

Pinto - Sow Blueprint Security review

Version 1.0

Table of Contents

1 About Egis Security **3**

2 About Pinto - Sow Blueprint feature **3**

3 Disclaimer **3**

4 Risk classification **4**
4.1 Impact 4
4.2 Likelihood 4
4.3 Actions required by severity level 4

5 Executive summary **5**

6 Findings **6**
6.1 High risk 6
6.1.1 SiloHelpers::withdrawbeansFromSources - Will withdraw incorrect `plan.amounts` when calling `removeLiquidityOneToken`. 6
6.2 Medium risk 7
6.2.1 `getTokensAscendingPrice` can be manipulated to grief publisher. . . . 7
6.2.2 If lp token gets de-whitelisted, user defined token indexes will be influenced. . 8
6.2.3 Impossible to sow under certain conditions 9
6.3 Low risk 11
6.3.1 Exploiter can honeypot operator to execute blueprint with negative tip amount. 11
6.3.2 Consider making `SLIPPAGE_RATIO` configurable 11
6.3.3 Can call `publishRequisition` using canceled blueprints 11
6.4 Informational 13
6.4.1 Consider making `withdrawBeansFromSources` accept `WithdrawalPlan` to safe gas, instead of calling `getWithdrawalPlan` two times 13
6.4.2 Consider making `SiloHelpers::tip` accept `fromMode` and `toMode` in order to make it more generic. 13
6.4.3 Stale approval of beanToken in `SiloHelpers::withdrawBeansFromSources`. 13
6.4.4 Consider implementing QuickSort, instead of BubbleSort to save gas for all sorting functions. 13
6.4.5 Comment above `sendTokenToInternalBalance` is wrong. 13

1 About Egis Security

Egis Security is a team of experienced smart contract researchers, who strive to provide the best smart contract security services possible to DeFi protocols.

The team has a proven track record on public auditing platforms like Code4rena, Sherlock, and Cantina, earning top placements and rewards exceeding $170,000. They have identified over 150 high and medium-severity vulnerabilities in both public contests and private audits.

2 About Pinto - Sow Blueprint feature

Pinto Blueprints define off-chain, EIP-712-signed instructions that let Farmers delegate complex on-chain actions to third-party Tractor Operators. For mowing, two contracts are used:

- SiloHelpers – a general-purpose helper contract reusable across different blueprint types.
- SowBlueprintV0 – a mowing-specific blueprint that encodes sowing logic.

Tractor Operators execute these Blueprints permissionlessly via the `tractor(...)` function, enabling decentralized, intent-based automation on behalf of Farmers.

3 Disclaimer

Audits are a time, resource, and expertise bound effort where trained experts evaluate smart contracts using a combination of automated and manual techniques to identify as many vulnerabilities as possible. Audits can show the presence of vulnerabilities **but not their absence**.

4 Risk classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4.1 Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost or a functionality of the protocol is affected.
- **Low** - any kind of unexpected behaviour that's not so critical.

4.2 Likelihood

- **High** - direct attack vector; the cost is relatively low to the amount of funds that can be lost.
- **Medium** - only conditionally incentivized attack vector, but still relatively likely.
- **Low** - too many or too unlikely assumptions; provides little or no incentive.

4.3 Actions required by severity level

- **Critical** - client **must** fix the issue.
- **High** - client **must** fix the issue.
- **Medium** - client **should** fix the issue.
- **Low** - client **could** fix the issue.

5 Executive summary

Overview

Project Name	Pinto - Sow Blueprint
Repository	https://github.com/pinto-org/protocol
Commit hash	c9eb6ff25dd0ae87d454ab4e5b719af3ca402f72
Resolution	378c67ea1dcfdd184a227c24a42dc81f8ff136cb
Documentation	https://docs.pinto.money/farm/toolshed/tractor
Methods	Manual review

Scope

contracts/ecosystem/SowBlueprintv0.sol
contracts/ecosystem/SiloHelpers.sol
contracts/ecosystem/PriceManipulation.sol
contracts/ecosystem/PerFunctionPausable.sol
contracts/ecosystem/OperatorWhitelist.sol

Issues Found

Critical risk	0
High risk	1
Medium risk	3
Low risk	3
Informational	2

6 Findings

6.1 High risk

6.1.1 SiloHelpers::withdrawbeansFromSources - Will withdraw incorrect `plan.amounts` when calling `removeLiquidityOneToken`.

