Beanstalk - Farms Update

Smart Contract Security Audit

Prepared by: **Halborn**

Date of Engagement: **September 5th, 2022 - November 14th, 2022**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	11/02/2022	Francisco González
0.2	Document Updates	11/10/2022	Francisco González
0.3	Document Updates	11/14/2022	Francisco González
0.4	Draft Review	11/14/2022	Kubilay Onur Gungor
0.5	Draft Review	11/15/2022	Gabi Urrutia
1.0	Remediation Plan	12/13/2022	Francisco González
1.1	Remediation Plan Review	12/13/2022	Roberto Reigada
1.2	Remediation Plan Review	12/13/2022	Piotr Cielas
1.3	Remediation Plan Review	12/13/2022	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
Kubilay Onur Gungor	Halborn	Kubilay.Gungor@halborn.com
Francisco González	Halborn	Francisco.Villarejo@halborn.com
Roberto Reigada	Halborn	Roberto.Reigada@halborn.com
Piotr Cielas	Halborn	Piotr.Cielas@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

Beanstalk engaged Halborn to conduct a security audit on their smart contracts beginning on September 5th, 2022 and ending on November 14th, 2022. The security assessment was scoped to the smart contracts provided in the GitHub repository BeanstalkFarms/Beanstalk.

This report also includes some findings that were already reported and fixed in some parallel audits performed to separate protocol components. However, since some of those vulnerabilities are present in the scoped code, they have been included to improve readability by unifying the findings and obtaining a standalone report which covers the vulnerabilities present in the code without the need to switch between different reports.

1.2 AUDIT SUMMARY

The team at Halborn was provided 10 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:

- Perform a re-audit of the complete BeanStalk codebase.
- Ensure that smart contract functions operate as intended.
- Identify potential security issues with the smart contracts.

In summary, Halborn identified a few 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 smart contracts contained in the scoped repository.

Audit Commit ID:

1447c2426a97a069cc633e6f8b2b1937c91d5da9

Pod Marketplace V2 Fixed Commit ID:

- 0bdd376263b0fe94af84aaf4adb6391b39fa80ab

BIP 24 Fixed Commit ID:

- 6699e071626a17283facc67242536037989ecd91

2. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
1	0	2	1	8

LIKELIHOOD

IMPACT

				(HAL-01)
		(HAL-02) (HAL-03)		
(HAL-12)	(HAL-04)			
(HAL-05) (HAL-06) (HAL-07) (HAL-08) (HAL-09) (HAL-10) (HAL-11)				

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL01 - UNDERLYING TOKENS CAN BE DRAINED THROUGH THE UNRIPEFACET.CHOP FUNCTION	Critical	SOLVED - 09/16/2022
HAL02 - MULTIPLE UNDERFLOWS/OVERFLOWS	Medium	SOLVED - 10/17/2022
HAL03 - LISTINGS CAN BE DELETED BY ANYONE	Medium	SOLVED - 10/27/2022
HAL04 - PLOTS CAN BE UNCONTROLLABLY SPLITTED	Low	SOLVED - 10/27/2022
HAL05 - USING POSTFIX OPERATORS IN LOOPS	Informational	ACKNOWLEDGED
HAL06 - UNNEEDED INITIALIZATION OF UINT256 VARIABLES TO 0	Informational	ACKNOWLEDGED
HAL07 - USAGE OF AND OPERATOR IN REQUIRE STATEMENTS	Informational	ACKNOWLEDGED
HAL08 - OPTIMIZE UNSIGNED INTEGER COMPARISON	Informational	ACKNOWLEDGED
HAL09 - INCOMPLETE NATSPEC DOCUMENTATION	Informational	ACKNOWLEDGED
HAL10 - INCREASE OPTIMIZER RUNS	Informational	ACKNOWLEDGED
HAL11 - SOLC 0.7.6 COMPILER VERSION CONTAINS MULTIPLE BUGS	Informational	ACKNOWLEDGED
HAL12 - OUT OF DATE OPENZEPPELIN PACKAGES	Informational	ACKNOWLEDGED

FINDINGS & TECH DETAILS

3.1 (HAL-01) UNDERLYING TOKENS CAN BE DRAINED THROUGH THE UNRIPEFACET.CHOP FUNCTION - CRITICAL

Description:

Note that this finding was also reported on the **BIP24** audit, and it has already been fixed by the Beanstalk team.

