Beanstalk

Fix Review

July 22, 2022

Prepared for:
Publius
Beanstalk

Prepared by: **Jaime Iglesias and Bo Henderson**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 80+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

Table of Contents

Executive Summary

Engagement Overview

Beanstalk engaged Trail of Bits to review the security of its Beanstalk protocol. Specifically, Trail of Bits reviewed the state of the protocol during the Barn Raise, a community fundraiser intended to recapitalize the protocol after an attack in April of 2022, resulting in the loss of approximately $77 million in assets. From June 2 to July 6, 2022, a team of two consultants conducted a security review of the client-provided source code, with eight person-weeks of effort. Details of the project's scope, timeline, test targets, and coverage are provided in the original audit report.

Beanstalk contracted Trail of Bits to review the fixes implemented for issues identified in the original report. From July 12 to July 14, 2022, one consultant conducted a review of the client-provided source code, with three person-days of effort.

Summary of Findings

The original audit uncovered significant flaws that could impact system confidentiality, integrity, or availability. A summary of the original findings is provided below.

EXPOSURE ANALYSIS

Severity	*Count*
High	3
Medium	3
Low	1
Informational	3
Undetermined	3

CATEGORY BREAKDOWN

Category	*Count*
Data Validation	8
Economic	3
Undefined Behavior	2

Overview of Fix Review Results

Beanstalk has sufficiently addressed most of the issues described in the original audit report. The Beanstalk team has acknowledged and accepted the risks associated with four of the issues reported, including an informational-severity issue regarding the use of a Solidity optimizer and three economic/governance issues; the team provided comments describing the rationale for its acceptance of the risks associated with the economic/governance issues. All other issues have been sufficiently fixed.

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Anne Marie Barry, Project Manager
annemarie.barry@trailofbits.com

The following engineers were associated with this project:

Jaime Iglesias, Consultant
jaime.iglesias@trailofbits.com

Bo Henderson, Consultant
bo.henderson@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
June 2, 2022	Pre-project kickoff call
June 13, 2022	Status update meeting #1
June 17, 2022	Status update meeting #2
June 24, 2022	Status update meeting #3
July 6, 2022	Delivery of report draft
July 7, 2022	Report readout meeting
July 22, 2022	Delivery of final report
July 22, 2022	Delivery of fix review

Project Methodology

Our work in the fix review included the following:

- A review of the findings in the original audit report
- A manual review of the client-provided source code and configuration material
- A check for any updates to the documentation and the unit test suite that Beanstalk may have made after the completion of the original audit
  - In terms of documentation, we found that Beanstalk updated the public protocol documentation. However, the documentation is still a work in progress and does not yet feature an adequate glossary.
  - In terms of the unit test suite, we found that Beanstalk added new tests, many of which will prevent issues similar to those reported in the original audit from being (re)introduced. The team also commented out certain tests, some of which may have required updates following the team's fixes to certain issues. We recommend reimplementing these commented-out tests, fixing them, and running them alongside the others as part of an automated process.

Project Targets

The engagement involved a review of the fixes implemented in the following target.

Beanstalk

Repository	https://github.com/BeanstalkFarms/Beanstalk-Replanted
Version	`9422ad60cbb4ece7cfb4f0925c4586fb4582e7df`
Type	Solidity
Platform	EVM

Summary of Fix Review Results

The table below summarizes each of the original findings and indicates whether the issue has been sufficiently resolved.

ID	Title	Status
1	Attackers could mint more Fertilizer than intended due to an unused variable	Resolved
2	Lack of a two-step process for ownership transfer	Resolved
3	Possible underflow could allow more Fertilizer than MAX_RAISE to be minted	Resolved
4	Risk of Fertilizer id collision that could result in loss of funds	Resolved
5	The sunrise() function rewards callers only with the base incentive	Resolved
6	Solidity compiler optimizations can be problematic	Unresolved
7	Lack of support for external transfers of nonstandard ERC20 tokens	Resolved
8	Plot transfers from users with allowances revert if the owner has an existing pod listing	Resolved
9	Users can sow more soil than Bean tokens than are burned	Resolved
10	Pods may never ripen	Unresolved
11	Bean and the offer backing it are strongly correlated	Unresolved
12	Ability to whitelist assets uncorrelated with Bean price, misaligning governance incentives	Unresolved
13	Unchecked burnFrom return value	Resolved

Detailed Fix Review Results

1. Attackers could mint more Fertilizer than intended due to an unused variable	
Status: **Resolved**	
Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-BEANS-001
Target: `protocol/contracts/farm/facets/FertilizerFacet.sol`	

Description

Due to an unused local variable, an attacker could mint more Fertilizer than should be allowed by the sale.

