Beanstalk - Pod Market V2

Smart Contract Security Audit

Prepared by: **Halborn**

Date of Engagement: **September 26th, 2022 - October 10th, 2022**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	09/26/2022	Roberto Reigada
0.2	Document Updates	10/10/2022	Roberto Reigada
0.3	Draft Review	10/12/2022	Gabi Urrutia
1.0	Remediation Plan	10/17/2022	Roberto Reigada
1.1	Remediation Plan Review	10/18/2022	Gabi Urrutia
1.2	Remeditation Plan Update	10/27/2022	Francisco González
1.3	Remediation Plan Review	10/28/2022	Kubilay Onur Gungor
1.4	Remediation Plan Review	10/28/2022	Gabi Urrutia
1.5	Remeditation Plan Update	11/04/2022	Francisco González
1.6	Remediation Plan Review	11/04/2022	Kubilay Onur Gungor
1.7	Remediation Plan Review	11/04/2022	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
Roberto Reigada	Halborn	Roberto.Reigada@halborn.com
Francisco González	Halborn	Francisco.Villarejo@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

Beanstalk engaged Halborn to conduct a security audit on their Pod Market V2 smart contracts beginning on September 26th, 2022 and ending on October 10th, 2022. The security assessment was scoped to the smart contracts provided in the GitHub repository BeanstalkFarms/Beanstalk/tree/Pod-Pricing-Functions.

The assessment also included the confirmation of a critical security finding identified by the Beanstalk team. Halborn confirmed the issue and verified the fix applied by the Beanstalk team during the remediation phase. For more details about the finding and remediation, please see:
- HAL01 - INCORRECT POD PRICE CALCULATION WHEN USING PRICE FUNCTIONS CONTAINING NEGATIVE POLYNOMIAL COEFFICIENTS.

1.2 AUDIT SUMMARY

The team at Halborn was provided 2 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 a few security risks that were addressed by the Beanstalk team. The critical security finding was identified and fixed by the Beanstalk team, and Halborn verified the fix.

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 all the changes performed related to the new Pod Market V2. The contracts that were affected by this change were:

- LibPolynomial.sol
- LibBytes.sol
- MarketplaceFacet.sol
- Listing.sol
- Order.sol
- LibSiloPermit.sol
- AppStorage.sol
- SiloFacet.sol
- TokenFacet.sol
- LibTokenApprove.sol
- LibTokenPermit.sol
- LibTransfer.sol

Initial Commit ID:
- d2a9a232f50e1d474d976a2e29488b70c8d19461

Fixed Commit ID 1:
- e1f74ae6e87df0911148e9b5c74403326ab92ba4

Fixed Commit ID 2:
- b6a567d842e72c73176099ffd8ddb04cae2232e6

Fixed Commit ID 3:
- 0bdd376263b0fe94af84aaf4adb6391b39fa80ab

Changes from Fixed Commit ID 2:
LibPolynomial.sol:
- Fixed polynomial integration calculation bug which caused negativeSum component to be zeroed when a price function containing negative polynomial coefficient was used.

Changes from Fixed Commit ID 1:

Listing.sol:

- Modified the order of deleting and creating new listings when a list is partially filled to prevent listing overwriting.
- Added minFillAmount parameter to listings and orders to minimize plot splitting, allowing the user who creates the listing or the order to specify the minimal amount to be filled.

MarketplaceFacet.sol:

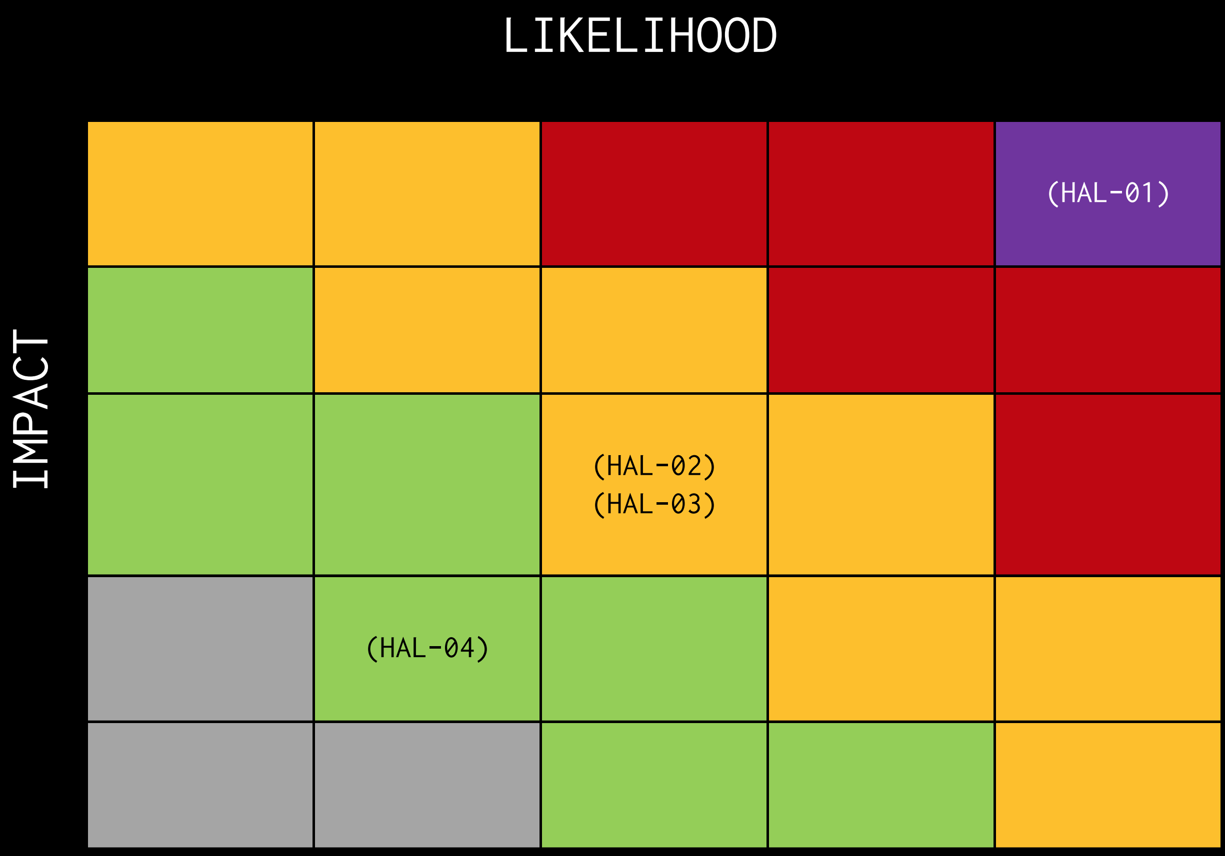
- Added minFillAmount parameter to listings and orders to minimize plot splitting, allowing the user who creates the listing or the order to specify the minimal amount to be filled.

Order.sol:

- Added minFillAmount parameter to listings and orders to minimize plot splitting, allowing the user who creates the listing or the order to specify the minimal amount to be filled.

2. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
1	0	2	1	0



SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL01 - INCORRECT POD PRICE CALCULATION WHEN USING PRICE FUNCTIONS CONTAINING NEGATIVE POLYNOMIAL COEFFICIENTS	Critical	SOLVED - 11/04/2022
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

FINDINGS & TECH DETAILS

3.1 (HAL-01) INCORRECT POD PRICE CALCULATION WHEN USING PRICE FUNCTIONS CONTAINING NEGATIVE POLYNOMIAL COEFFICIENTS - CRITICAL

Description:

Note that the finding was identified by the Beanstalk team.

Pod Marketplace V2 introduces a new feature that allows users to create and fill pod listings and orders using polynomial price functions.

When any user fills an order (sell his Pods to an active offer) by calling `fillPodOrderV2()`, the number of **BEAN** that the seller will receive is calculated by calling `getAmountBeansToFillOrderV2()`, which evaluates the polynomial integration of the price function between A and B boundaries, which are defined by the place in line of the Pods that are being sold.

To calculate this polynomial integration, the function `evaluatePolynomialIntegration()`, contained in `LibPolynomial.sol` uses the second fundamental theorem of calculus, which defines the area contained under any curve defined by a continuous function f(x) between two points A and B as the difference of the antiderivative function of f(x), F(x), evaluated in B and A:

$$\int_a^b f(x)\,dx = F(b) - F(a).$$

To implement that, `evaluatePolynomialIntegration()` splits the integration into two components, `positiveSum` and `negativeSum`, being `positiveSum` defined as the sum of the antiderivative functions associated to each positive term on f(x) evaluated between upper and lower integration limits, and `negativeSum` being the same, but for negative coefficients.

Once these two components have been calculated, the final result of the integration will be calculated as positiveSum - negativeSum.