In the UnripeFacet, the chop() function is used to burn unripeTokens in order to receive in exchange an underlyingToken like, for example, Beans:

Listing 1: UnripeFacet.sol (Line 61)

```
51 function chop(
52     address unripeToken,
53     uint256 amount,
54     LibTransfer.From fromMode,
55     LibTransfer.To toMode
56 ) external payable nonReentrant returns (uint256 underlyingAmount)
   ↳  {
57     underlyingAmount = getPenalizedUnderlying(unripeToken, amount)
   ↳ ;
58
59     LibUnripe.decrementUnderlying(unripeToken, underlyingAmount);
60
61     LibTransfer.burnToken(IBean(unripeToken), amount, msg.sender,
   ↳ fromMode);
62
63     address underlyingToken = s.u[unripeToken].underlyingToken;
64
65     IERC20(underlyingToken).sendToken(underlyingAmount, msg.sender
   ↳ , toMode);
66
67     emit Chop(msg.sender, unripeToken, amount, underlyingAmount);
68 }
```

The burn of the unripeTokens is done through the LibTransfer.burnToken()

call:

Listing 2: LibTransfer.sol (Lines 87,95)

```
82 function burnToken(
83     IBean token,
84     uint256 amount,
85     address sender,
86     From mode
87 ) internal returns (uint256 burnt) {
88     // burnToken only can be called with Unripe Bean, Unripe Bean
↳ :3Crv or Bean token, which are all Beanstalk tokens.
89     // Beanstalk's ERC-20 implementation uses OpenZeppelin's
↳ ERC20Burnable
90     // which reverts if burnFrom function call cannot burn full
↳ amount.
91     if (mode == From.EXTERNAL) {
92         token.burnFrom(sender, amount);
93         burnt = amount;
94     } else {
95         burnt = LibTransfer.receiveToken(token, amount, sender,
↳ mode);
96         token.burn(burnt);
97     }
98 }
```

The LibTransfer.burnToken() function returns the actual amount of tokens that were burnt.

The LibTransfer.From fromMode has 4 different modes:

- EXTERNAL
- INTERNAL
- EXTERNAL_INTERNAL
- INTERNAL_TOLERANT

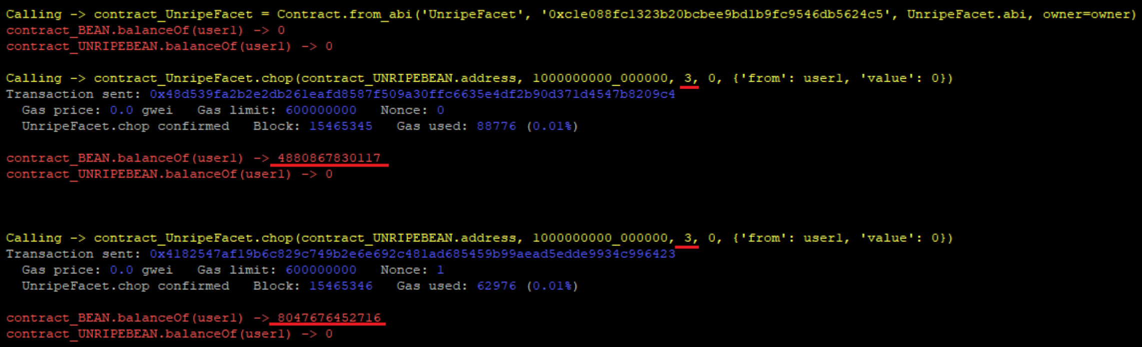
With the INTERNAL_TOLERANT, fromMode tokens will be collected from the user's Internal Balance, and the transaction will not fail if there are not enough tokens.

This INTERNAL_TOLERANT fromMode can be used in the UnripeFacet.chop()

call. As the chop() function is not checking the return value of the LibTransfer.burnToken(), the contract will always assume that the full amount is being burnt when that will not always be true. If a user actually has 0 unripeTokens and uses the INTERNAL_TOLERANT fromMode, no tokens will be burned at all, but the full amount of underlyingTokens will be sent to the user.