The `mintFertilizer()` function checks that the `_amount` variable is no greater than the `remaining` variable; this ensures that more Fertilizer than intended cannot be minted; however, the `_amount` variable is not used in subsequent function calls—instead, the `amount` variable is used; the code effectively skips this check, allowing users to mint more Fertilizer than required to recapitalize the protocol.

```
function mintFertilizer(
    uint128 amount,
    uint256 minLP,
    LibTransfer.From mode
) external payable {
    uint256 remaining = LibFertilizer.remainingRecapitalization();
    uint256 _amount = uint256(amount);
    if (_amount > remaining) _amount = remaining;
    LibTransfer.receiveToken(
        C.usdc(),
        uint256(amount).mul(1e6),
        msg.sender,
        mode
    );
    uint128 id = LibFertilizer.addFertilizer(
        uint128(s.season.current),
        amount,
        minLP
    );
    C.fertilizer().beanstalkMint(msg.sender, uint256(id), amount, s.bpf);
}
```

Figure 1.1: The `mintFertilizer()` function in `FertilizerFacet.sol#L35-56`

Note that this flaw can be exploited only once: if users mint more Fertilizer than intended, the `remainingRecapitalization()` function returns `0` because the `dollarPerUnripeLP()` and `unripeLP().totalSupply()` variables are constants.

```
function remainingRecapitalization()
    internal
    view
    returns (uint256 remaining)
{
    AppStorage storage s = LibAppStorage.diamondStorage();
    uint256 totalDollars = C
        .dollarPerUnripeLP()
        .mul(C.unripeLP().totalSupply())
        .div(DECIMALS);
    if (s.recapitalized >= totalDollars) return 0;
    return totalDollars.sub(s.recapitalized);
}
```

Figure 1.2: The `remainingRecapitalization()` function in `LibFertilizer.sol#L132-145`

Fix Analysis

This issue has been resolved. The `_amount` variable has been removed, and the previous assignment to that variable now overwrites `amount` instead. This fixes the implementation issue and also eliminates the risk of having two similarly named variables, decreasing the likelihood that a similar implementation issue will be reintroduced.

2. Lack of a two-step process for ownership transfer	
Status: **Resolved**	
Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-BEANS-002
Target: `protocol/contracts/farm/facets/OwnershipFacet.sol`	

Description

The `transferOwnership()` function is used to change the owner of the Beanstalk protocol. This function calls the `setContractOwner()` function, which immediately sets the contract's new owner. Transferring ownership in one function call is error-prone and could result in irrevocable mistakes.

```
function transferOwnership(address _newOwner) external override {
    LibDiamond.enforceIsContractOwner();
    LibDiamond.setContractOwner(_newOwner);
}
```

Figure 2.1: The `transferOwnership()` function in `OwnershipFacet.sol#L13-16`

Fix Analysis

This issue has been resolved. The `transferOwnership` method now sets an `ownerCandidate` state variable, and a subsequent `claimOwnership` method must be called by the `ownerCandidate` to confirm the ownership transfer. This sufficiently mitigates the risk of making an irrevocable mistake while transferring ownership.

3. Possible underflow could allow more Fertilizer than MAX_RAISE to be minted

Status: **Resolved**	
Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-BEANS-003
Target: `protocol/contracts/fertilizer/FertilizerPremint.sol`	

Description

The `remaining()` function could underflow, which could allow the Barn Raise to continue indefinitely.

Fertilizer is an ERC1155 token issued for participation in the Barn Raise, a community fundraiser intended to recapitalize the Beanstalk protocol with Bean and liquidity provider (LP) tokens that were stolen during the April 2022 governance hack.

Fertilizer entitles holders to a pro rata portion of one-third of minted Bean tokens if the Fertilizer token is active, and it can be minted as long as the recapitalization target ($77 million) has not been reached.

Users who want to buy Fertilizer call the `mint()` function and provide one USDC for each Fertilizer token they want to mint.

```
    function mint(uint256 amount) external payable nonReentrant {
        uint256 r = remaining();
        if (amount > r) amount = r;
        __mint(amount);
        IUSDC.transferFrom(msg.sender, CUSTODIAN, amount);
    }
```

Figure 3.1: The `mint()` function in `FertilizerPremint.sol#L51-56`

The `mint()` function first checks how many Fertilizer tokens remain to be minted by calling the `remaining()` function (figure 3.2); if the user is trying to mint more Fertilizer than available, the `mint()` function mints all of the Fertilizer tokens that remain.

```
    function remaining() public view returns (uint256) {
        return MAX_RAISE - IUSDC.balanceOf(CUSTODIAN);
    }
```

Figure 3.2: The `remaining()` function in `FertilizerPremint.sol#L84-87`

However, the `FertilizerPremint` contract does not use Solidity 0.8, so it does not have native overflow and underflow protection. As a result, if the amount of Fertilizer purchased reaches `MAX_RAISE` (i.e., 77 million), an attacker could simply send one USDC to the `CUSTODIAN` wallet to cause the `remaining()` function to underflow, allowing the sale to continue indefinitely.