Severity: *Medium risk*

Context: SiloHelpers.sol#L306-L307

Description: In `SiloHelpers::withdrawBeansFromSources` when we withdraw tokens != bean, the code always uses `plan.amounts[i][0]` when calling `transferInternalTokenFrom` and `removeLiquidityOneToken`

```
// Transfer LP tokens to this contract's external balance
                beanstalk.transferInternalTokenFrom(
                    IERC20(sourceToken),
                    account,
                    address(this),
                    plan.amounts[i][0], // Use first amount since it's the total LP
                        amount
                    LibTransfer.To.EXTERNAL
                );

                // Then remove liquidity to get Beans
                IERC20(sourceToken).approve(sourceToken, plan.amounts[i][0]);
                uint256 beansOut = IWell(sourceToken).removeLiquidityOneToken(
                    plan.amounts[i][0],
                    IERC20(beanToken),
                    plan.availableBeans[i], // Min amount out should be the
                        remaining amount needed
                    address(this),
                    type(uint256).max
                );
```

This will only be correct if `plan.amounts[i]` is an array with 1 element, for all cases where it's more than 1, the function will only withdraw partially and `amountWithdrawn` won't be enough.

Recommendation: Transfer the entire `plan.amounts` and `removeLiquidityOneToken` with the all the `plan.amounts`

Resolution: Fixed in the commit: pinto-org/protocol@2141449.

6.2 Medium risk

6.2.1 `getTokensAscendingPrice` can be manipulated to grief publisher.

Severity: *Medium risk*

Context: SiloHelpers.sol#L113-L114

Description: `SiloHelpers` handles key values for `tokenIndices` as a strategy switches. Currently we have two strategies: `ascending price strategy` and `ascending seeds strategy`. `ascending price strategy` means we should fill the `tokenIndices` array with whitelisted tokens, ordered by USD price per pinto from lowest to highest. However, `getTokensAscendingPrice` uses `beanstalkPrice.price()`, which is `non-manipulation resistant` because it uses the spot price for price calculations. This means that an exploiter can manipulate the price for wells, where the pinto is cheaper and move them to the end of the array, giving priority to the more expensive pools. The exploit will succeed as long as publisher has lp assets for the expensive wells and we fill `targetAmount` before we reach the manipulated wells:

```
        for (uint256 i = 0; i < tokenIndices.length && vars.remainingBeansNeeded >
            0; i++) {
            require(tokenIndices[i] < vars.whitelistedTokens.length, "Invalid token
                index");
```

In `withdrawBeansFromSources` we only check `priceManipulation.isValidSlippage` for the pools on the withdraw queue. This will pass because the exploiter has bricked `getTokensAscendingPrice` logic, but `plan.sourceTokens` don't include the manipulated pools. Impact is that a victim which expect best results (withdraw from pool with lowest price per pinto) expectations are violated.

Recommendation: One solution is to check all well lp tokens for `isValidSlippage` in `getTokensAscendingPrice`. To safe gas, potentially you can set a flag that slippage is already checked, so you don't have to do it in `withdrawBeansFromSources`

Resolution: Fixed.

