -> 0x0befe01bb74b4db07c9523b05d2d4d3ada113b53de9063373b91b0dc999d7883
Calling -> contract_BEAN.approve(contract_MarketplaceFacet.address, 2510, {'from': user3})
Transaction sent: 0xcb589a9d7c96a4854647ac870ce52bc246eae8d28b7c7440e97bac05b801d188
  Gas price: 0.0 gwei   Gas limit: 600000000   Nonce: 0
  BEAN.approve confirmed   Block: 14835554   Gas used: 44180 (0.01%)

Calling -> contract_MarketplaceFacet.fillPodListing((user1.address, 0, 500, 115792089237316195423570985008687907853269984665640564039457584007913129639935, 5_000000, 0, 1), 2510, 0, {'from': user3, 'value': 0})
Transaction sent: 0x4932947926d500ffc2a1c2d429df67e92ac6d4c53f65dd99cec3289dba05a4bc
  Gas price: 0.0 gwei   Gas limit: 600000000   Nonce: 1
  Transaction confirmed (SafeMath: subtraction overflow)   Block: 14835555   Gas used: 136022 (0.02%)
```

On the other hand, a similar issue occurs in the roundAmount() function:

**Listing 17: Listing.sol (Line 169)**

```
162 // If remainder left (always <1 pod) that would otherwise be unpurchaseable
163 // due to rounding from calculating amount, give it to last buyer
164 function roundAmount(PodListing calldata l, uint256 amount)
165     private
166     pure
167     returns (uint256)
168 {
169     if ((l.amount - amount) < (1000000 / l.pricePerPod)) amount = l.amount;
170     return amount;
171 }
```

Risk Level:

**Likelihood - 1**
**Impact - 3**

Recommendation:

It is recommended to make use of the SafeMath library in the functions described above.

Remediation Plan:

**SOLVED**: The Beanstalk team corrected the issue. All the overflows were addressed.

## 3.9 (HAL-09) FERTILIZERPREMINT.BUYANDMINT FUNCTION COULD BE SANDWICHED - INFORMATIONAL

Description:

In the FertilizerPreMint, the function buy() is used to swap Ether into USDC through the UniswapV3 router:

Listing 18: FertilizerPreMint.sol (Line 104)

```
 94 function buy(uint256 minAmountOut) private returns (uint256 amountOut) {
 95     IWETH(WETH).deposit{value: msg.value}();
 96     ISwapRouter.ExactInputSingleParams memory params =
 97         ISwapRouter.ExactInputSingleParams({
 98             tokenIn: WETH,
 99             tokenOut: USDC,
100             fee: POOL_FEE,
101             recipient: CUSTODIAN,
102             deadline: block.timestamp,
103             amountIn: msg.value,
104             amountOutMinimum: minAmountOut,
105             sqrtPriceLimitX96: 0
106         });
107     amountOut = ISwapRouter(SWAP_ROUTER).exactInputSingle(params);
108 }
```

The amountOutMinimum is set with a user controlled parameter minAmountOut . If the Ether sent through msg.value is higher than the minAmountOut in USDC the transaction may get sandwiched causing the user to swap Ether for USDC at a higher cost, receiving less USDC for the same amount of Ether.

The issue was flagged as informational, as there is a function in the FertilizerPreMint contract that allows to get the exact amount of USDC for a given amount of Ether after swap. We assume that this function is used in the backend mitigating the issue. Only users interacting with the smart contract directly may have the problem described.

### Risk Level:

**Likelihood - 1**
**Impact - 1**

### Recommendation:

It is recommended to inform the users, specially whales, that they should try to avoid interacting with the smart contract directly for this and that if they do, inform them on how they should determine the minAmountOut preventing them from getting sandwiched.

### Remediation Plan:

**SOLVED**: The Beanstalk team documented their code mentioning that any slippage should be properly accounted by the users:

Listing 19: FertilizerPreMint.sol (Line 49)

```
// Note: Slippage should be properly be accounted for in
// minBuyAmount when calling the buyAndMint function directly.
function buyAndMint(uint256 minBuyAmount) external payable nonReentrant {
    uint256 amount = buy(minBuyAmount);
    require(IUSDC.balanceOf(CUSTODIAN) <= MAX_RAISE, "Fertilizer: Not enough remaining");
    __mint(amount);
}
```

## 3.10 (HAL-10) POD PRICE IS LIMITED TO 16.7 BEANS - INFORMATIONAL

Description:

In the MarketplaceFacet, the functions createPodListing() and createPodOrder() make use of an uint24 to hold the pricePerPod parameter.

As the maximum value that an uint24 can hold is 16_777215 the users will not be able to set a price higher than that for a Pod.

Listing 20: MarketplaceFacet.sol (Line 26)