However, it was detected that, due to an error on evaluatePolynomialIntegration() function, negativeSum was being calculated as the sum of the antiderivative functions associated to each negative term on f(x), but picking the same value as upper and lower integration limit, causing negativeSum to be always zero.
This caused the result returned by evaluatePolynomialIntegration() to be much higher than it should be if the pricing function used contained any negative term.
This excesively high result would translate into a higher value of beanAmount obtained in getAmountBeansToFillOrderV2, and it would mean that the seller could sell any amount of beans at a much higher price than the buyer intended to pay in the first place. Ultimately, this would end up in having the **BEAN** deposited in any order drained, and the buyer receiving way less Pods than it should have received, rendering the order non-fillable and the **BEAN** deposited by the buyer lost.

Code Location:

Listing 1: LibPolynomial.sol (Line 130)

```
    /**
     * @notice Computes the integral of a cubic polynomial
     * @dev Polynomial is of the form a(x-k)^3 + b(x-k)^2 + c(x-k)
       + d where k is the start of the piecewise interval
     * @param f The encoded piecewise polynomial
     * @param pieceIndex Which piece of the polynomial to evaluate
     * @param numPieces The number of pieces in the polynomial
     * @param start The lower bound of the integration. (can
       overflow past 10e13)
     * @param end The upper bound of the integration. (can overflow
       past 10e13)
     */
    function evaluatePolynomialIntegration(
        bytes calldata f,
        uint256 pieceIndex,
        uint256 numPieces,
        uint256 start, //start of breakpoint is assumed to be
```

```
↳ subtracted
        uint256 end //start of breakpoint is assumed to be
↳ subtracted
    ) internal pure returns (uint256) {
        uint256 positiveSum;
        uint256 negativeSum;

        uint256[4] memory significands = getSignificands(f,
↳ pieceIndex, numPieces);
        uint8[4] memory exponents = getExponents(f, pieceIndex,
↳ numPieces);
        bool[4] memory signs = getSigns(f, pieceIndex, numPieces);

        for(uint256 degree; degree <= MAX_DEGREE; ++degree) {

            if(signs[degree]) {
                //uint256 max value is 1e77 and this has been
↳ tested to work to not overflow for values less than 1e14.
                //Note: susceptible to overflows past 1e14
                positiveSum = positiveSum.add(pow(end, 1 + degree)
↳ .mul(significands[degree]).div(pow(10, exponents[degree]).mul(1 +
↳ degree)));

                positiveSum = positiveSum.sub(pow(start, 1 +
↳ degree).mul(significands[degree]).div(pow(10, exponents[degree]).
↳ mul(1 + degree)));
            } else {
                negativeSum = negativeSum.add(pow(end, 1 + degree)
↳ .mul(significands[degree]).div(pow(10, exponents[degree]).mul(1 +
↳ degree)));

                negativeSum = negativeSum.sub(pow(end, 1 + degree)
↳ .mul(significands[degree]).div(pow(10, exponents[degree]).mul(1 +
↳ degree)));
            }
        }

        return positiveSum.sub(negativeSum);
    }
```

Proof of Concept:

For this PoC, two calls to getAmountBeansToFillOrderV2() and getAmountBeansToFillOrderV2Wrong() will be performed. getAmountBeansToFillOrderV2() is the original function with the fixes proposed in **Recommendation** section applied, and getAmountBeansToFillOrderV2Wrong() is a function defined for this PoC which contains the incorrect integration calculation, zeroing negativeSum.

A random pricing function has been defined:

Listing 2: Pricing function points

```
const set_13Pieces = {
  xs: [1000 , 5000 , 6000 , 7000 , 8000 , 9000 , 10000 , 11000 , 12000 , 13000 , 14000 , 18000 , 20000 ],
  ys: [1000, 2000, 4000, 9000, 15000, 30000, 65000, 100000, 105000, 120000, 750000, 1000000, 2000000]
}
```

Then, a test that calls both version of getAmountBeansToFillOrderV2() is defined:

Listing 3: getAmountBeansToFillOrderV2() testing

```
        describe("Comparing old and fixed integration calculation logic", async function () {
          beforeEach(async function () {
            this.f = interpolatePoints(hugeValueSet_13Pieces.xs, hugeValueSet_13Pieces.ys);
          })

          it("fixed logic", async function () {
            const startPlaceInLine = 10000000000000;
            const amountPodsFromOrder = 185000000000000;
            const orderBeanAmount = getAmountOrder(this.f, startPlaceInLine, amountPodsFromOrder);
            expect(await this.marketplace.connect(user).getAmountBeansToFillOrderV2(startPlaceInLine, amountPodsFromOrder, this.f.packedFunction)).to.be.equal(orderBeanAmount);
            console.log(await this.marketplace.connect(user).
```

```
↳ getAmountBeansToFillOrderV2(startPlaceInLine, amountPodsFromOrder,
↳  this.f.packedFunction));
          })

          it("old logic", async function () {
            const startPlaceInLine = 1000000000000;
            const amountPodsFromOrder = 185000000000000;
            const orderBeanAmount = getAmountOrder(this.f,
↳ startPlaceInLine, amountPodsFromOrder);
            expect(await this.marketplace.connect(user).
↳ getAmountBeansToFillOrderV2(startPlaceInLine, amountPodsFromOrder,
↳  this.f.packedFunction)).to.be.equal(orderBeanAmount);
            console.log(await this.marketplace.connect(user).
↳ getAmountBeansToFillOrderV2Wrong(startPlaceInLine,
↳ amountPodsFromOrder, this.f.packedFunction));
          })
        })
```

When executed, and as expected, the **BEAN** amount needed to pay for the same amount of Pods is higher when using the same old logic, meaning that the buyer would lose these **BEAN** difference because of the miscalculation:

```
          Comparing old and fixed integration calculation logic
BigNumber { value: "134484074236257" }
           ✓ fixed logic
BigNumber { value: "139081829083511" }
           ✓ old logic
```

The amount of **BEANS** lost would entirely depend on the pricing function set by the seller and the position in the line of the Pods being sold.

Risk Level:

Likelihood - 5
Impact - 5

Recommendation:

It is recommended to fix the calculation of negativeSum by using start integration limit on L#130.

Remediation Plan:

SOLVED: The Beanstalk team solved the issue by using the lower integration limit on L#130, preventing negativeSum from being zeroed.

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

Description:

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

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

Listing 4: Listing.sol (Line 68)

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

```
        mode
    );

    emit PodListingCreated(
        msg.sender,
        index,
        start,
        amount,
        pricePerPod,
        maxHarvestableIndex,
        mode
    );
}
```

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 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 }
```

On the other hand, a similar issue occurs in:

`Listing.sol`
- Line 92:
`require(plotSize >= (start + amount)&& amount > 0, "Marketplace: Invalid Plot/Amount.");`

- Line 134:

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

- Line 162:

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

- Line 251:

```
uint256 pricePerPod = LibPolynomial.evaluatePolynomialPiecewise(
pricingFunction, l.index + l.start - s.f.harvestable);
```

Order.sol

- Line 98:

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

- Line 99:

```
require((index + start - s.f.harvestable + amount)<= o.maxPlaceInLine,
"Marketplace: Plot too far in line.");
```

- Line 125:

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

- Line 126:

```
require((index + start - s.f.harvestable + amount)<= o.maxPlaceInLine,
"Marketplace: Plot too far in line.");
```

- Line 129:

```
uint256 costInBeans = getAmountBeansToFillOrderV2(index + start - s.f.
harvestable, amount, pricingFunction);
```

- Line 190:

```
beanAmount = LibPolynomial.evaluatePolynomialIntegrationPiecewise(
pricingFunction, placeInLine, placeInLine + amountPodsFromOrder);
```

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.

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

Description:

MarketplaceFacet.soland 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
130     ) private {
131         // Note: If l.amount < amount, the function roundAmount
    ↳ will revert
132
```

```
        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.

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

Description:

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 different users partially fulfill, that will end up in many different 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.

MANUAL TESTING

4.1 INTRODUCTION

Halborn performed different manual tests in all the different Facets of the Beanstalk protocol, trying to find any logic flaws and vulnerabilities.

During the manual tests, the following areas were reviewed carefully:

1. Hash collisions in the Marketplace listing/orders.
2. Signature replay attacks related to the new deposit Permits implementation.
3. `receiveToken()` function calls.

4.2 TESTING

MARKETPLACE LISTING/ORDERS: HASH COLLISIONS:

In the MarketplaceFacet the POD listings hashes are built this way:

Listing 7: Listing.sol (Lines 265,277)

