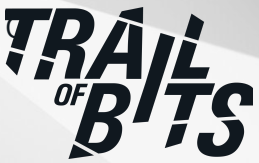

Beanstalk

Security Assessment

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 mutually 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.

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 timeline, test targets, and coverage are provided in subsequent sections of this report.

Project Scope

Our testing efforts were focused on the identification of flaws that could result in the compromise of a smart contract, a loss of funds, or unexpected behavior in the target system. We conducted this audit with access to the Beanstalk source code. We used Slither to automatically detect common problems and performed a manual review of the source code.

Summary of Findings

The audit uncovered significant flaws that could result in unexpected behavior. A summary of the findings and details on notable findings are 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

Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

- **Lack of data validation throughout the codebase (TOB-BEANS-004, TOB-BEANS-009)**
  During the review, we identified multiple issues that stem from a lack of data validation. We discovered multiple high-severity/low-difficulty issues that would cause users to receive fewer funds than they should. The accounting error described in TOB-BEANS-009 allows users to sow Bean tokens without burning the same number of tokens, rendering the mechanism for driving the price of Bean back to its peg useless.

- **Failure to adhere to smart contract best practices (TOB-BEANS-001)**
  A medium-severity overflow issue in the `FertilizerPremint` contract (which was live during the review) could allow the Barn Raise to continue indefinitely. This issue was patched by the Beanstalk team, and the contract was upgraded.

- **Economic and governance risks (TOB-BEANS-010, TOB-BEANS-011, TOB-BEANS-012)**
  We identified scenarios in which the incentivization mechanisms could fail to provide adequate incentives, preventing the system from reliably maintaining the Bean token's value peg.

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

Project Goals

The engagement was scoped to provide a security assessment of the Beanstalk protocol. Specifically, we sought to answer the following non-exhaustive list of questions:

- Could the fundraiser raise more than its target of $77 million in Fertilizer sales?
- Do users receive the correct yield from purchasing Fertilizer tokens?
- Are the system's incentives properly applied?
- Does the debt issuance, sowing, and harvesting mechanism behave correctly?
- Could an attacker steal funds from the system?
- Are there appropriate access controls in place for user and admin operations?
- Could an attacker trap the system?
- Could users lose access to their funds?

Project Targets

The engagement involved a review and testing of the following target.

Beanstalk

Repository	https://github.com/BeanstalkFarms/Beanstalk-Replanted
Version	`f501c25eb41e391c35a2926dacca7d9912e700f3`
Type	Solidity
Platform	EVM

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches and their results include the following:

- **The oracle.** The system relies on time-weighted average values from Curve pools to determine the deviation between Bean and its value peg. We checked that the values supplied by the oracle are in sync with Curve's liquidity, and we looked for ways to manipulate these values, including through the use of flash loans.

- **Fertilizer.** Fertilizer is an ERC1155 token issued to investors during the Barn Raise, a fundraiser intended to recapitalize Bean and liquidity provider (LP) tokens that were lost during the April 2022 governance exploit. Fertilizer entitles holders to a pro rata portion of one third of minted Bean tokens while the Fertilizer token is active, and it can be minted as long as the recapitalization target ($77 million) has not been reached.

  In reviewing the Fertilizer-related contracts and libraries, we focused on a deep manual review of the minting and redeeming mechanisms to ensure that Fertilizer could not be minted beyond the expected $77 million target. We also ensured that users receive the yield they are entitled to and looked for common Solidity vulnerabilities (e.g., overflows and underflows).

- **The marketplace.** The marketplace allows users to create, fill, and cancel pod listings and pod orders.

  In addition to checking for common Solidity vulnerabilities (e.g., overflows and underflows), we manually reviewed the logic of the `MarketplaceFacet` contract and the marketplace dependencies. We focused on ensuring that the internal accounting is performed correctly when listings and orders are created, fulfilled, and transferred.

- **The field.** The field is Beanstalk's native credit facility. Whenever there is soil in Beanstalk, users can sow Bean tokens into pods (i.e., burn them). A group of pods that are created together is called a plot, which is placed at the end of the pod line. Pods mature on a first-in-first-out (FIFO) basis whenever the protocol mints new Bean tokens.

  In reviewing the `FieldFacet` and related contracts, we focused on manually reviewing the sowing mechanism (e.g., to ensure that the correct number of Bean tokens is burned and that the soil supply is reduced accordingly) and the redeeming mechanism (e.g., to ensure that a given pod cannot be redeemed multiple times). We also looked for common Solidity vulnerabilities (e.g., overflows and underflows).