In this particular case, Beanstalk protocol funds are not at risk because all the USDC used to purchase Fertilizer tokens is sent to a Beanstalk community-owned multisignature wallet; however, users who buy Fertilizer after such an exploit would lose the gas funds they spent, and the project would incur further reputational damage.

Fix Analysis

This issue has been resolved. An additional check has been added to the `remaining()` function that will return zero instead of underflowing if the custodian's balance is above the value of `MAX_RAISED`.

4. Risk of Fertilizer id collision that could result in loss of funds	
Status: **Resolved**	
Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-BEANS-004
Target: `protocol/contracts/fertilizer/Fertilizer.sol`	

Description

If a user mints Fertilizer tokens twice during two different seasons, the same token `id` for both tokens could be calculated, and the first entry will be overridden; if this occurs and the `bpf` value changes, the user would be entitled to less yield than expected.

To mint new Fertilizer tokens, users call the `mintFertilizer()` function in the `FertilizerFacet` contract. An `id` is calculated for each new Fertilizer token that is minted; not only is this `id` an identifier for the token, but it also represents the `endBpf` period, which is the moment at which the Fertilizer reaches "maturity" and can be redeemed without incurring any penalty.

```
function mintFertilizer(
    uint128 amount,
    uint256 minLP,
    LibTransfer.From mode
) external payable {
    uint256 remaining = LibFertilizer.remainingRecapitalization();
    uint256 _amount = uint256(amount);
    if (_amount > remaining) _amount = remaining;
    LibTransfer.receiveToken(
        C.usdc(),
        uint256(amount).mul(1e6),
        msg.sender,
        mode
    );
    uint128 id = LibFertilizer.addFertilizer(
        uint128(s.season.current),
        amount,
        minLP
    );
    C.fertilizer().beanstalkMint(msg.sender, uint256(id), amount, s.bpf);
}
```

Figure 4.1: The `mintFertilizer()` function in `Fertilizer.sol#L35-55`

The id is calculated by the `addFertilizer()` function in the `LibFertilizer` library as the sum of 1 and the `bpf` and `humidity` values.

```
function addFertilizer(
    uint128 season,
    uint128 amount,
    uint256 minLP
) internal returns (uint128 id) {
    AppStorage storage s = LibAppStorage.diamondStorage();
    uint256 _amount = uint256(amount);
    // Calculate Beans Per Fertilizer and add to total owed
    uint128 bpf = getBpf(season);
    s.unfertilizedIndex = s.unfertilizedIndex.add(
        _amount.mul(uint128(bpf))
    );
    // Get id
    id = s.bpf.add(bpf);
    [...]
}

function getBpf(uint128 id) internal pure returns (uint128 bpf) {
    bpf = getHumidity(id).add(1000).mul(PADDING);
}

function getHumidity(uint128 id) internal pure returns (uint128 humidity) {
    if (id == REPLANT_SEASON) return 5000;
    if (id >= END_DECREASE_SEASON) return 200;
    uint128 humidityDecrease = id.sub(REPLANT_SEASON + 1).mul(5);
    humidity = RESTART_HUMIDITY.sub(humidityDecrease);
}
```

Figure 4.2: The `id` calculation in `LibFertilizer.sol#L32-67`

However, the method that generates these token `id`s does not prevent collisions. The `bpf` value is always increasing (or does not move), and `humidity` decreases every season until it reaches 20%. This makes it possible for a user to mint two tokens in two different seasons with different `bpf` and `humidity` values and still get the same token `id`.

```
    function beanstalkMint(address account, uint256 id, uint128 amount, uint128 bpf)
external onlyOwner {
        _balances[id][account].lastBpf = bpf;
        _safeMint(
            account,
            id,
            amount,
            bytes('0')
        );
    }
```

Figure 4.3: The `beanstalkMint()` function in `Fertilizer.sol#L40-48`

An `id` collision is not necessarily a problem; however, when a token is minted, the value of the `lastBpf` field is set to the `bpf` of the current season, as shown in figure 4.3. This field is very important because it is used to determine the penalty, if any, that a user will incur when redeeming Fertilizer.