Proof of Concept:

This test was done forking the Ethereum mainnet on block 15465331 (Sep-03-2022 12:16:18 PM +UTC):



```
Calling -> contract_UnripeFacet = Contract.from_abi('UnripeFacet', '0xc1e088fc1323b20bcbee9bd1b9fc9546db5624c5', UnripeFacet.abi, owner=owner)
contract_BEAN.balanceOf(user1) -> 0
contract_UNRIPEBEAN.balanceOf(user1) -> 0

Calling -> contract_UnripeFacet.chop(contract_UNRIPEBEAN.address, 1000000000_000000, 3, 0, {'from': user1, 'value': 0})
Transaction sent: 0x48d539fa2b2e2db261eafd8587f509a30ffc6635e4df2b90d371d4547b8209c4
  Gas price: 0.0 gwei   Gas limit: 600000000   Nonce: 0
  UnripeFacet.chop confirmed   Block: 15465345   Gas used: 88776 (0.01%)

contract_BEAN.balanceOf(user1) -> 4880867830117
contract_UNRIPEBEAN.balanceOf(user1) -> 0


Calling -> contract_UnripeFacet.chop(contract_UNRIPEBEAN.address, 1000000000_000000, 3, 0, {'from': user1, 'value': 0})
Transaction sent: 0x4182547af19b6c829c749b2e6e692c481ad685459b99aead5edde9934c996423
  Gas price: 0.0 gwei   Gas limit: 600000000   Nonce: 1
  UnripeFacet.chop confirmed   Block: 15465346   Gas used: 62976 (0.01%)

contract_BEAN.balanceOf(user1) -> 8047676452716
contract_UNRIPEBEAN.balanceOf(user1) -> 0
```

Risk Level:

Likelihood - 5
Impact - 5

Recommendation:

It is recommended to save the return value of the LibTransfer.burnToken() call and overwrite the amount variable with that return as shown below:

Listing 3: UnripeFacet.sol (Line 57)

```
function chop(
    address unripeToken,
    uint256 amount,
    LibTransfer.From fromMode,
    LibTransfer.To toMode
) external payable nonReentrant returns (uint256 underlyingAmount) {
    amount = LibTransfer.burnToken(IBean(unripeToken), amount, msg.sender, fromMode);

    underlyingAmount = getPenalizedUnderlying(unripeToken, amount);

    LibUnripe.decrementUnderlying(unripeToken, underlyingAmount);

    address underlyingToken = s.u[unripeToken].underlyingToken;

    IERC20(underlyingToken).sendToken(underlyingAmount, msg.sender, toMode);

    emit Chop(msg.sender, unripeToken, amount, underlyingAmount);
}
```

Remediation Plan:

SOLVED: The Beanstalk team fixed the issue by now taking also considering the return value of the LibTransfer.burnToken().

Commit ID: 822863f253b251abb6ea656c122dd8d421dc42b3

3.2 (HAL-02) MULTIPLE UNDERFLOWS/OVERFLOWS - MEDIUM

Description:

Note that this finding was also reported on **Pod Market V2** audit, and it has been already fixed by the Beanstalk team.

In some MarketplaceFacet related contracts, there are multiple overflows that can cause some inconsistencies.