- **The unripe token, pausing, whitelist, ownership, and Bean-denominated-value (BDV) facets.** During our review of these modules (facets) and their dependencies, we performed a manual review of the mechanisms they implement in order to understand their purpose and to ensure that they work as intended. We also looked for common Solidity vulnerabilities and ensured that best practices, such as the use of a two-step ownership transfer mechanism, are followed.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that warrant further review:

- **The convert, Curve, and silo components.** Due to the time-boxed nature of this work, we deprioritized the review of these components and their dependencies. We did not closely review the logic of these modules or their interactions with other components. Instead, we focused on looking for common Solidity vulnerabilities within them.
- **Out of scope components.** The source code for the following components was either explicitly out of scope or was not provided as part of the target project:
    - **`unripeToken`.** The source code for the `unripeToken` contract was not provided. If its `burnFrom` method returns `false` instead of reverting, issue TOB-BEANS-013 will become a high-severity/low-difficulty issue.
    - **The voting and upgrading mechanisms.** The code responsible for collecting votes, tallying them, and executing decisions was not included in the target project and was, therefore, not reviewed.
    - **The Merkle tree.** The accounts and balances affected by the April 2022 hack are saved in a Merkle tree and are verified when users claim unripe tokens. The generation of these Merkle roots and proofs were not included in the target project and were, therefore, not reviewed.

Two elements of the system limited our overall coverage:

- **Language.** Farm-themed language is used to describe the system, which can be a valuable community-building tool. However, describing smart contracts using terminology that is unrelated to their logic greatly increases the risk that security problems could go unnoticed. We spent a significant amount of time compiling a glossary of terms to help us understand the farm-themed language used throughout the system (appendix C).
- **Complex architecture**. The system is divided into facets that do not have clear responsibilities and often modify parts of the state that other components of the

system are responsible for. This significantly increases the complexity of the system and requires that reviewers thoroughly understand multiple modules to determine the correctness of a single feature.

For example, the `MarketplaceFacet` contract is responsible for the listing, fulfillment, and cancellation of pod orders and pod listings; however, the `FieldFacet` contract, which is responsible for the sowing and harvesting of pods, directly accesses listings and cancels them whenever a plot is harvested.

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	While performing arithmetic operations, the protocol sometimes uses the `SafeMath` library, but it more often uses the unchecked native math operators of Solidity 0.7. Several complex arithmetic operations are undocumented and untested; as a result, it is difficult to understand what the bounds of the variables are (e.g., can the "weather" be below one?), whether these bounds hold, and what actors can influence them. As the system relies heavily on its underlying arithmetic, it is a key component that requires more thorough documentation and testing to ease its review. We identified an issue in which an underflow could cause the Fertilizer sale to fail to end when intended (TOB-BEANS-001).	**Weak**
Auditing	The Beanstalk protocol's critical state-changing operations emit sufficient events. The protocol has an incident response plan, but it lacks sufficient detail; see appendix E for our recommendations on documenting and maintaining an incident response plan.	**Moderate**
Authentication / Access Controls	The level of access provided to users is limited, and the protocol's owner role is the protocol governance system. However, the current governance system is managed by a multisignature wallet and was out of scope for this review.	**Further Investigation Required**
Complexity Management	The protocol is divided into multiple facets that the main contract (`Diamond`) `delegatecall`s into. This allows the contracts to be upgradeable and the system to bypass the	**Weak**

	contract size limit. Additionally, most shared functionality is separated into libraries, providing the facets easy access to the functionality and reducing code duplication. However, in many instances, multiple facets access and modify the same state, which makes it difficult to isolate state changes. It also requires that multiple contracts be reviewed and thoroughly understood to evaluate the correctness of a single module. Additionally, in some instances, such as TOB-BEAN-004, a single variable represents multiple properties, adding extra complexity to the code. Finally, the farm-themed language applied to internal variables and functions obscures the developers' intentions and significantly hampered our code review. The protocol would benefit greatly from a review of its design with a focus on defining clear responsibilities for each of the facets and on ensuring that responsibilities are not shared between them. Additionally, creating and maintaining a component diagram of the system would help drive and document the protocol's design and highlight opportunities for improvement.	
Decentralization	The protocol is mostly decentralized and has a governance system; however, our current understanding is that this governance system (previously a DAO) was replaced by a multisignature wallet after the governance exploit. This aspect of the system was out of scope for this review and, therefore, could not be evaluated.	**Further Investigation Required**
Documentation	The system extensively uses farm-themed language (i.e., soil, sow, pods, harvest, etc.) to explain how the system operates and as a naming convention; this adds unnecessary complexity to the code and makes it significantly harder to review critical business logic. Furthermore, the codebase generally lacks inline documentation, and the glossary of terms in the white paper is incomplete and often uses farm-themed terms to define other farm-themed terms; such an approach to	**Weak**