6.2.2 If lp token gets de-whitelisted, user defined token indexes will be influenced.

Severity: *Medium risk*

Context: SiloHelpers.sol#L131-L132

Description: When a user sign a blueprint to sow, he can specify `sourceTokenIndices` in `SowParams` stuct:

```
    /**
     * @notice Struct to hold sow parameters
     * @param sourceTokenIndices Indices of source tokens to withdraw from
     * @param sowAmounts Amounts for sowing
     * @param minTemp Minimum temperature required for sowing
     * @param maxPodlineLength Maximum podline length allowed
     * @param maxGrownStalkPerBdv Maximum grown stalk per BDV allowed
     * @param runBlocksAfterSunrise Number of blocks to wait after sunrise before
        executing
     */
    struct SowParams {
        uint8[] sourceTokenIndices;
        SowAmounts sowAmounts;
        uint256 minTemp;
        uint256 maxPodlineLength;
        uint256 maxGrownStalkPerBdv;
        uint256 runBlocksAfterSunrise;
    }
```

Those indexes are based on `beanstalk.getWhitelistedTokens()` in time of the creation of the blueprint. And example of whitelisted tokens is:

```
- PINTO:WETH Well -> 0
- PINTO:cbETH Well -> 1
- PINTO:cbBTC Well -> 2
- PINTO:USDC Well -> 3
```

They are stored in `s.sys.silo.whitelistStatuses` under the above indexes. Now if a user wants to sign a blueprint that allows a withdraw only from `PINTO:cbETH Well` and `PINTO:cbBTC Well`, he will fill `sourceTokenIndices` with {1; 2}. But if something happens with `PINTO:cbETH Well` token and team wants to delist it, deploy and whitelist a new one, user blueprint will no longer satisfy his initial intentions because beanstalk.getWhitelistedTokens() will return a result resulting in an unexpected `PINTO:USDC Well` lp tokens withdraw:

```
- PINTO:WETH Well -> 0
- PINTO:cbBTC Well -> 1
- PINTO:USDC Well -> 2
```

Impact: Unexpected behavior for the victim (wrong lp funds withdrawn and used for sowing)

Recommendation: Implement a helper func `getWhitelistedFromStatuses` and use `beanstalk.getWhitelistStatuses`, fetch the token index from there and check if it is still whitelisted

Resolution: Fixed.

6.2.3 Impossible to sow under certain conditions

Severity: *Medium risk*

Context: SowBlueprintv0.sol#L353-L357

Description: `totalAmountToSow` represents how much pinto the publisher is willing to sow. He can also specify a min and a max amount to sow. It's also possible to not sow the entire `totalAmountToSow` at once, this is determined in `determineTotalAmountToSow`. We can see that if `totalAmountToSow > availableSoil` then we limit `totalAmountToSow` to `availableSoil`. If this happens, then we have to set a `counterValue` in order to know how much more pinto we can sow, if we have any leftover. Here we set it to the `totalAmountToSow` from the `params`, how much the user wants to sow based on the blueprint. We can see that if `totalAmountToSow > availableSoil` then we limit `totalAmountToSow` to `availableSoil`. If this happens, then we have to set a `counterValue` in order to know how much more pinto we can sow, if we have any leftover. Here we set it to the `totalAmountToSow` from the `params`, how much the user wants to sow based on the blueprint.

```
        if (vars.counterValue == 0) {
            updatePintoLeftToSowCounter(
                vars.orderHash,
                params.sowParams.sowAmounts.totalAmountToSow
            );
            vars.counterValue = params.sowParams.sowAmounts.totalAmountToSow;
        }
```

Then we update it to represent how much pinto is leftover to sow. And then we `sowWithMin`. It's possible to have a situation where it's impossible to sow the leftover pinto, even if everything allows it.

Example:

1. Alice creates a bluerpint with: `totalAmountToSow = 100 minAmountToSowPerSeason = 30 maxAmountToSowPerSeason = 10000`
2. An operator executes Alice's blueprint.
3. Currently `availableSoil = 80`, so in `determineTotalAmountToSow` we limit `totalAmountToSow = 80`
4. Initially `counterValue` is set to `100`, then it gets updated to `20`

```
if (vars.counterValue - vars.totalAmountToSow == 0) {
            updatePintoLeftToSowCounter(vars.orderHash, type(uint256).max);
        } else {
            updatePintoLeftToSowCounter(vars.orderHash, vars.counterValue - vars.
                totalAmountToSow);
        }
```