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

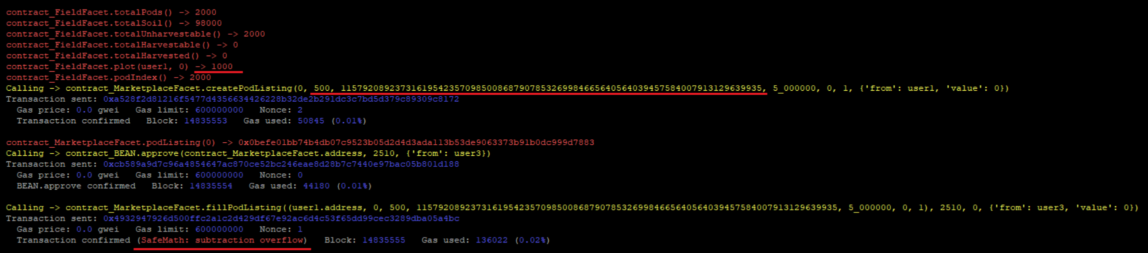
Listing 4: Listing.sol (Line 111)

```
 97     function _fillListing(PodListing calldata l, uint256
  ↳ beanAmount) internal {
 98         bytes32 lHash = hashListing(
 99             l.start,
100             l.amount,
101             l.pricePerPod,
102             l.maxHarvestableIndex,
103             l.mode
104         );
105         require(
106             s.podListings[l.index] == lHash,
107             "Marketplace: Listing does not exist."
108         );
109         uint256 plotSize = s.a[l.account].field.plots[l.index];
110         require(
111             plotSize >= (l.start + l.amount) && l.amount > 0,
112             "Marketplace: Invalid Plot/Amount."
113         );
114         require(
115             s.f.harvestable <= l.maxHarvestableIndex,
116             "Marketplace: Listing has expired."
117         );
118
119         uint256 amount = beanAmount.mul(1000000).div(l.pricePerPod
  ↳ );
120         amount = roundAmount(l, amount);
121
122         __fillListing(msg.sender, l, amount);
123         _transferPlot(l.account, msg.sender, l.index, l.start,
 ↳ amount);
124     }
```

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

Listing 5: 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:

Order.sol

- Line 93:

require(s.a[msg.sender].field.plots[index] >= (start + amount), " Marketplace: Invalid Plot.");

- Line 97:

```
uint256 placeInLineEndPlot = index + start + amount - s.f.harvestable;
```

Risk Level:

Likelihood - 3
Impact - 3

Recommendation:

Using the SafeMath library in all the code lines described above is recommended.

Remediation Plan:

SOLVED: The Beanstalk team solved the issue and now uses the SafeMath library in all the code lines suggested.

Commit ID: 1ddb2f4773e39fc3a18e60a7fa0789a45e017f4c

3.3 (HAL-03) LISTINGS CAN BE DELETED BY ANYONE - MEDIUM

Description:

Note that this finding was also reported on **Pod Market V2** audit, and it has been already fixed by the Beanstalk team.

MarketplaceFacet.sol and its related contracts and libraries implements Listings and Orders, which allow users to buy and sell their pod in a decentralized, trustless fashion.

When any user wants to sell their pods, a listing containing the plot, the pods being sold within the plot, the price per pod, and the expiration time (in the number of pods). When another user wants to buy these pods, he has to fulfill the listing.

Listings can be partially fulfilled, meaning that users can buy only a part of the pods listed. When a listing is partially fulfilled, a new listing is created in the index (currentIndex + beanAmount) containing the remaining unsold pods, and the previous listing is deleted.

However, it has been detected that a griefer could fill a listing introducing 0 in beanAmount, forcing the new position to be created at the same index, and then deleted, causing the position to be cancelled. This could allow any well motivated griefer to constantly prevent any user to sell his pods, cancel listings whose pods are about to become harvestable, etc.

Code Location:

Listing 6: Listing.sol (Lines 134-140,142)

```
126     function __fillListing(
127         address to,
128         PodListing calldata l,
129         uint256 amount
```

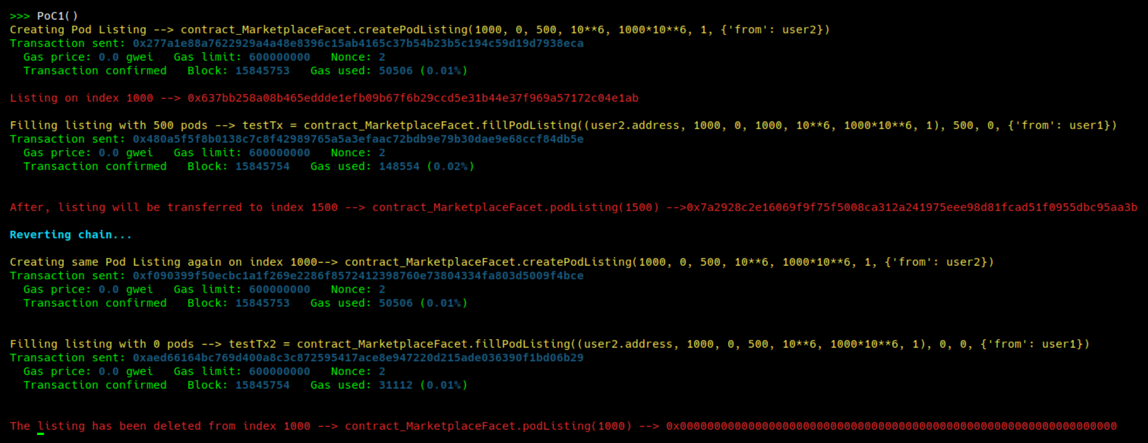
```
    ) private {
        // Note: If l.amount < amount, the function roundAmount will revert

        if (l.amount > amount)
            s.podListings[l.index.add(amount).add(l.start)] = hashListing(
                0,
                l.amount.sub(amount),
                l.pricePerPod,
                l.maxHarvestableIndex,
                l.mode
            );
        emit PodListingFilled(l.account, to, l.index, l.start, amount);
        delete s.podListings[l.index];
    }
```