	defining terms results in unclear definitions, which could be confusing to those who are new to the project. Specifications that describe how each function is expected to behave are missing. See appendix C for a non-exhaustive glossary of terms that we produced during our review of the code and of the white paper and through our communications with the Beanstalk team. This appendix also provides recommendations on maintaining the glossary and integrating it into the existing documentation.	
Front-Running Resistance	The risk of front-running is inherent in the protocol's design, through components like the marketplace, which is an on-chain orderbook.	**Moderate**
Low-Level Manipulation	Assembly and low-level calls are rarely used in the protocol; where they are used, we identified no issues.	**Satisfactory**
Testing and Verification	The system's security properties and assumptions are not indicated in the tests. The insufficient test suite reflects the lack of a specification and should be expanded. Certain issues found during this audit (e.g., TOB-BEANS-001, TOB-BEANS-005) could have been discovered by adequate unit testing. The presence of such bugs indicates that the system would benefit from more thorough testing and automated testing through fuzzing and symbolic execution.	**Weak**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Attackers could mint more Fertilizer than intended due to an unused variable	Data Validation	Medium
2	Lack of a two-step process for ownership transfer	Data Validation	High
3	Possible underflow could allow more Fertilizer than MAX_RAISE to be minted	Data Validation	Medium
4	Risk of Fertilizer id collision that could result in loss of funds	Data Validation	High
5	The sunrise() function rewards callers only with the base incentive	Data Validation	Medium
6	Solidity compiler optimizations can be problematic	Undefined Behavior	Informational
7	Lack of support for external transfers of nonstandard ERC20 tokens	Data Validation	Informational
8	Plot transfers from users with allowances revert if the owner has an existing pod listing	Data Validation	Low
9	Users can sow more soil than Bean tokens than are burned	Data Validation	High
10	Pods may never ripen	Economic	Undetermined
11	Bean and the offer backing it are strongly correlated	Economic	Undetermined

12	Ability to whitelist assets uncorrelated with Bean price, misaligning governance incentives	Economic	**Undetermined**
13	Unchecked burnFrom return value	Undefined Behavior	**Informational**

Detailed Findings

1. Attackers could mint more Fertilizer than intended due to an unused variable

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`

Exploit Scenario

Recapitalization of the Beanstalk protocol is almost complete; only 100 units of Fertilizer for sale remain. Eve, a malicious user, calls `mintFertilizer()` with an `amount` of 10 million, significantly over-funding the system. Because the Fertilizer supply increased significantly above the theoretical maximum, other users are entitled to a much smaller yield than expected.

Recommendations

Short term, use `_amount` instead of `amount` as the parameter in the functions that are called after `mintFertilizer()`.

Long term, thoroughly document the expected behavior of the `FertilizerFacet` contract and the properties (invariants) it should enforce, such as "token amounts above the maximum recapitalization threshold cannot be sold." Expand the unit test suite to test that these properties hold.

2. Lack of a two-step process for ownership transfer	
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`

Exploit Scenario

The owner of the Beanstalk contracts is a community controlled multisignature wallet. The community agrees to upgrade to an on-chain voting system, but the wrong address is mistakenly provided to its call to `transferOwnership()`, permanently misconfiguring the system.

Recommendations

Short term, implement a two-step process to transfer contract ownership, in which the owner proposes a new address and then the new address executes a call to accept the role, completing the transfer.

Long term, identify and document all possible actions that can be taken by privileged accounts and their associated risks. This will facilitate reviews of the codebase and prevent future mistakes.

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

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.

Exploit Scenario

The Barn Raise is a total success: the `MAX_RAISE` amount is hit, meaning that 77 million Fertilizer tokens have been minted.

Alice, a malicious user, notices the underflow risk in the `remaining()` function; she sends one USDC to the `CUSTODIAN` wallet, triggering the underflow and causing the function to return the `maxuint256` instead of `MAX_RAISE`. As a result, the sale continues even though the `MAX_RAISE` amount was reached.

Other users, not knowing that the Barn Raise should be complete, continue to successfully mint Fertilizer tokens until the bug is discovered and the system is paused to address the issue. While no Beanstalk funds are lost as a result of this exploit, the users who continued minting Fertilizer after the `MAX_RAISE` was reached lose all the gas funds they spent.

Recommendations

Short term, add a check in the `remaining()` function so that it returns `0` if `USDC.balanceOf(CUSTODIAN)` is greater than or equal to `MAX_RAISE`. This will prevent the underflow from being triggered.

Because the function depends on the `CUSTODIAN`'s balance, it is still possible for someone to send USDC directly to the `CUSTODIAN` wallet and reduce the amount of "available" Fertilizer; however, attackers would lose their money in the process, meaning that there are no incentives to perform this kind of action.

Long term, thoroughly document the expected behavior of the `FertilizerPremint` contract and the properties (invariants) it should enforce, such as "no tokens can be minted once the `MAX_RAISE` is reached." Expand the unit test suite to test that these properties hold.

4. Risk of Fertilizer id collision that could result in loss of funds

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.