To redeem Fertilizer, users call the `claimFertilizer()` function, which in turn calls the `beanstalkUpdate()` function on the `Fertilizer` contract.

```
function claimFertilized(uint256[] calldata ids, LibTransfer.To mode)
    external
    payable
{
    uint256 amount = C.fertilizer().beanstalkUpdate(msg.sender, ids, s.bpf);
    LibTransfer.sendToken(C.bean(), amount, msg.sender, mode);
}
```

Figure 4.4: The `claimFertilizer()` function in `FertilizerFacet.sol#L27-33`

```
function beanstalkUpdate(
    address account,
    uint256[] memory ids,
    uint128 bpf
) external onlyOwner returns (uint256) {
    return __update(account, ids, uint256(bpf));
}

function __update(
    address account,
    uint256[] memory ids,
    uint256 bpf
) internal returns (uint256 beans) {
    for (uint256 i = 0; i < ids.length; i++) {
        uint256 stopBpf = bpf < ids[i] ? bpf : ids[i];
        uint256 deltaBpf = stopBpf - _balances[ids[i]][account].lastBpf;
        if (deltaBpf > 0) {
            beans = beans.add(deltaBpf.mul(_balances[ids[i]][account].amount));
            _balances[ids[i]][account].lastBpf = uint128(stopBpf);
        }
    }
    emit ClaimFertilizer(ids, beans);
}
```

Figure 4.5: The update flow in `Fertilizer.sol#L32-38` and `L72-86`

The `beanstalkUpdate()` function then calls the `__update()` function. This function first calculates the `stopBpf` value, which is one of two possible values. If the Fertilizer is being redeemed early, `stopBpf` is the `bpf` at which the Fertilizer is being redeemed; if the token is being redeemed at "maturity" or later, `stopBpf` is the token `id` (i.e., the `endBpf` value). Afterward, `__update()` calculates the `deltaBpf` value, which is used to determine the penalty, if any, that the user will incur when redeeming the token; `deltaBpf` is calculated using the `stopBpf` value that was already defined and the `lastBpf` value, which is the `bpf` corresponding to the last time the token was redeemed or, if it was never redeemed, the `bpf` at the moment the token was minted. Finally, the token's `lastBpf` field is updated to the `stopBpf`.

Because of the `id` collision, users could accidentally mint Fertilizer tokens with the same `id` in two different seasons and override their first mint's `lastBpf` field, ultimately reducing the amount of yield they are entitled to.

Fix Analysis

This issue has been resolved. Collisions of Fertilizer `id`s are still possible; however, the Beanstalk team added an additional call to `__update` to claim unclaimed Bean tokens and to update values such that Fertilizer tokens with the same `id` also have the same `lastBpf`. This prevents funds from being lost, a risk described in this finding.

5. The sunrise() function rewards callers only with the base incentive	
Status: **Resolved**	
Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-BEANS-005
Target: `protocol/contracts/farm/facets/SeasonFacet/SeasonFacet.sol`	

Description

The increasing incentive that encourages users to call the `sunrise()` function in a timely manner is not actually applied.

According to the Beanstalk white paper, the reward paid to users who call the `sunrise()` function should increase by 1% every second (for up to 300 seconds) after this method is eligible to be called; this incentive is designed so that, even when gas prices are high, the system can move on to the next season in a timely manner.

This increasing incentive is calculated and included in the emitted logs, but it is not actually applied to the number of Bean tokens rewarded to users who call `sunrise()`.

```
function incentivize(address account, uint256 amount) private {
    uint256 timestamp = block.timestamp.sub(
        s.season.start.add(s.season.period.mul(season()))
    );
    if (timestamp > 300) timestamp = 300;
    uint256 incentive = LibIncentive.fracExp(amount, 100, timestamp, 1);
    C.bean().mint(account, amount);
    emit Incentivization(account, incentive);
}
```

Figure 5.1: The incentive calculation in `SeasonFacet.sol#70-78`

Fix Analysis

This issue has been resolved. The `incentive` value, instead of the provided `amount`, is now used to mint the rewarded Bean tokens. This properly applies the increasing incentive described by the white paper.

6. Solidity compiler optimizations can be problematic	
Status: **Unresolved**	
Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-BEANS-006
Target: The Beanstalk protocol	

Description

Beanstalk has enabled optional compiler optimizations in Solidity.

There have been several optimization bugs with security implications. Moreover, optimizations are actively being developed. Solidity compiler optimizations are disabled by default, and it is unclear how many contracts in the wild actually use them. Therefore, it is unclear how well they are being tested and exercised.

High-severity security issues due to optimization bugs have occurred in the past. A high-severity bug in the `emscripten`-generated `solc-js` compiler used by Truffle and Remix persisted until late 2018. The fix for this bug was not reported in the Solidity CHANGELOG. Another high-severity optimization bug resulting in incorrect bit shift results was patched in Solidity 0.5.6. More recently, another bug due to the incorrect caching of `keccak256` was reported.

A compiler audit of Solidity from November 2018 concluded that the optional optimizations may not be safe.