Proof of Concept:

For this PoC, user2 will list 1000 pods on index 1000. Subsequently, another user will fill that listing with 500 pods, meaning that a new listing will be created on index 1500 with the remaining 500 pods. That would represent a typical use case.

Thereafter, the chain will be reverted, and the same listing will be created, but this time, the listing will be filled with 0 pods. That means a new listing with the remaining pods (1000) will be created on the same index (previous index + beanAmount which is 0), and then the listing on the previous index will be deleted. This will result in having the listing canceled by an external user:



```
>>> PoC1()
Creating Pod Listing --> contract_MarketplaceFacet.createPodListing(1000, 0, 500, 10**6, 1000*10**6, 1, {'from': user2})
Transaction sent: 0x277a1e88a7622929a4a48e8396c15ab4165c37b54b23b5c194c59d19d7938eca
  Gas price: 0.0 gwei   Gas limit: 600000000   Nonce: 2
  Transaction confirmed   Block: 15845753   Gas used: 50506 (0.01%)

Listing on index 1000 --> 0x637bb258a08b465eddde1efb09b67f6b29ccd5e31b44e37f969a57172c04e1ab

Filling listing with 500 pods --> testTx = contract_MarketplaceFacet.fillPodListing((user2.address, 1000, 0, 1000, 10**6, 1000*10**6, 1), 500, 0, {'from': user1})
Transaction sent: 0x480a5f5f8b0138c7c8f42989765a5a3efaac72bdb9e79b30dae9e68ccf84db5e
  Gas price: 0.0 gwei   Gas limit: 600000000   Nonce: 2
  Transaction confirmed   Block: 15845754   Gas used: 148554 (0.02%)


After, listing will be transferred to index 1500 --> contract_MarketplaceFacet.podListing(1500) -->0x7a2928c2e16069f9f75f5008ca312a241975eee98d81fcad51f0955dbc95aa3b

Reverting chain...

Creating same Pod Listing again on index 1000--> contract_MarketplaceFacet.createPodListing(1000, 0, 500, 10**6, 1000*10**6, 1, {'from': user2})
Transaction sent: 0xf090399f50ecbc1a1f269e2286f8572412398760e73804334fa803d5009f4bce
  Gas price: 0.0 gwei   Gas limit: 600000000   Nonce: 2
  Transaction confirmed   Block: 15845753   Gas used: 50506 (0.01%)


Filling listing with 0 pods --> testTx2 = contract_MarketplaceFacet.fillPodListing((user2.address, 1000, 0, 500, 10**6, 1000*10**6, 1), 0, 0, {'from': user1})
Transaction sent: 0xaed66164bc769d400a8c3c872595417ace8e947220d215ade036390f1bd06b29
  Gas price: 0.0 gwei   Gas limit: 600000000   Nonce: 2
  Transaction confirmed   Block: 15845754   Gas used: 31112 (0.01%)


The listing has been deleted from index 1000 --> contract_MarketplaceFacet.podListing(1000) --> 0x0000000000000000000000000000000000000000000000000000000000000000
```

Risk Level:

Likelihood - 3
Impact - 3

Recommendation:

It is recommended first to delete the original listing when it gets partially fulfilled and then create the new one containing the remaining pods. This way, it can be assured that the new listing will not be deleted in case it is created in the same index as the previous one (listings with 0 start parameters and filled with 0 beanAmount).