Exploit Scenario

Imagine the following scenario:

- It is currently the first season; the `bpf` is `0` and the `humidity` is 40%. Alice mints 100 Fertilizer tokens with an `id` of `41` (the sum of 1 and the `bpf` (`0`) and `humidity` (`40`) values), and `lastBpf` is set to `0`.
- Some time goes by, and it is now the third season; the `bpf` is `35` and the `humidity` is 5%. Alice mints one additional Fertilizer token with an `id` of `41` (the sum of 1 and the `bpf` (`35`) and `humidity` (`5`) values), and `lastBpf` is set to `35`.

Because of the second mint, the `lastBpf` field of Alice's Fertilizer tokens is overridden, making her lose a substantial amount of the yield she was entitled to:

- Using the formula for calculating the number of BEAN tokens that users are entitled to, shown in figure 4.5, Alice's original yield at "maturity" would have been 4,100 tokens:
  - `deltaBpf = id - lastBpf = 41 - 0 = 41`
  - `balance = 100`
  - `beans received = deltaBpf * balance = 41 * 100 = 4100`
- As a result of the overridden `lastBpf` field, Alice's yield instead ends up being only 606 tokens:
  - `deltaBpf = id - lastBpf = 41 - 35 = 6`
  - `balance = 101`
  - `beans received = deltaBpf * balance = 6 * 101 = 606`

Recommendations

Short term, separate the role of the `id` into two separate variables for the token index and `endBpf`. That way, the index can be optimized to prevent collisions, while `endBpf` can accurately represent the data it needs to represent.

Alternatively, modify the relevant code so that when an `id` collision occurs, it either reverts or redeems the previous Fertilizer first before minting the new tokens.

However, these alternate remedies could introduce new edge cases or could result in a degraded user experience; if either alternate remedy is implemented, it would need to be thoroughly documented to inform the users of its particular behavior.

Long term, thoroughly document the expected behavior of the associated code and include regression tests to prevent similar issues from being introduced in the future.

Additionally, exercise caution when using one variable to serve two purposes. Gas savings should be measured and weighed against the increased complexity. Developers should be aware that performing optimizations could introduce new edge cases and increase the code's complexity.

5. The sunrise() function rewards callers only with the base incentive	
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`

Exploit Scenario
Gas prices suddenly increase to the point that it is no longer profitable to call `sunrise()`. Given the lack of an increasing incentive, the function goes uncalled for several hours, preventing the system from reacting to changing market conditions.

Recommendations
Short term, pass the `incentive` value instead of `amount` into the `mint()` function call.

Long term, thoroughly document the expected behavior of the `SeasonFacet` contract and the properties (invariants) it should enforce, such as "the caller of the `sunrise()` function receives the right incentive." Expand the unit test suite to test that these properties hold.

Additionally, thoroughly document how the system would be affected if the `sunrise()` function were not called for a long period of time (e.g., in times of extreme network congestion).

Finally, determine whether the Beanstalk team should rely exclusively on third parties to call the `sunrise()` function or whether an alternate system managed by the Beanstalk team should be adopted in addition to the current system. For example, an alternate system could involve an off-chain monitoring system and a trusted execution flow.

6. Solidity compiler optimizations can be problematic	
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.

Exploit Scenario
A latent or future bug in Solidity compiler optimizations—or in the Emscripten transpilation to `solc-js`—causes a security vulnerability in the Beanstalk contracts.

Recommendations
Short term, measure the gas savings from optimizations and carefully weigh them against the possibility of an optimization-related bug.

Long term, monitor the development and adoption of Solidity compiler optimizations to assess their maturity.

7. Lack of support for external transfers of nonstandard ERC20 tokens

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.

Exploit Scenario

Alice uses the `TokenFacet` contract to transfer nonstandard ERC20 tokens that return `false` on failure to another contract. However, Alice accidentally inputs an amount higher than her balance. The transaction is successfully executed, but because there is no check of the `false` return value, Alice does not know that her tokens were not transferred.

Recommendations

Short term, use the `safeERC20` library for external token transfers.

Long term, thoroughly review and document all interactions with arbitrary tokens to prevent similar issues from being introduced in the future.

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

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`

Exploit Scenario

A new smart contract that integrates with the `MarketplaceFacet` contract is deployed. This contract has features allowing it to manage users' pods on their behalf.

Alice approves the contract so that it can manage her pods.

Some time passes, and Alice calls one of the smart contract's functions, which requires Alice to transfer ownership of her plot to the contract. Because Alice has already approved the smart contract, it can perform the transfer on her behalf. To do so, it calls the

`transferPlot()` function in the `MarketplaceFacet` contract; however, this call reverts because Alice has an open listing for the pods that the contract is trying to transfer.