It is likely that there are latent bugs related to optimization and that new bugs will be introduced due to future optimizations.

Fix Analysis

This issue has not been resolved. The Beanstalk team understands the risks of using compiler optimizations and has chosen to accept them without making any changes to the contract compilation process.

7. Lack of support for external transfers of nonstandard ERC20 tokens	
Status: **Resolved**	
Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-BEANS-007
Target: `protocol/contracts/farm/facets/TokenFacet.sol`	

Description

For external transfers of nonstandard ERC20 tokens via the `TokenFacet` contract, the code uses the standard `transferFrom` operation from the given token contract without checking the operation's `returndata`; as a result, successfully executed transactions that fail to transfer tokens will go unnoticed, causing confusion in users who believe their funds were successfully transferred.

The `TokenFacet` contract exposes `transferToken()`, an external function that users can call to transfer ERC20 tokens both to and from the contract and between users.

```
function transferToken(
    IERC20 token,
    address recipient,
    uint256 amount,
    LibTransfer.From fromMode,
    LibTransfer.To toMode
) external payable {
    LibTransfer.transferToken(token, recipient, amount, fromMode, toMode);
}
```

Figure 7.1: The `transferToken()` function in `TokenFacet.sol#L39-47`

This function calls the `LibTransfer` library, which handles the token transfer.

```
function transferToken(
    IERC20 token,
    address recipient,
    uint256 amount,
    From fromMode,
    To toMode
) internal returns (uint256 transferredAmount) {
    if (fromMode == From.EXTERNAL && toMode == To.EXTERNAL) {
        token.transferFrom(msg.sender, recipient, amount);
        return amount;
    }
```

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

Figure 7.2: The `transferToken()` function in `LibTransfer.sol#L29-43`

The `LibTransfer` library uses the `fromMode` and `toMode` values to determine a transfer's sender and receiver, respectively; in most cases, it uses the `safeERC20` library to execute transfers.

However, if `fromMode` and `toMode` are both marked as `EXTERNAL`, then the `transferFrom` function of the token contract will be called directly, and `safeERC20` will not be used. Essentially, if a user tries to transfer a nonstandard ERC20 token that does not revert on failure and instead indicates a transaction's success or failure in its return data, the user could be led to believe that failed token transfers were successful.

Fix Analysis

This issue has been resolved. Transfers of nonstandard ERC20 tokens are now performed using `safeTransferFrom`, and afterward, an additional underflow-resistant balance check is performed. This ensures that such transactions will revert on invalid transfers, including those attempting to transfer nonstandard tokens that return `false` instead of reverting on invalid transfers.

8. Plot transfers from users with allowances revert if the owner has an existing pod listing

Status: **Resolved**	
Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-BEANS-008
Target: `protocol/contracts/farm/facets/MarketplaceFacet.sol`	

Description

Whenever a plot transfer is executed by a user with an allowance (i.e., a transfer in which the caller was approved by the plot's owner), the transfer will revert if there is an existing listing for the pods contained in that plot.

The `MarketplaceFacet` contract exposes a function, `transferPlot()`, that allows the owner of a plot to transfer the pods in that plot to another user; additionally, the owner of a plot can call the `approvePods()` function (figure 8.1) to approve other users to transfer these pods on the owner's behalf.

```
function approvePods(address spender, uint256 amount)
    external
    payable
    nonReentrant
{
    require(spender != address(0), "Field: Pod Approve to 0 address.");
    setAllowancePods(msg.sender, spender, amount);
    emit PodApproval(msg.sender, spender, amount);
}
```

Figure 8.1: The `approvePods()` function in `MarketplaceFacet.sol#L147-155`

Once approved, the given address can call the `transferPlot()` function to transfer pods on the owner's behalf. The function checks and decreases the allowance and then checks whether there is an existing pod listing for the target pods. If there is an existing listing, the function tries to cancel it by calling the `_cancelPodListing()` function.

```
function transferPlot(
    address sender,
    address recipient,
    uint256 id,
    uint256 start,
```

```
        uint256 end
    ) external payable nonReentrant {
        require(
            sender != address(0) && recipient != address(0),
            "Field: Transfer to/from 0 address."
        );
        uint256 amount = s.a[sender].field.plots[id];
        require(amount > 0, "Field: Plot not owned by user.");
        require(end > start && amount >= end, "Field: Pod range invalid.");
        amount = end - start; // Note: SafeMath is redundant here.
        if (
            msg.sender != sender &&
            allowancePods(sender, msg.sender) != uint256(-1)
        ) {
            decrementAllowancePods(sender, msg.sender, amount);
        }

        if (s.podListings[id] != bytes32(0)) {
            _cancelPodListing(id); // TODO: Look into this cancelling.
        }
        _transferPlot(sender, recipient, id, start, amount);
    }
```