```
22 function createPodListing(
23     uint256 index,
24     uint256 start,
25     uint256 amount,
26     uint24 pricePerPod,
27     uint256 maxHarvestableIndex,
28     LibTransfer.To mode
29 ) external payable {
30     _createPodListing(
31         index,
32         start,
33         amount,
34         pricePerPod,
35         maxHarvestableIndex,
36         mode
37     );
38 }
```

Listing 21: MarketplaceFacet.sol (Line 73)

```
71 function createPodOrder(
72     uint256 beanAmount,
73     uint24 pricePerPod,
74     uint256 maxPlaceInLine,
75     LibTransfer.From mode
76 ) external payable returns (bytes32 id) {
77     beanAmount = LibTransfer.receiveToken(C.bean(), beanAmount,
   ↳ msg.sender, mode);
```

```
    return _createPodOrder(beanAmount, pricePerPod, maxPlaceInLine);
}
```

### Risk Level:

**Likelihood - 1**
**Impact - 1**

### Recommendation:

It is recommended to consider using an uint64 instead to allow users to set higher prices for the Pods.

### Remediation Plan:

**SOLVED**: The Beanstalk team documented their code mentioning that the highest price to list a Pod for is 16_777215 Beans:

**Listing 22: MarketplaceFacet.sol (Line 22)**

```
22 // Note: pricePerPod is bounded by 16_777_215 Beans.
23 function createPodListing(
24     uint256 index,
25     uint256 start,
26     uint256 amount,
27     uint24 pricePerPod,
28     uint256 maxHarvestableIndex,
29     LibTransfer.To mode
30 ) external payable {
31     _createPodListing(
32         index,
33         start,
34         amount,
35         pricePerPod,
36         maxHarvestableIndex,
37         mode
38     );
39 }
```

Listing 23: MarketplaceFacet.sol (Line 72)

```
// Note: pricePerPod is bounded by 16_777_215 Beans.
function createPodOrder(
    uint256 beanAmount,
    uint24 pricePerPod,
    uint256 maxPlaceInLine,
    LibTransfer.From mode
) external payable returns (bytes32 id) {
    beanAmount = LibTransfer.receiveToken(C.bean(), beanAmount, msg.sender, mode);
    return _createPodOrder(beanAmount, pricePerPod, maxPlaceInLine);
}
```

## 3.11 (HAL-11) FARMFACET: USE OF DELEGATECALL IN A FOR LOOP - INFORMATIONAL

Description:

The FarmFacet allows performing multiple delegatecalls inside a for loop:

Listing 24: FarmFacet.sol (Lines 23,37,43)

```
23 function _farm(bytes calldata data) private {
24     LibDiamond.DiamondStorage storage ds;
25     bytes32 position = LibDiamond.DIAMOND_STORAGE_POSITION;
26     assembly {
27         ds.slot := position
28     }
29     bytes4 functionSelector;
30     assembly {
31         functionSelector := calldataload(data.offset)
32     }
33     address facet = ds
34         .selectorToFacetAndPosition[functionSelector]
35         .facetAddress;
36     require(facet != address(0), "Diamond: Function does not exist
   ↳ ");
37     (bool success, ) = address(facet).delegatecall(data);
38     require(success, "FarmFacet: Function call failed!");
39 }
40
41 function farm(bytes[] calldata data) external payable {
42     for (uint256 i = 0; i < data.length; i++) {
43         _farm(data[i]);
44     }
45     if (msg.value > 0 && address(this).balance > 0) {
46         (bool success, ) = msg.sender.call{value: address(this).
   ↳ balance}(
47             new bytes(0)
48         );
49         require(success, "Farm: Eth transfer Failed.");
50     }
51 }
```

In this situation, msg.sender and msg.value would be persisted across the different iterations/delegatecalls in the loop. For example, a user could submit 1 Ether as msg.value to the farm(bytes[] calldata data) call and in the data array add 3 different calls that each of those made use of that Ether. If the Diamond contract had some Ether, user would be paying just that Ether and the 2 remaining Ether would be taken from the smart contract balance.

Currently, there is no exploitation path for this issue, as the contracts should never be holding any Ether. Also, the remaining Ether in the contract is sent back to msg.sender after the _farm() calls.

For this reason, we have set this risk as informational.

### References:

Multi Delegatecall: Solidity 0.8
samczsun's blog post

### Risk Level:

**Likelihood - 1**
**Impact - 1**

### Recommendation:

It is recommended to make sure that the overall logic and future upgrades of the contracts are compatible with this functionality, so no bugs are introduced in the code.

### Remediation Plan:

**SOLVED/ACKNOWLEDGED**: The Beanstalk team is aware of the issue and will take this into account in future upgrades.

## 3.12 (HAL-12) CRITICAL DEPENDENCY ON CURVE METAPOOL FACTORIES - INFORMATIONAL

Description:

In the CurveFacet there are multiple functions that make use of the approveToken() function, for example:

Listing 25: CurveFacet.sol (Line 45)