Recommendations

Short term, add a new input to `_cancelPodListing()` that is equal to `msg.sender` if the caller is the owner of the listing, but equal to the pod owner if the caller is a user who was approved by the owner.

Long term, thoroughly document the expected behavior of the `MarketplaceFacet` contract and the properties (invariants) it should enforce, such as "plot transfers initiated by users with an allowance cancel the owner's listing." Expand the unit test suite to test that these properties hold.

9. Users can sow more Bean tokens than are burned

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`.

Exploit Scenario

Eve, a malicious user, spots the vulnerability in the `FieldFacet` contract and waits until Bean is below its peg and the protocol starts issuing soil.

Bean finally goes below its peg, and the protocol issues 1,000 soil.

Eve deposits a single Bean token into the contract by calling the `transferToken()` function in the `TokenFacet` contract. She then calls the `sow()` function with `amount` equal to `1000` and `mode` equal to `INTERNAL_TOLERANT`.

The `sow()` function is executed, sowing 1,000 Bean tokens but burning only a single token.

Recommendations

Short term, modify the relevant code so that users' Bean tokens are burned before the accounting for the soil and pods are updated and so that, if the `mode` field is not `EXTERNAL`, the `amount` returned by `receiveToken()` is used as the input to `LibDibbler.sow()`.

Long term, thoroughly document the expected behavior of the `FieldFacet` contract and the properties (invariants) it should enforce, such as "the `sow()` function always sows as many Bean tokens as were burned." Expand the unit test suite to test that these properties hold.

10. Pods may never ripen	
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.

Exploit Scenario
Bean goes through an extended period of increasing demand, overextending its supply. Then, demand for Bean tokens slowly and steadily declines, and the pod line grows in length. At a certain point, some users decide that their time-weighted rate of return is unfavorable or too uncertain despite the promised high yields. Instead of lending their Bean tokens to the protocol, they sell.

Recommendations
Explore options for backing Bean's value with an offer that is guaranteed to eventually be fulfilled.

11. Bean and the offer backing it are strongly correlated	
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.

Exploit Scenario

Bean goes through an extended period of increasing demand, overextending its supply. Then, the demand for Bean slowly and steadily declines as some users lose interest in holding Bean. These same users also lose interest in receiving Bean tokens in the future, so rather than loaning their tokens to Beanstalk to earn a very high Bean-denominated yield, they sell.

Recommendations

Explore options for backing Bean's value with an offer that is not correlated with demand for Bean.

12. Ability to whitelist assets uncorrelated with Bean price, misaligning governance incentives

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.

Exploit Scenario
Stalk holders vote to whitelist ETH as a depositable asset. They proceed to deposit ETH and begin receiving shares of rewards, including 3CRV tokens acquired during SoPs. Governance is now incentivized to increase the supply of Bean as high as possible to obtain more 3CRV rewards, which eventually results in an overextension of the Bean supply and a subsequent price crash. After the Bean price crashes, Stalk holders withdraw their deposited ETH and 3CRV rewards. Because ETH is not strongly correlated with the price of Bean, they do not suffer financial loss as a result of the crash.

Alternatively, because of the lack of on-chain enforcement of off-chain votes, the above scenario could occur if the community multisignature wallet whitelists ETH, even if no related vote occurred.

Recommendations

Do not allow any assets that are not strongly correlated with the price of Bean to be whitelisted. Additionally, implement monitoring systems that provide alerts every time a new asset is whitelisted.

13. Unchecked burnFrom return value

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`

Exploit Scenario

Alice notices that the `burnFrom` function is implemented incorrectly in the `unripeToken` contract. She calls `ripen` with an `amount` greater than her unripe token balance and is able to receive the contract's entire balance of underlying tokens.

Recommendations

Short term, add an `assert` statement to ensure that users who call `ripen` have sufficient balance to burn the given `amount` of unripe tokens.

Long term, implement all security-critical assertions on user-supplied input in the beginning of external functions. Do not rely on untrusted code to perform required safety checks or to behave as expected.

Summary of Recommendations

The Beanstalk protocol is a work in progress with multiple planned iterations. Trail of Bits recommends that Beanstalk address the findings detailed in this report and take the following additional steps prior to deployment:

- Rename Solidity variables throughout the codebase so that they more closely reflect their underlying responsibilities; apply farm-themed language only to non-security-critical components, such as the user interface. Using farm-themed language to describe the system can be a valuable community-building tool; however, describing smart contracts using terminology that is unrelated to their logic greatly increases the risk that security problems could go unnoticed.

- Rearchitect the system to split up the state. The current system provides all state variables to all functions. This makes it difficult to determine a given function's responsibilities and increases the risk that one component could cause a problem in an unrelated component. We recommend following the principle of least privilege by exposing groups of functions only to the state variables they need.