Figure 8.2: The `transferPlot()` function in `MarketplaceFacet.sol#L119-145`

The `_cancelPodListing()` function receives only an `id` as the input and relies on the `msg.sender` to determine the listing's owner. However, if the transfer is executed by a user with an allowance, the `msg.sender` is the user who was granted the allowance, not the owner of the listing. As a result, the function will revert.

```
    function _cancelPodListing(uint256 index) internal {
        require(
            s.a[msg.sender].field.plots[index] > 0,
            "Marketplace: Listing not owned by sender."
        );
        delete s.podListings[index];
        emit PodListingCancelled(msg.sender, index);
    }
```

Figure 8.3: The `_cancelPodListing()` function in `Listing.sol#L149-156`

Fix Analysis

This issue has been resolved. The `_cancelPodListing` function now accepts an `owner` parameter instead of using `msg.sender`.

9. Users can sow more Bean tokens than are burned	
Status: **Resolved**	
Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-BEANS-009
Target: `protocol/contracts/farm/facets/FieldFacet.sol`	

Description

An accounting error allows users to sow more Bean tokens than the available soil allows.

Whenever the price of Bean is below its peg, the protocol issues soil. Soil represents the willingness of the protocol to take Bean tokens off the market in exchange for a pod. Essentially, Bean owners loan their tokens to the protocol and receive pods in exchange. We can think of pods as non-callable bonds that mature on a first-in-first-out (FIFO) basis as the protocol issues new Bean tokens.

Whenever soil is available, users can call the `sow()` and `sowWithMin()` functions in the `FieldFacet` contract.

```
function sowWithMin(
    uint256 amount,
    uint256 minAmount,
    LibTransfer.From mode
) public payable returns (uint256) {
    uint256 sowAmount = s.f.soil;
    require(
        sowAmount >= minAmount && amount >= minAmount && minAmount > 0,
        "Field: Sowing below min or 0 pods."
    );
    if (amount < sowAmount) sowAmount = amount;
    return _sow(sowAmount, mode);
}
```

Figure 9.1: The `sowWithMin()` function in `FieldFacet.sol#L41-53`

The `sowWithMin()` function ensures that there is enough soil to sow the given number of Bean tokens and that the call will not sow fewer tokens than the specified `minAmount`. Once it makes these checks, it calls the `_sow()` function.

```
function _sow(uint256 amount, LibTransfer.From mode)
    internal
```

```
        returns (uint256 pods)
    {
        pods = LibDibbler.sow(amount, msg.sender);
        if (mode == LibTransfer.From.EXTERNAL)
            C.bean().burnFrom(msg.sender, amount);
        else {
            amount = LibTransfer.receiveToken(C.bean(), amount, msg.sender, mode);
            C.bean().burn(amount);
        }
    }
```

Figure 9.2: The `_sow()` function in `FieldFacet.sol#L55-65`

The `_sow()` function first calculates the number of pods that will be sown by calling the `sow()` function in the `LibDibbler` library, which performs the internal accounting and calculates the number of pods that the user is entitled to.

```
    function sow(uint256 amount, address account) internal returns (uint256) {
        AppStorage storage s = LibAppStorage.diamondStorage();
        // We can assume amount <= soil from getSowAmount
        s.f.soil = s.f.soil - amount;
        return sowNoSoil(amount, account);
    }

    function sowNoSoil(uint256 amount, address account)
        internal
        returns (uint256)
    {
        AppStorage storage s = LibAppStorage.diamondStorage();
        uint256 pods = beansToPods(amount, s.w.yield);
        sowPlot(account, amount, pods);
        s.f.pods = s.f.pods.add(pods);
        saveSowTime();
        return pods;
    }

    function sowPlot(
        address account,
        uint256 beans,
        uint256 pods
    ) private {
        AppStorage storage s = LibAppStorage.diamondStorage();
        s.a[account].field.plots[s.f.pods] = pods;
        emit Sow(account, s.f.pods, beans, pods);
    }
```

Figure 9.3: The `sow()`, `sowNoSoil()`, and `sowPlot()` functions in `LibDibbler.sol#L41-53`

Finally, the `sowWithMin()` function burns the Bean tokens from the caller's account, removing them from the supply. To do so, the function calls `burnFrom()` if the `mode` parameter is EXTERNAL (i.e., if the Bean tokens to be burned are not escrowed in the contract) and burn() if the Bean tokens are escrowed.

If the mode parameter is not EXTERNAL, the receiveToken() function is executed to update the internal accounting of the contract before burning the tokens. This function returns the number of tokens that were "transferred" into the contract.

