HALBORN

Beanstalk - Pod Marketplace 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

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 Pod Marketplace 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.

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 one security risk that was 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.

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 Marketplace V2. The contracts that were affected by this change were:

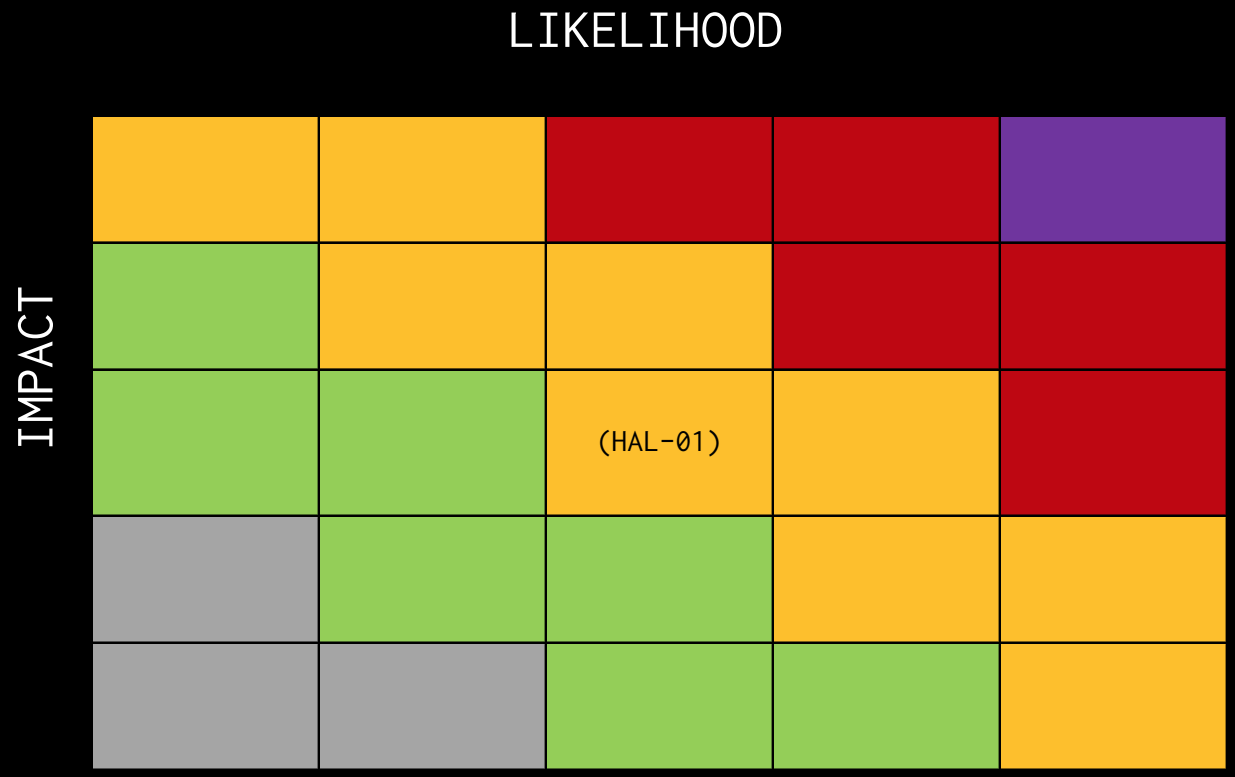
- LibPolynomial.sol
- LibBytes.sol
- MarketplaceFacet.sol
- Listing.sol
- Order.sol
- LibSiloPermit.sol
- AppStorage.sol
- SiloFacet.sol
- TokenFacet.sol
- LibTokenApprove.sol
- LibTokenPermit.sol
- LibTransfer.sol

Initial Commit ID:
- d2a9a232f50e1d474d976a2e29488b70c8d19461

Fixed Commit ID:
- e1f74ae6e87df0911148e9b5c74403326ab92ba4

2. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	1	0	0



EXECUTIVE OVERVIEW

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL01 - MULTIPLE UNDERFLOWS/OVERFLOWS	Medium	SOLVED - 10/17/2022

FINDINGS & TECH DETAILS

3.1 (HAL-01) 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 1: 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,
```

```
87          mode
88      );
89
90      emit PodListingCreated(
91          msg.sender,
92          index,
93          start,
94          amount,
95          pricePerPod,
96          maxHarvestableIndex,
97          mode
98      );
99  }
```

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 2: 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, 11579208923731619542357098500868790785326998466564056403945758400791312963993S, 5_000000, 0, 1, {'from': user1, 'value': 0})
Transaction sent: 0xa528f2d81216f5477d43566344262280b32de2b291dc3c7bd5d379e89309e8172
  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: 0xcb589a9d7c96a4854647ac870ce52bc246eaed2bb7c7440e97bac05b801d188
  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, 11579208923731619542357098500868790785326998466564056403945758400791312963993S, 5_000000, 0, 1), 2510, 0, {'from': user3, 'value': 0})
Transaction sent: 0x493294792 6d500ffc2a1c2d429df67e92ac6d4c53f65dd99cec3289dba05a4bc
  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:

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:

It is recommended to make use of the SafeMath library in all the code lines described above.

Remediation Plan:

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

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 3: 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 4: Order.sol (Lines 202,212)

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

```
200         uint256 maxPlaceInLine
201     ) internal pure returns (bytes32 id) {
202         id = keccak256(abi.encodePacked(account, pricePerPod,
 ↳ maxPlaceInLine));
203     }
204
205     function createOrderIdV2(
206         address account,
207         uint24 pricePerPod,
208         uint256 maxPlaceInLine,
209         bytes calldata pricingFunction
210     ) internal pure returns (bytes32 id) {
211         require(pricingFunction.length == LibPolynomial.
 ↳ getNumPieces(pricingFunction).mul(168).add(32), "Marketplace:
 ↳ Invalid pricing function.");
212         id = keccak256(abi.encodePacked(account, pricePerPod,
 ↳ maxPlaceInLine, pricingFunction));
213     }
```

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 5: 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 crosschain 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