  For example, move all of the marketplace functions into a separate contract that contains only the state variables related to marketplace management (e.g., `podListing` and `podOrders`). Instead of modifying the `podOrders` state variable directly from the field, the `cancelOrder` function could be called on the marketplace contract. This would both allow the system to be deployed in smaller pieces and grant developers the ability to review all marketplace-related logic in one place while ensuring that no other function in the system can make invalid changes to the marketplace state.

- Either upgrade to Solidity 0.8 or use `SafeMath` for all arithmetic operations. If certain arithmetic operations are intended to be performed without overflow or underflow checks, add inline comments and documentation to clarify developer intentions.

- Create a specification for each state variable involved in the system's arithmetic operations (starting with the most critical ones); each specification should contain the following:

  - A description of the variable
  - The boundaries of the variable
  - The origin of the variable (i.e., who is able to modify it)

  Once the specifications are defined, use them to drive testing.

- Expand the current unit test suite to cover all the system's arithmetic operations (especially those involving unchecked math). Adding more unit tests to the current test suite and adding the suite to a continuous integration pipeline would help prevent implementation-level bugs such as TOB-BEANS-003 and TOB-BEANS-005 from being reintroduced.
- Integrate automated end-to-end tests to help identify complex arithmetic bugs and other errors that may not be caught by isolated unit tests. Adding a suite of fuzz tests to a continuous integration pipeline would help prevent complicated logic bugs such as TOB-BEANS-004 from being reintroduced.
- Discuss economic issues TOB-BEANS-010, TOB-BEANS-011, and TOB-BEANS-012 with the Beanstalk community to decide which findings pose an acceptable level of risk and how others could be mitigated.

A. 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
Economic	Risky or misaligned economic incentives
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.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Front-Running Resistance	The system's resistance to front-running attacks
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Glossary and Recommendations

This appendix provides a non-exhaustive list of farm-themed terms used throughout the Beanstalk protocol and our recommendations for improving the documentation related to these terms.

Glossary

- **Bean**: Beanstalk's stablecoin
  Bean is the protocol's core ERC20 token, pegged to the US dollar; the protocol is designed to maintain the peg.

- **Silo**: Asset storage
  The silo is the protocol's asset storage and is responsible for allocating rewards to depositors. Investors deposit assets to and withdraw assets from the silo.

- **Seeds**: Reward allocation
  Seeds entitle investors to rewards. Investors who deposit assets into the silo receive seeds that entitle them to rewards in Bean tokens. Seeds are forfeited upon withdrawal.

- **Stalk**: Governance token
  The Stalk token is an ERC20 governance token that is rewarded to investors who deposit whitelisted assets and is forfeited upon withdrawal. Stalk holders are entitled to non-Bean rewards that are generated when Bean tokens are sold in the market to put downward pressure on the price of Bean.

- **Roots**: Shares of governance tokens
  Roots represent the proportion of governance tokens that a user holds. Roots are used internally to allocate rewards, similar to the role of DEX LP tokens in representing shares of the underlying pool.

- **Sow**: To lend
  To sow Bean tokens is to lend them to the Beanstalk protocol. The tokens are burned once they are sown, decreasing the total supply, and are re-minted upon repayment.

- **Pods**: Loans
  Pods can be thought of as non-callable bonds without a predefined maturity date. Pods are non-transferable vouchers that the protocol will repay on a FIFO basis when the price of Bean is above its peg.

- **Plots**: Loan bundles
  Plots are bundles of pods that were created together. Plots can be bought or sold for Bean tokens through the integrated marketplace.

- **Pod line**: Queue of lenders
  The pod line is the queue of lenders awaiting repayment. The protocol repays the first lender in the pod line from the available supply of newly minted Bean tokens when the price of Bean is above its value peg.

- **Soil**: Loan request
  Soil represents a loan request from the Beanstalk protocol. One Bean token can be loaned to the protocol for each soil unit available. Soil acts as a Bean sink.

- **Harvest**: To repay
  To harvest is to repay a loan. Once a loan is repayable, the lender can harvest to receive his principal and interest.

- **Weather**: Interest
  Weather determines how much interest a lender will receive after repayment and is set when Bean tokens are loaned.

- **Season**: Epoch
  A season is a time interval (of approximately one hour) over which interest rates are constant and Bean's weighted average distance from its value peg is measured.

- **Rain**: Dampener
  Rain represents the time during which rewards are delayed. When the price of Bean is high and system debt is low, rewards are delayed to help disincentivize sudden spikes in demand.

- **Pod rate**: Debt/supply ratio
  The pod rate is the protocol's total Bean debt divided by the total Bean supply. It acts as a high-level health factor for the protocol; a low ratio (pod rate) indicates a healthy system.

- **Season of plenty (SoP)**: Market sale of beans
  An SoP occurs following a prolonged period of high Bean price and low system debt. During an SoP, Bean tokens are sold on the market and the proceeds of the sale are distributed to investors.