In essence, the receiveToken() function allows the contract to correctly account for token transfers into it and to manage internal balances without performing token transfers.

```
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;
    }
    token.safeTransferFrom(sender, address(this), amount - receivedAmount);
    return amount;
}
```

Figure 9.4: The receiveToken() function in FieldFacet.sol#L41-53

However, if the mode parameter is INTERNAL_TOLERANT, the contract allows the user to partially fill amount (i.e., to transfer as much as the user can), which means that if the user does not own the given amount of Bean tokens, the protocol simply burns as many tokens as the user owns but still allows the user to sow the full amount.

Fix Analysis

This issue has been resolved. The number of Bean tokens sown now depends on the amount transferred rather than the provided input amount.

10. Pods may never ripen	
Status: **Unresolved**	
Severity: **Undetermined**	Difficulty: **Undetermined**
Type: Economic	Finding ID: TOB-BEANS-010
Target: The Beanstalk protocol	

Description

Whenever the price of Bean is below its peg, the protocol takes Bean tokens off the market in exchange for a number of pods dependent on the current interest rate. Essentially, Bean owners loan their tokens to the protocol and receive pods in exchange. We can think of pods as loans that are repaid on a FIFO basis as the protocol issues new Bean tokens. A group of pods that are created together is called a plot.

The queue of plots is referred to as the pod line. The pod line has no practical bound on its length, so during periods of decreasing demand, it can grow indefinitely. No yield is awarded until the given plot owner is first in line and until the price of Bean is above its value peg.

While the protocol does not default on its debt, the only way for pods to ripen is if demand increases enough for the price of Bean to be above its value peg for some time. While the price of Bean is above its peg, a portion of newly minted Bean tokens is used to repay the first plot in the pod line until fully repaid, decreasing the length of the pod line.

During an extended period of decreasing supply, the pod line could grow long enough that lenders receive an unappealing time-weighted rate of return, even if the yield is increased; a sufficiently long pod line could encourage users—uncertain of whether future demand will grow enough for them to be repaid—to sell their Bean tokens rather than lending them to the protocol. Under such circumstances, the protocol will be unable to disincentivize Bean market sales, disrupting its ability to return Bean to its value peg.

Fix Analysis

This issue has not been resolved. The Beanstalk team provided the following rationale for its acceptance of the associated risk:

> Pods are zero coupon bonds without a fixed maturity. The fact that they may never Ripen is true by definition, so this feels a bit tautological. If you are going to leave

this issue, you should add that Unfertilized Beans may never become Fertilized (redeemable) either, as they are also zero coupon bonds without a fixed maturity.

Although unfertilized Bean tokens are not guaranteed to mature either, Fertilizer is not directly critical for maintaining Bean's value peg; therefore, Fertilizer's lack of a maturity date does not pose a significant risk.

11. Bean and the offer backing it are strongly correlated	
Status: **Unresolved**	
Severity: **Undetermined**	Difficulty: **Undetermined**
Type: Economic	Finding ID: TOB-BEANS-011
Target: The Beanstalk protocol	

Description

In response to prolonged periods of decreasing demand for Bean tokens, the Beanstalk protocol offers to borrow from users who own Bean tokens, decreasing the available Bean supply and returning the Bean price to its peg. To incentivize users to lend their Bean tokens to the protocol rather than immediately selling them in the market, which would put further downward pressure on the price of Bean, the protocol offers users a reward of more Bean tokens in the future.

The demand for holding Bean tokens at present and the demand for receiving Bean tokens in the future are strongly correlated, introducing reflexive risk. If the demand for Bean decreases, we can expect a proportional increase in the marginal Bean supply and a decrease in demand to receive Bean in the future, weakening the system's ability to restore Bean to its value peg.

The FIFO queue of lenders is designed to combat reflexivity by encouraging rational actors to quickly support a dip in Bean price rather than selling. However, this mechanism assumes that the demand for Bean will increase in the future; investors may not share this assumption if present demand for Bean is low. Reflexivity is present whenever a stablecoin and the offer backing it are strongly correlated, even if the backing offer is time sensitive.

Fix Analysis

This issue has not been resolved. The Beanstalk team provided the following rationale for its acceptance of the associated risk:

> The primary source of Bean price stability is the credit of the protocol (*i.e.*, Beanstalk's ability to borrow from the market). As demand for Beans decreases, causing a short term decrease in the price of a Bean, the benefit for lending to the protocol is inversely correlated with the price. This is a function of the FIFO Pod harvest schedule. As the price decreases to X, the yield for lending to Beanstalk increases by 1/X.

However, we recommend describing the yield as Bean-denominated benefits because profits from an increasing price are shared with all Bean holders, not just lenders. The expectation of positive price movements does not resolve the underlying issue.