```
function exchange(
    address pool,
    address fromToken,
    address toToken,
    uint256 amountIn,
    uint256 minAmountOut,
    bool stable,
    LibTransfer.From fromMode,
    LibTransfer.To toMode
) external payable nonReentrant {
    (int128 i, int128 j) = getIandJ(fromToken, toToken, pool, stable);
    amountIn = IERC20(fromToken).receiveToken(
        amountIn,
        msg.sender,
        fromMode
    );
    IERC20(fromToken).approveToken(pool, amountIn);

    if (toMode == LibTransfer.To.EXTERNAL) {
        ICurvePoolR(pool).exchange(
            i,
            j,
            amountIn,
            minAmountOut,
            msg.sender
        );
    } else {
        uint256 amountOut = ICurvePool(pool).exchange(
            i,
            j,
            amountIn,
            minAmountOut
        );
        msg.sender.increaseInternalBalance(IERC20(toToken), amountOut);
    }
}
```

pool and fromToken are user controlled parameters. On the other hand, the LibTransfer.From fromMode set to INTERNAL_TOLERANT would allow anyone to bypass this receiveToken() call.

The only blocker to avoid an attacker of approving his own address and extract all the tokens of the contract is the following require statement:

Listing 26: CurveFacet.sol (Line 301)

```
286 function getIandJ(
287     address from,
288     address to,
289     address pool,
290     bool stable
291 ) private view returns (int128 i, int128 j) {
292     address factory = stable ? STABLE_FACTORY : CRYPTO_FACTORY;
293     address[4] memory coins = ICurveFactory(factory).get_coins(pool);
294     i = 4;
295     j = 4;
296     for (uint256 _i = 0; _i < 4; ++_i) {
297         if (coins[_i] == from) i = int128(_i);
298         else if (coins[_i] == to) j = int128(_i);
299         else if (coins[_i] == address(0)) break;
300     }
301     require(i < 4 && j < 4, "Curve: Tokens not in pool");
302 }
```

In case of a malicious Curve Metapool Factory (0xB9fC157394Af804a3578134A6585C0dc9cc99 or 0x0959158b6040D32d04c301A72CBFD6b39E21c9AE), all the tokens in the contracts could be drained.

Risk Level:

**Likelihood - 1**
**Impact - 1**

Recommendation:

No recommendation against this issue. The issue described likelihood is minimum but something to be aware of.

Remediation Plan:

**ACKNOWLEDGED**: The Beanstalk team acknowledges this.

# 3.13 (HAL-13) SAFETRANSFER IS NOT USED FOR ALL THE TOKEN TRANSFERS - INFORMATIONAL

Description:

SafeERC20.safeTransferFrom() is used in all the code base. Although in the LibTransfer.transferToken() function, the standard ERC20.transferFrom() is still used.

Code Location:

Listing 27: LibTransfer.sol (Line 37)