Remediation Plan:

SOLVED: The Beanstalk team solved the issue by switching the order in which the new listing is created, and the original one is removed, ensuring that it does not get deleted.

Commit ID: b6a567d842e72c73176099ffd8ddb04cae2232e6

3.4 (HAL-04) PLOTS CAN BE UNCONTROLLABLY SPLITTED - LOW

Description:

Note that this finding was also reported on **Pod Market V2** audit, and it has been already fixed by the Beanstalk team.

As described in the previous finding, the Marketplace can be used to buy and sell pods, and listings or orders can be partially filled. When an order or listing is partially filled, the pods contained on each plot are split to be able to assign the acquired pods to the buyer.

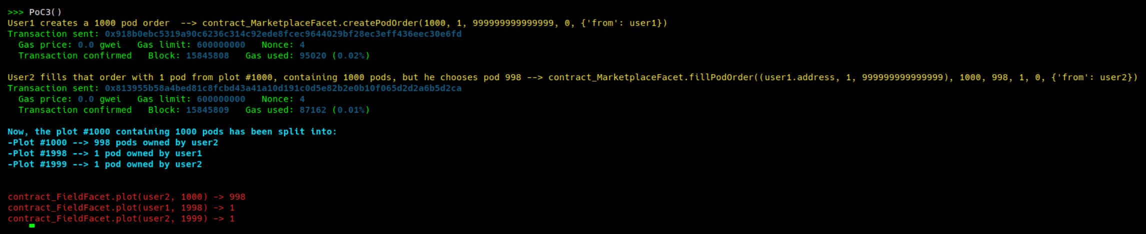
However, it has been detected that there is no limit on the granularity in which the plots can be split. This allows any griefer to fill any listing or orders with the minimal amount of beanAmount allowed by the data type (1), which would cause, in the case of orders, the buyer would end with a large amount of tiny plots, which would be extremely uncomfortable to manage.

This could also naturally happen without needing a griefer. If any user creates a large order that many users partially fulfill, that will end up in many sub-plots, which would have to be separately sold, harvested, etc. This also means that gas costs would be increased.

Proof of Concept:

For this PoC, user1 will create a 1000 pods orders. Thereafter, the user2 user will partially fill that listing with 1 pod from his plot on index 1000, but he will choose 998 as the first pod.

Subsequently, the original plot will be split into 3 subplots now with a single order fill:



```
>>> PoC3()
User1 creates a 1000 pod order  --> contract_MarketplaceFacet.createPodOrder(1000, 1, 999999999999999, 0, {'from': user1})
Transaction sent: 0x918b0ebc5319a90c6236c314c92ede8fcec9644029bf28ec3eff436eec30e6fd
  Gas price: 0.0 gwei   Gas limit: 600000000   Nonce: 4
  Transaction confirmed   Block: 15845808   Gas used: 95020 (0.02%)

User2 fills that order with 1 pod from plot #1000, containing 1000 pods, but he chooses pod 998 --> contract_MarketplaceFacet.fillPodOrder((user1.address, 1, 999999999999999), 1000, 998, 1, 0, {'from': user2})
Transaction sent: 0x813955b58a4bed81c8fcbd43a41a10d191c0d5e82b2e0b10f065d2d2a6b5d2ca
  Gas price: 0.0 gwei   Gas limit: 600000000   Nonce: 4
  Transaction confirmed   Block: 15845809   Gas used: 87162 (0.01%)

Now, the plot #1000 containing 1000 pods has been split into:
-Plot #1000 --> 998 pods owned by user2
-Plot #1998 --> 1 pod owned by user1
-Plot #1999 --> 1 pod owned by user2


contract_FieldFacet.plot(user2, 1000) -> 998
contract_FieldFacet.plot(user1, 1998) -> 1
contract_FieldFacet.plot(user2, 1999) -> 1
```

Suppose this gets repeated over time (intentionally or unintentionally). In that case, it will result in many plots containing a few pods each, which would significantly increase management gas costs.