- **Unripe tokens**: Pre-exploit assets
  Bean and LP tokens that existed in the protocol before the 2022 governance hack are considered "unripe" while the protocol is being recapitalized. Holders of unripe

tokens can burn them to receive a pro rata share of the assets minted during the fundraiser.

- **Ripen**: To vest
  To ripen is to burn unripe tokens in exchange for a share of the assets minted during the fundraiser. Unripe tokens can be ripened early with a penalty; as Bean spends time above its peg, this penalty decreases.

Recommendations

The above glossary is not exhaustive; there are many other internal variables with farm-themed names that would benefit from plain English definitions. We recommend that the Beanstalk team continue to improve and expand its existing glossary and, in doing so, consider the following guidelines:

- In writing a plain English definition for an internal variable, start with a summary of 1–3 *words* that outlines the meaning of the term in the context of the Beanstalk protocol in the simplest possible way. This micro-summary should act as a retrieval cue that can remind knowledgeable users of the term's details; therefore, it can sacrifice some precision in the interest of concision.

- Following this micro-summary, provide a summary of 1–3 *sentences* describing in greater detail how the term relates to the rest of the system. Low-level details are not necessary; however, references to implementation details in the code or mathematical details in the white paper would be beneficial.

- In writing each definition, ensure that it can stand alone. Avoid using other farm-themed terms within each term's definition. Users who are new to the protocol should be able to read a definition and gain a better understanding of the term without needing to recursively look up other terms.

- Define all of the acronyms used in the protocol. Including the acronyms that users will encounter in the protocol alongside their spelled-out versions (e.g., including "(SoP)" after the term "season of plenty") will make it easier for users to search for acronyms and find the relevant definitions.

D. Code Quality Recommendations

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

- Invert the `require` statement in `Listing.sol#L64`, from `0 < pricePerPod` to `pricePerPod > 0`, to improve the code's readability.
- Rename the `getBpf()` function in `LibFertilizer.sol#L58` to `getEndBpf()`. It calculates the `endBpf` of the Fertilizer token, but its current name suggests that it retrieves the current season's `bpf`, which could cause confusion.
- Consider removing the unused `_status` state variable in `Silo.sol#L45` if it is not planned to be used by any feature in the future.
- Consider using a more verbose naming convention for the state variables so that their meanings are clearer to users. In many instances, the names assigned to state variables do not immediately convey their meaning. For example, the `pods` state variable represents "the total pods ever minted"; a name such as `totalPods` or `totalPodMints` would more effectively convey that meaning.

E. Incident Response Recommendations

This section provides recommendations on formulating an incident response plan.

- **Identify the parties (either specific people or roles) responsible for implementing the mitigations when an issue occurs (e.g., deploying smart contracts, pausing contracts, upgrading the front end, etc.).**
- **Document internal processes for addressing situations in which a deployed remedy does not work or introduces a new bug.**
    - Consider documenting a plan of action for handling failed remediations.
- **Clearly describe the intended contract deployment process.**
- **Outline the circumstances under which Beanstalk will compensate users affected by an issue (if any).**
    - Issues that warrant compensation could include an individual or aggregate loss or a loss resulting from user error, a contract flaw, or a third-party contract flaw.
- **Document how the team plans to stay up to date on new issues that could affect the system; awareness of such issues will inform future development work and help the team secure the deployment toolchain and the external on-chain and off-chain services that the system relies on.**
    - Identify sources of vulnerability news for each language and component used in the system, and subscribe to updates from each source. Consider creating a private Discord channel in which a bot will post the latest vulnerability news; this will provide the team with a way to track all updates in one place. Lastly, consider assigning certain team members to track news about vulnerabilities in specific components of the system.
- **Determine when the team will seek assistance from external parties (e.g., auditors, affected users, other protocol developers, etc.) and how it will onboard them.**
    - Effective remediation of certain issues may require collaboration with external parties.
- **Define contract behavior that would be considered abnormal by off-chain monitoring solutions.**

It is best practice to perform periodic dry runs of scenarios outlined in the incident response plan to find omissions and opportunities for improvement and to develop "muscle memory." Additionally, document the frequency with which the team should perform dry runs of various scenarios, and perform dry runs of more likely scenarios more regularly. Create a template to be filled out with descriptions of any necessary improvements after each dry run.

F. Token Integration Checklist

The following checklist provides recommendations for interactions with arbitrary tokens. Every unchecked item should be justified, and its associated risks, understood. For an up-to-date version of the checklist, see `crytic/building-secure-contracts`.