```
258 function hashListing(
259     uint256 start,
260     uint256 amount,
261     uint24 pricePerPod,
262     uint256 maxHarvestableIndex,
263     LibTransfer.To mode
264 ) internal pure returns (bytes32 lHash) {
265     lHash = keccak256(abi.encodePacked(start, amount, pricePerPod,
  ↳ maxHarvestableIndex,  mode == LibTransfer.To.EXTERNAL));
266 }
267
268 function hashListingV2(
269     uint256 start,
270     uint256 amount,
271     uint24 pricePerPod,
272     uint256 maxHarvestableIndex,
273     bytes calldata pricingFunction,
274     LibTransfer.To mode
275 ) internal pure returns (bytes32 lHash) {
276     require(pricingFunction.length == LibPolynomial.getNumPieces(
  ↳ pricingFunction).mul(168).add(32), "Marketplace: Invalid pricing
  ↳ function.");
277     lHash = keccak256(abi.encodePacked(start, amount, pricePerPod,
  ↳ maxHarvestableIndex, mode == LibTransfer.To.EXTERNAL,
  ↳ pricingFunction));
278 }
```

On the other hand, the orders are built as shown below:

Listing 8: Order.sol (Lines 202,212)

```
197     function createOrderId(
198         address account,
199         uint24 pricePerPod,
```

```
        uint256 maxPlaceInLine
    ) internal pure returns (bytes32 id) {
        id = keccak256(abi.encodePacked(account, pricePerPod, maxPlaceInLine));
    }

    function createOrderIdV2(
        address account,
        uint24 pricePerPod,
        uint256 maxPlaceInLine,
        bytes calldata pricingFunction
    ) internal pure returns (bytes32 id) {
        require(pricingFunction.length == LibPolynomial.getNumPieces(pricingFunction).mul(168).add(32), "Marketplace: Invalid pricing function.");
        id = keccak256(abi.encodePacked(account, pricePerPod, maxPlaceInLine, pricingFunction));
    }
```

For both cases, taking into consideration how orders and listings hashes are built, there is no way to intentionally create, for example, a different order with the same hash in order to steal the Beans sent in a previous order. 2^256 (the number of possible keccak-256 hashes) is around the number of atoms in the known observable universe. With the current code, a collision would be as unlikely as picking two atoms at random and having them turn out to be the same.

DEPOSIT PERMITS: SIGNATURE REPLAY ATTACKS:

The deposit permits were implemented with the following code:

Listing 9: LibSiloPermit.sol (Lines 36,53)

```
25 function permit(
26     address owner,
27     address spender,
28     address token,
29     uint256 value,
30     uint256 deadline,
31     uint8 v,
32     bytes32 r,
33     bytes32 s
34 ) internal {
35     require(block.timestamp <= deadline, "Silo: permit expired deadline");
36     bytes32 structHash = keccak256(abi.encode(DEPOSIT_PERMIT_TYPEHASH, owner, spender, token, value, _useNonce(owner), deadline));
37     bytes32 hash = _hashTypedDataV4(structHash);
38     address signer = ECDSA.recover(hash, v, r, s);
39     require(signer == owner, "Silo: permit invalid signature");
40 }
41
42 function permits(
43     address owner,
44     address spender,
45     address[] memory tokens,
46     uint256[] memory values,
47     uint256 deadline,
48     uint8 v,
49     bytes32 r,
50     bytes32 s
51 ) internal {
52     require(block.timestamp <= deadline, "Silo: permit expired deadline");
53     bytes32 structHash = keccak256(abi.encode(DEPOSITS_PERMIT_TYPEHASH, owner, spender, keccak256(abi.encodePacked(tokens)), keccak256(abi.encodePacked(values)), _useNonce(owner), deadline));
54     bytes32 hash = _hashTypedDataV4(structHash);
55     address signer = ECDSA.recover(hash, v, r, s);
56     require(signer == owner, "Silo: permit invalid signature");
```

```
57 }
```

ECDSA library was used following the best practices. This library prevents any kind of signature malleability attack. On the other hand, the signatures use a domain separator which is built with the chain.id, the Beanstalk smart contract address and other parameters like name, version... This totally prevents any kind of cross-chain signature replay attacks.

Lastly, it is known that using abi.encodePacked() with dynamic parameters is vulnerable to hash collisions. Any attack vector related to this was very well prevented in the following line by doing a keccak256 hash of the tokens and values arrays:

```
bytes32 structHash = keccak256(abi.encode(DEPOSITS_PERMIT_TYPEHASH,
owner, spender, keccak256(abi.encodePacked(tokens)), keccak256(abi.
encodePacked(values)), _useNonce(owner), deadline));
```

RECEIVETOKEN FUNCTION CALLS:

If the receiveToken() call return value is not checked, users can abuse this by using the INTERNAL_TOLERANT fromMode. Every receiveToken() call was checked carefully and all of them are considering its return value.

THANK YOU FOR CHOOSING

HALBORN