Risk Level:

Likelihood - 2
Impact - 2

Recommendation:

It is recommended to introduce a parameter that defines the minimum fill amount for orders and listings to prevent plots from being split into smaller than desired subplots.

Remediation Plan:

SOLVED: The Beanstalk team fixed the issue by adding a minFillAmount parameter in listings and orders to allow users to control the minimum desired plot size.

Commit ID: bd26a50db10af2284df73be08c79f53df41c49ce

3.5 (HAL-05) 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:

CurvePrice.sol

- Line 77:

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

- Line 84:

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

- Line 86:

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

BeanstalkPrice.sol

- Line 21:

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

LibPlainCurveConvert.sol

- Line 79:

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

LibDiamond.sol

- Line 104:

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

- Line 129:

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

- Line 147:

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

- Line 162:

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

SiloFacet.sol

- Line 108:

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

DiamondLoupeFacet.sol

- Line 32:

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

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:

ACKNOWLEDGED: The Beanstalk team acknowledged this issue.

3.6 (HAL-06) 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:

CurvePrice.sol
- Line 77:

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

- Line 84:

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

- Line 86:

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

BeanstalkPrice.sol
- Line 21:

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

LibPlainCurveConvert.sol
- Line 79:

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

SiloFacet.sol
- Line 108:

```
for (uint256 i = 0; i < amounts.length; 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:

ACKNOWLEDGED: The Beanstalk team acknowledged this issue.

3.7 (HAL-07) USAGE OF AND OPERATOR IN REQUIRE STATEMENTS - INFORMATIONAL

Description:

Instead of using the && operator in a single require statement to check multiple conditions, using multiple require statements with one condition per require statement will save 8 GAS per condition.
The gas difference would only be materialized if the revert condition is met.

Code Location:

Listing 7: LibTokenSilo.sol (Line 107)

```
106             require(
107                 newBase <= uint128(-1) && newAmount <= uint128(-1),
108                 "Silo: uint128 overflow."
109             );
```

Listing 8: MarketplaceFacet.sol (Lines 129,134)

```
128         require(
129             sender != address(0) && recipient != address(0),
130             "Field: Transfer to/from 0 address."
131         );
132         uint256 amount = s.a[sender].field.plots[id];
133         require(amount > 0, "Field: Plot not owned by user.");
134         require(end > start && amount >= end, "Field: Pod range invalid.");
135
```

Listing 9: Listing.sol (Line 60)

```
59          require(
60              plotSize >= start.add(amount) && amount > 0,
61              "Marketplace: Invalid Plot/Amount."
62          );
```

Listing 10: Listing.sol (Line 111)

```
110         require(
111             plotSize >= (l.start + l.amount) && l.amount > 0,
112             "Marketplace: Invalid Plot/Amount."
113         );
```

Listing 11: FieldFacet.sol (Line 48)

```
47          require(
48              sowAmount >= minAmount && amount >= minAmount &&
 ↳  minAmount > 0,
49              "Field: Sowing below min or 0 pods."
50          );
```

Listing 12: FieldFacet.sol (Line 50)

```
47          require(
48              sowAmount >= minAmount && amount >= minAmount &&
 ↳  minAmount > 0,
49              "Field: Sowing below min or 0 pods."
50          );
```

Listing 13: CurveFacet.sol (Line 370)

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

Listing 14: CurveFacet.sol (Line 387)

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

Listing 15: ConvertFacet.sol (Line 144)

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

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

If possible, it is recommended to split the different conditions on each require statement into different statements to both improve code readability and save gas.

Remediation Plan:

ACKNOWLEDGED: The Beanstalk team acknowledged this issue.

3.8 (HAL-08) OPTIMIZE UNSIGNED INTEGER COMPARISON - INFORMATIONAL

Description:

The check != 0 costs less gas compared to > 0 for unsigned integers in require statements with the optimizer enabled. While it may seem that > 0 is cheaper than !=0, this is only true without the optimizer enabled and outside a require statement. If the optimizer is enabled at 10k and it is in a require statement, that would be more gas efficient.

Risk Level:

Likelihood - 1
Impact - 1

Code Location:

ConvertFacet.sol
- Line 144:

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

FieldFacet.sol
- Line 48:

```
sowAmount >= minAmount && amount >= minAmount && minAmount > 0,
```

- Line 93:

```
require(pods > 0, "Field: Plot is empty.");
```

FundraiserFacet.sol
- Line 46:

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

Listing.sol
- Line 60:

```
plotSize >= start.add(amount)&& amount > 0,
```

- Line 64:

```
pricePerPod > 0,
```

- Line 111:

```
plotSize >= (l.start + l.amount)&& l.amount > 0,
```

- Line 151:

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

MarketplaceFacet.sol

- Line 133:

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

Order.sol

- Line 62:

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

SiloFacet.sol

- Line 107:

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

- Line 109:

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

Recommendation:

Consider changing `> 0` comparison with `!= 0`.

Remediation Plan:

ACKNOWLEDGED: The Beanstalk team acknowledged this issue.

3.9 (HAL-09) INCOMPLETE NATSPEC DOCUMENTATION - INFORMATIONAL

Description:

Natspec documentation are useful for internal developers that need to work on the project, external developers that need to integrate with the project, auditors that have to review it but also for end users given that many chain explorers have officially integrated the support for it directly on their site.

It has been detected that, while many contracts have a complete **natspec** documentation, other contracts are little to no documented.

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

Consider adding the missing **natspec** documentation.

Remediation Plan:

ACKNOWLEDGED: The Beanstalk team acknowledged this issue.

3.10 (HAL-10) INCREASE OPTIMIZER RUNS - INFORMATIONAL

Description:

Solidity 0.8.7 has a good optimizer that saves a gas when compiling to bytecode. The team can use it by increasing the number of runs to something like **2000** at least in the config.

Risk Level:

Likelihood - 1
Impact - 1

Code Location:

Config File

Listing 16

```
solidity: {
  version: "0.7.6",
  settings: {
    optimizer: {
      enabled: true,
      runs: 1000
    }
  }
},
```

Recommendation:

Consider increasing optimizer runs.

Remediation Plan:

ACKNOWLEDGED: The Beanstalk team acknowledged this issue.

3.11 (HAL-11) SOLC 0.7.6 COMPILER VERSION CONTAINS MULTIPLE BUGS - INFORMATIONAL

Description:

The scoped contracts have configured the Solidity version to 0.7.6 in Hardhat configuration file. The latest solidity compiler version, 0.8.17 , fixed important bugs in the compiler along with new native protections, such as the arithmetic checks now performed by default, preventing plausible under/overflows. The current version is missing the following fixes: 0.8.0, 0.8.1, 0.8.2, 0.8.3, 0.8.4, 0.8.5, 0.8.6, 0.8.7, 0.8.8, 0.8.9, 0.8.12, 0.8.13, 0.8.14, 0.8.15, 0.8.16, 0.8.17.

The official Solidity recommendations are that you should use the latest released version of Solidity when deploying contracts. Apart from exceptional cases, only the most recent version receives security fixes.

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

It is recommended to use the latest Solidity compiler version as possible.

Remediation Plan:

ACKNOWLEDGED: The Beanstalk team acknowledged this issue.

3.12 (HAL-12) OUT OF DATE OPENZEPPELIN PACKAGES - INFORMATIONAL

Description:

The OpenZeppelin packages used are out of date, it is good practice to use the latest version of these packages. Please note that this finding only will apply if Solidity version is upgraded from 0.7.6, since OpenZeppelin's 3.4.0 is the last version supporting Solidity <0.8.0.

Risk Level:

Likelihood - 1
Impact - 2

Code Location:

Package Json

Listing 17

```
"dependencies": {
  "@openzeppelin/contracts": "^3.4.0",
  "@openzeppelin/contracts-upgradeable": "^3.4.0",
  "dotenv": "^10.0.0",
  "eth-permit": "^0.2.1",
  "keccak256": "^1.0.6",
  "merkletreejs": "^0.2.31"
}
```

Recommendation:

Update the versions of @openzeppelin/contracts and @openzeppelin/contracts-upgradeabl to be the latest in package.json. It is also recommended to checking the versions of other dependencies as a precaution, as they may include important bug fixes.

Remediation Plan:

ACKNOWLEDGED: The Beanstalk team acknowledged this issue.

THANK YOU FOR CHOOSING

HALBORN