For convenience, all Slither utilities can be run directly on a token address, such as the following:

```
slither-check-erc 0xdac17f958d2ee523a2206206994597c13d831ec7 TetherToken --erc erc20
slither-check-erc 0x06012c8cf97BEaD5deAe237070F9587f8E7A266d KittyCore --erc erc721
```

To follow this checklist, use the below output from Slither for the token:

```
slither-check-erc [target] [contractName] [optional: --erc ERC_NUMBER]
slither [target] --print human-summary
slither [target] --print contract-summary
slither-prop . --contract ContractName # requires configuration, and use of Echidna
and Manticore
```

General Considerations

- ❏ **The contract has a security review.** Avoid interacting with contracts that lack a security review. Check the length of the assessment (i.e., the level of effort), the reputation of the security firm, and the number and severity of the findings.
- ❏ **You have contacted the developers.** You may need to alert their team to an incident. Look for appropriate contacts on `blockchain-security-contacts`.
- ❏ **They have a security mailing list for critical announcements.** Their team should advise users (like you!) when critical issues are found or when upgrades occur.

Contract Composition

- ❏ **The contract avoids unnecessary complexity.** The token should be a simple contract; a token with complex code requires a higher standard of review. Use Slither's `human-summary` printer to identify complex code.
- ❏ **The contract uses `SafeMath`.** Contracts that do not use `SafeMath` require a higher standard of review. Inspect the contract by hand for `SafeMath` usage.
- ❏ **The contract has only a few non-token-related functions.** Non-token-related functions increase the likelihood of an issue in the contract. Use Slither's `contract-summary` printer to broadly review the code used in the contract.

- ❏ **The token has only one address.** Tokens with multiple entry points for balance updates can break internal bookkeeping based on the address (e.g., `balances[token_address][msg.sender]` may not reflect the actual balance).

Owner Privileges

- ❏ **The token is not upgradeable.** Upgradeable contracts may change their rules over time. Use Slither's `human-summary` printer to determine whether the contract is upgradeable.
- ❏ **The owner has limited minting capabilities.** Malicious or compromised owners can abuse minting capabilities. Use Slither's `human-summary` printer to review minting capabilities, and consider manually reviewing the code.
- ❏ **The token is not pausable.** Malicious or compromised owners can trap contracts relying on pausable tokens. Identify pausable code by hand.
- ❏ **The owner cannot blacklist the contract.** Malicious or compromised owners can trap contracts relying on tokens with a blacklist. Identify blacklisting features by hand.
- ❏ **The team behind the token is known and can be held responsible for abuse.** Contracts with anonymous development teams or teams that reside in legal shelters require a higher standard of review.

ERC20 Tokens

ERC20 Conformity Checks

Slither includes a utility, `slither-check-erc`, that reviews the conformance of a token to many related ERC standards. Use `slither-check-erc` to review the following:

- ❏ **`Transfer` and `transferFrom` return a boolean.** Several tokens do not return a boolean on these functions. As a result, their calls in the contract might fail.
- ❏ **The `name`, `decimals`, and `symbol` functions are present if used.** These functions are optional in the ERC20 standard and may not be present.
- ❏ **`Decimals` returns a `uint8`.** Several tokens incorrectly return a `uint256`. In such cases, ensure that the value returned is below 255.
- ❏ **The token mitigates the known ERC20 race condition.** The ERC20 standard has a known ERC20 race condition that must be mitigated to prevent attackers from stealing tokens.

Slither includes a utility, `slither-prop`, that generates unit tests and security properties that can discover many common ERC flaws. Use `slither-prop` to review the following:

- ❏ **The contract passes all unit tests and security properties from `slither-prop`.** Run the generated unit tests and then check the properties with Echidna and Manticore.

Risks of ERC20 Extensions

The behavior of certain contracts may differ from the original ERC specification. Conduct a manual review of the following conditions:

- ❏ **The token is not an ERC777 token and has no external function call in `transfer` or `transferFrom`.** External calls in the transfer functions can lead to reentrancies.
- ❏ **`Transfer` and `transferFrom` should not take a fee.** Deflationary tokens can lead to unexpected behavior.
- ❏ **Potential interest earned from the token is taken into account.** Some tokens distribute interest to token holders. This interest may be trapped in the contract if not taken into account.

Token Scarcity

Reviews of token scarcity issues must be executed manually. Check for the following conditions:

- ❏ **The supply is owned by more than a few users.** If a few users own most of the tokens, they can influence operations based on the tokens' repartition.
- ❏ **The total supply is sufficient.** Tokens with a low total supply can be easily manipulated.
- ❏ **The tokens are located in more than a few exchanges.** If all the tokens are in one exchange, a compromise of the exchange could compromise the contract relying on the token.
- ❏ **Users understand the risks associated with a large amount of funds or flash loans.** Contracts relying on the token balance must account for attackers with a large amount of funds or attacks executed through flash loans.
- ❏ **The token does not allow flash minting.** Flash minting can lead to substantial swings in the balance and the total supply, which necessitate strict and comprehensive overflow checks in the operation of the token.