12. Ability to whitelist assets uncorrelated with Bean price, misaligning governance incentives	
Status: **Unresolved**	
Severity: **Undetermined**	Difficulty: **Undetermined**
Type: Economic	Finding ID: TOB-BEANS-012
Target: The Beanstalk protocol	

Description

Stalk is the governance token of the system, rewarded to users who deposit certain whitelisted assets into the silo, the system's asset storage.

When demand for Bean increases, the protocol increases the Bean supply by minting new Bean tokens and allocating some of them to Stalk holders. Additionally, if the price of Bean remains above its peg for an extended period of time, then a season of plenty (SoP) occurs: Bean is minted and sold on the open market in exchange for exogenous assets such as ETH. These exogenous assets are allocated entirely to Stalk holders.

When demand for Bean decreases, the protocol decreases the Bean supply by borrowing Bean tokens from Bean owners. If the demand for Bean is persistently low and some of these loans are never repaid, Stalk holders are not directly penalized by the protocol. However, if the only whitelisted assets are strongly correlated with the price of Bean (such as ETH:BEAN LP tokens), then the value of Stalk holders' deposited collateral would decline, indirectly penalizing Stalk holders for an unhealthy system.

If, however, exogenous assets without a strong correlation to Bean are whitelisted, then Stalk holders who have deposited such assets will be protected from financial penalties if the price of Bean crashes.

Fix Analysis

This issue has not been resolved. The Beanstalk team provided the following rationale for its acceptance of the associated risk:

> The question is more about whether or not governance incentives are misaligned such that assets that do not have exposure to the Bean price would be whitelisted.
>
> We would argue given that all Stalk holders have exposure to Beans, and that the incentive for holding Beans and Stalk is Bean seigniorage, and that Bean seigniorage is a function of demand for Beans, it would not make any sense for Stalk holders to

> vote to distribute Bean seigniorage to non-Bean holders. If this is a risk, perhaps other risks like “minting infinite Beans misaligns governance incentives” should also be listed. Again, the question is whether the incentives of Stalk holders are such that the suggested behavior is economically beneficial them. In both instances, it is not.

The governance attack described in this issue is subtle, especially given that other stablecoins accept exogenous deposits for legitimate reasons. If users are sufficiently aware of this attack vector, then we agree with the Beanstalk team that it presents a risk comparable to that of simpler governance attacks, which are economically beneficial to the subset of attacking voters.

13. Unchecked burnFrom return value	
Status: **Resolved**	
Severity: **Informational**	Difficulty: **Undetermined**
Type: Undefined Behavior	Finding ID: TOB-BEANS-013
Target: `protocol/contracts/farm/facets/UnripeFacet.sol`	

Description

While recapitalizing the Beanstalk protocol, Bean and LP tokens that existed before the 2022 governance hack are represented as unripe tokens. Ripening is the process of burning unripe tokens in exchange for a pro rata share of the underlying assets generated during the Barn Raise. Holders of unripe tokens call the `ripen` function to receive their portion of the recovered underlying assets. This portion grows while the price of Bean is above its peg, incentivizing users to ripen their tokens later, when more of the loss has been recovered.

The `ripen` code assumes that if users try to redeem more unripe tokens than they hold, `burnFrom` will revert. If `burnFrom` returns `false` instead of reverting, the failure of the balance check will go undetected, and the caller will be able to recover all of the underlying tokens held by the contract. While `LibUnripe.decrementUnderlying` will revert on calls to ripen more than the contract's balance, it does not check the user's balance.

The source code of the `unripeToken` contract was not provided for review during this audit, so we could not determine whether its `burnFrom` method is implemented safely.

```
function ripen(
    address unripeToken,
    uint256 amount,
    LibTransfer.To mode
) external payable nonReentrant returns (uint256 underlyingAmount) {
    underlyingAmount = getPenalizedUnderlying(unripeToken, amount);

    LibUnripe.decrementUnderlying(unripeToken, underlyingAmount);

    IBean(unripeToken).burnFrom(msg.sender, amount);

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

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

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

```
}
```

Figure 13.1: The `ripen()` function in `UnripeFacet.sol#L51-67`

Fix Analysis

This issue is resolved; no fixes were necessary. The Beanstalk team did not add any extra assertions, so the `unripeToken` contract's `burnFrom` method must revert if the user has insufficient balance to burn the given `amount` of unripe tokens. The Beanstalk team has confirmed that the contract will inherit its `burnFrom` method from a standard OpenZeppelin `ERC20Burnable` library, which reverts safely and will prevent the issue from being exploited. Additionally, the team added comments to warn future maintainers that the `ripen` method depends on `burnFrom` to revert on underflows, decreasing the likelihood that a severe mistake will be made during future changes to the code.

A. Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	Description
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.

B. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	Description
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.