```
29 function transferToken(
30     IERC20 token,
31     address recipient,
32     uint256 amount,
33     From fromMode,
34     To toMode
35 ) internal returns (uint256 transferredAmount) {
36     if (fromMode == From.EXTERNAL && toMode == To.EXTERNAL) {
37         token.transferFrom(msg.sender, recipient, amount);
38         return amount;
39     }
40     amount = receiveToken(token, amount, msg.sender, fromMode);
41     sendToken(token, amount, recipient, toMode);
42     return amount;
43 }
```

Risk Level:

**Likelihood - 1**
**Impact - 1**

Recommendation:

It is recommended to use SafeERC20.safeTransferFrom() also in the LibTransfer.transferToken() function.

Remediation Plan:

**SOLVED**: Beanstalk team uses now SafeERC20 in all the token transfers.

## 3.14 (HAL-14) REQUIRE STATEMENT TYPOS - INFORMATIONAL

Description:

In the following require statements some typos were detected:

LibBalance.sol
- Line 73:

```
require(allowPartial || (currentBalance >= amount), "Balance:
Insufficnent internal balance");
```

TokenSilo.sol
- Line 285:

```
require(season <= s.season.current, "Claim: Withdrawal not recievable
.");
```

LibFertilizer.sol
- Line 153:

```
require(s.activeFertilizer == 0, "Still active fertliizer");
```

Risk Level:

**Likelihood - 1**
**Impact - 1**

Recommendation:

It is recommended to correct the require statement messages highlighted.

Remediation Plan:

**SOLVED**: Beanstalk team corrected the typos suggested.

## 3.15 (HAL-15) INITIALIZE FUNCTION IN FERTILIZER CONTRACT CAN BE REMOVED - INFORMATIONAL

Description:

Currently, the FertilizerPreMint contract is deployed behind a TransparentUpgradeableProxy.

After the replanting, when Beanstalk is unpaused, the BCM will call the function addFertilizerOwner() which will handle the process of adding BEAN:3CRV liquidity and minting new Deposited Beans for all the Fertilizer minted prior to unpause.

At the same time, the TransparentUpgradeableProxy contract will be upgraded to a new Fertilizer contract, instead of the FertilizerPreMint implementation used before. This will move the mintFertilizer() functionality to Beanstalk itself, instead of happening in the FertilizerPreMint contract.

At this point, Beanstalk will automatically add new liquidity for Unripe LP holders and new Beans in the same transaction as when Fertilizer is minted.

The new Fertilizer contract that will be used contains an initialize() function:

**Listing 28: Fertilizer.sol**

```
function decreaseInternalBalance(
function initialize() public initializer { //@audit can be removed
    __Internallize_init("");
}
```

As the TransparentUpgradeableProxy holds all the storage variables and will be already initialized in the FertilizerPreMint implementation, any call to this function will revert as the contract will be already initialized, hence this initialize() function can be removed from the Fertilizer contract.

Risk Level:

**Likelihood - 1**
**Impact - 1**

Recommendation:

It is recommended to consider removing the initialize() function from the Fertilizer contract in order to reduce the deployment gas costs.

Remediation Plan:

**SOLVED**: Beanstalk team removed the initialize() function from the Fertilizer contract.

## 3.16 (HAL-16) UNNEEDED INITIALIZATION OF UINT256 VARIABLES TO 0 - INFORMATIONAL

Description:

As i is an uint256, it is already initialized to 0. uint256 i = 0 reassigns the 0 to i which wastes gas.

Code Location:

Internalizer.sol
- Line 62:

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

Fertilizer.sol
- Line 77:

```
for (uint256 i = 0; i < ids.length; i++){
```

- Line 90:

```
for (uint256 i = 0; i < ids.length; i++){
```

- Line 99:

```
for (uint256 i = 0; i < ids.length; i++){
```

Fertilizer1155.sol
- Line 67:

```
for (uint256 i = 0; i < ids.length; ++i){
```

TokenSilo.sol
- Line 210:

```
for (uint256 i = 0; i < seasons.length; i++){
```

- Line 268:

```
for (uint256 i = 0; i < seasons.length; i++){
```

- Line 325:

```
for (uint256 i = 0; i < seasons.length; i++){
```

SiloFacet.sol

- Line 140:

```
for (uint256 i = 0; i < seasons.length; ++i){
```

TokenFacet.sol

- Line 82:

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

- Line 103:

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

- Line 124:

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

- Line 147:

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

FieldFacet.sol

- Line 84:

```
for (uint256 i = 0; i < plots.length; i++){
```

FarmFacet.sol

- Line 44:

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

CurveFacet.sol

- Line 109:

```
for (uint256 i = 0; i < nCoins; ++i){
```

- Line 167:

```
for (uint256 i = 0; i < nCoins; i++)
```

- Line 174:

```
for (uint256 i = 0; i < nCoins; i++)
```

- Line 186:

```
for (uint256 i = 0; i < nCoins; i++)
```

- Line 191:

```
for (uint256 i = 0; i < nCoins; ++i){
```

- Line 246:

```
for (uint256 i = 0; i < nCoins; ++i){
```

- Line 296:

```
for (uint256 _i = 0; _i < 4; ++_i){
```

- Line 313:

```
for (uint256 _i = 0; _i < 8; ++_i){
```

- Line 329:

```
for (uint256 _i = 0; _i < 4; ++_i){
```

LibPlainCurveConvert.sol

- Line 79:

```
for (uint256 k = 0; k < 256; k++){
```

LibCurve.sol

- Line 56:

```
for (uint256 _i = 0; _i < N_COINS; _i++){
```

- Line 68:

```
for (uint256 _i = 0; _i < 255; _i++){
```

- Line 85:

```
for (uint256 _i = 0; _i < xp.length; _i++){
```

- Line 92:

```
for (uint256 _i = 0; _i < 256; _i++){
```

- Line 94:

```
for (uint256 _j = 0; _j < xp.length; _j++){
```

LibIncentive.sol

- Line 34:

```
for (uint256 i = 0; i < p; ++i){
```

Risk Level:

**Likelihood - 1**
**Impact - 1**

Recommendation:

It is recommended to not initialize uint variables to 0 to save some gas.
For example, use instead:

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

Remediation Plan:

**SOLVED**: Beanstalk team followed Halborn’s suggestion reducing the gas costs.

## 3.17 (HAL-17) USING POSTFIX OPERATORS IN LOOPS - INFORMATIONAL

Description:

In the loops below, postfix (e.g. i++) operators were used to increment or decrement variable values. In loops, using prefix operators (e.g. ++i) costs less gas per iteration than using postfix operators.

Code Location:

Internalizer.sol

- Line 62:

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

Fertilizer.sol

- Line 77:

```
for (uint256 i = 0; i < ids.length; i++){
```

- Line 90:

```
for (uint256 i = 0; i < ids.length; i++){
```

- Line 99:

```
for (uint256 i = 0; i < ids.length; i++){
```

TokenSilo.sol

- Line 210:

```
for (uint256 i = 0; i < seasons.length; i++){
```

- Line 268:

```
for (uint256 i = 0; i < seasons.length; i++){
```

- Line 325:

```
for (uint256 i = 0; i < seasons.length; i++){
```

TokenFacet.sol

- Line 82:

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

- Line 103:

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

- Line 124:

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

- Line 147:

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

FieldFacet.sol

- Line 84:

```
for (uint256 i = 0; i < plots.length; i++){
```

DiamondLoupeFacet.sol

- Line 32:

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

FarmFacet.sol

- Line 44:

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

CurveFacet.sol

- Line 167:

```
for (uint256 i = 0; i < nCoins; i++)
```

- Line 174:

```
for (uint256 i = 0; i < nCoins; i++)
```

- Line 186:

```
for (uint256 i = 0; i < nCoins; i++)
```

LibPlainCurveConvert.sol

- Line 79:

```
for (uint256 k = 0; k < 256; k++){
```

LibDiamond.sol

- Line 110:

```
for (uint256 facetIndex; facetIndex < _diamondCut.length; facetIndex++)
{
```

- Line 135:

```
for (uint256 selectorIndex; selectorIndex < _functionSelectors.length;
selectorIndex++){
```

- Line 153:

```
for (uint256 selectorIndex; selectorIndex < _functionSelectors.length;
selectorIndex++){
```

- Line 168:

```
for (uint256 selectorIndex; selectorIndex < _functionSelectors.length;
selectorIndex++){
```

LibCurve.sol

- Line 56:

```
for (uint256 _i = 0; _i < N_COINS; _i++){
```

- Line 68:

```
for (uint256 _i = 0; _i < 255; _i++){
```

- Line 85:

```
for (uint256 _i = 0; _i < xp.length; _i++){
```

- Line 92:

```
for (uint256 _i = 0; _i < 256; _i++){
```

- Line 94:

```
for (uint256 _j = 0; _j < xp.length; _j++){
```

Decimal.sol

- Line 140:

```
for (uint256 i = 1; i < b; i++){
```

Proof of Concept:

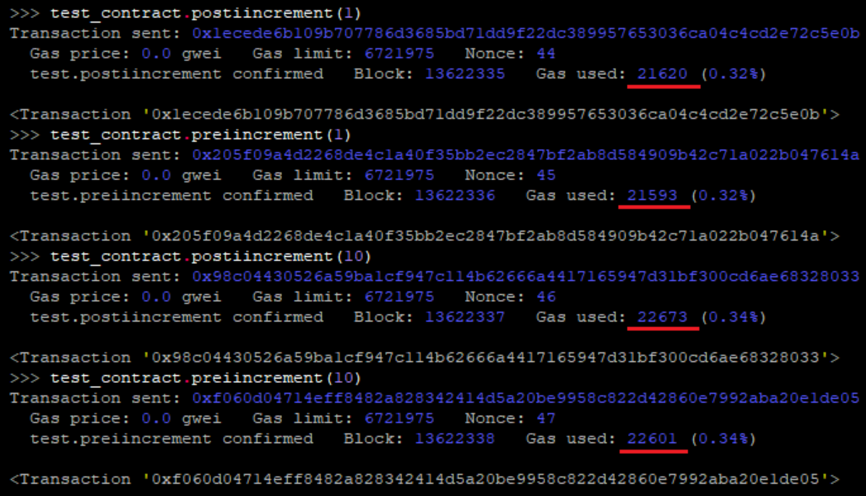
For example, based in the following test contract:

Listing 29: Test.sol

```
//SPDX-License-Identifier: MIT
pragma solidity 0.8.9;

contract test {
    function postiincrement(uint256 iterations) public {
        for (uint256 i = 0; i < iterations; i++) {
        }
    }
    function preiincrement(uint256 iterations) public {
        for (uint256 i = 0; i < iterations; ++i) {
        }
    }
}
```

We can see the difference in the gas costs:



```
>>> test_contract.postiincrement(1)
Transaction sent: 0x1ecede6b109b707786d3685bd71dd9f22dc389957653036ca04c4cd2e72c5e0b
  Gas price: 0.0 gwei   Gas limit: 6721975   Nonce: 44
  test.postiincrement confirmed   Block: 13622335   Gas used: 21620 (0.32%)

<Transaction '0x1ecede6b109b707786d3685bd71dd9f22dc389957653036ca04c4cd2e72c5e0b'>
>>> test_contract.preiincrement(1)
Transaction sent: 0x205f09a4d2268de4c1a40f35bb2ec2847bf2ab8d584909b42c71a022b047614a
  Gas price: 0.0 gwei   Gas limit: 6721975   Nonce: 45
  test.preiincrement confirmed   Block: 13622336   Gas used: 21593 (0.32%)

<Transaction '0x205f09a4d2268de4c1a40f35bb2ec2847bf2ab8d584909b42c71a022b047614a'>
>>> test_contract.postiincrement(10)
Transaction sent: 0x98c04430526a59ba1cf947c114b62666a4417165947d31bf300cd6ae68328033
  Gas price: 0.0 gwei   Gas limit: 6721975   Nonce: 46
  test.postiincrement confirmed   Block: 13622337   Gas used: 22673 (0.34%)

<Transaction '0x98c04430526a59ba1cf947c114b62666a4417165947d31bf300cd6ae68328033'>
>>> test_contract.preiincrement(10)
Transaction sent: 0xf060d04714eff8482a828342414d5a20be9958c822d42860e7992aba20e1de05
  Gas price: 0.0 gwei   Gas limit: 6721975   Nonce: 47
  test.preiincrement confirmed   Block: 13622338   Gas used: 22601 (0.34%)

<Transaction '0xf060d04714eff8482a828342414d5a20be9958c822d42860e7992aba20e1de05'>
```

## Risk Level:

**Likelihood - 1**
**Impact - 1**

## Recommendation:

It is recommended to use ++i instead of i++ to increment the value of an uint variable inside a loop. This does not only apply to the iterator variable. It also applies to increment/decrement done inside the loop code block.

## Remediation Plan:

**SOLVED**: Beanstalk team followed Halborn’s suggestion and now uses prefix operators to increment the value of an uint variable inside loops reducing the gas costs.

THANK YOU FOR CHOOSING

HALBORN