`updatePintoLeftToSowCounter(vars.orderHash, 100 - 80 (this is the limited value in determineTotalAmountToSow))` 5. We sow `80` pinto and we have `20` leftover. 6. The next season comes and the operator executes the blueprint again. 7. `determineTotalAmountToSow` sets `totalAmountToSow` to `20`, since `20 < 100`:

```
if (counterValue < totalAmountToSow) {
            totalAmountToSow = counterValue;
        }
```

8. The rest of the checks pass and we hit:

```
// Sow the withdrawn beans
        beanstalk.sowWithMin(
            vars.totalAmountToSow,
            params.sowParams.minTemp,
            params.sowParams.sowAmounts.minAmountToSowPerSeason,
            LibTransfer.From.INTERNAL
        );
```

9. `sowWithMin` is called with `vars.totalAmountToSow = 20` and `params.sowParams.sowAmounts.minAmountToSowPerSeason = 30`
10. This will always revert inside `sowWithMin`

```
function sowWithMin(
        uint256 beans,
        uint256 minTemperature,
        uint256 minSoil,
        LibTransfer.From mode
    ) internal returns (uint256 pods) {
        // 'soil' is the remaining Soil
        (uint256 soil, uint256 _morningTemperature, bool abovePeg) =
            _totalSoilAndTemperature();

        require(soil >= minSoil && beans >= minSoil, "Field: Soil Slippage");
```

`beans >= minSoil, 20 >= 30` reverts

No matter when the blueprint is executed again or under different market conditions, the leftover pinto cannot be sown through the blueprint itself, the publisher has to manually sow the leftover.

Recommendation: If `counterValue < minAmountToSowPerSeason`, override `minAmountToSowPerSeason` to `counterValue`

Resolution: Acknowledged.

6.3 Low risk

6.3.1 Exploiter can honeypot operator to execute blueprint with negative tip amount.

Severity: *Low risk*

Context: SiloHelpers.sol#L829-L839

Description: If the tip amount is negative, an operator should previously approve publisher to spend `tipAmount` of his behalf. If the operator has approved user for a larger amount (to safe gas from batching transactions every time), the publisher can steal the `tipAmount`, without waiting operator executing the blueprint.

Recommendation: Consider documenting the risk for operators that execute blueprints with negative tip amounts.

Resolution: Fixed by documenting the risk.

6.3.2 Consider making `SLIPPAGE_RATIO` configurable

Severity: *Low risk*

Context: SiloHelpers.sol#L26-L27

Description: Make `SLIPPAGE_RATIO` configurable by an admin. As such the value can be configured depending on the volatility of the market. Currently if the market becomes very volatile, blueprints might not get executed because the slippage is always hit.

Recommendation: Make `SIPPAGE_RATIO` configurable instead of hardcoding it.

Resolution: Fixed.

6.3.3 Can call `publishRequisition` using canceled blueprints

Severity: *Low risk*

Context: AbstractSiloedPinto.sol#L14

Description: `cancelBlueprint` sets the `maxNonce` of a blueprint to `type(uint256).max`, making the blueprint unrunnable through `tractor`.

```
/**
     * @notice Destroy existing blueprint
     */
    function cancelBlueprint(
        LibTractor.Requisition calldata requisition
    ) external fundsSafu noNetFlow noSupplyChange verifyRequisition(requisition)
        nonReentrant {
        require(msg.sender == requisition.blueprint.publisher, "TractorFacet: not
            publisher");
        LibTractor._cancelBlueprint(requisition.blueprintHash);
        emit CancelBlueprint(requisition.blueprintHash);
    }

/**
```

```
     * @notice Cancel blueprint.
     * @dev set blueprintNonce to type(uint256).max
     * @param blueprintHash blueprint hash
     */
    function _cancelBlueprint(bytes32 blueprintHash) internal {
        _tractorStorage().blueprintNonce[blueprintHash] = type(uint256).max;
    }
```

When `tractor` is called, the tx will revert in `runBlueprint`:

```
  modifier runBlueprint(LibTractor.Requisition calldata requisition) {
        require(
            LibTractor._getBlueprintNonce(requisition.blueprintHash) <
                requisition.blueprint.maxNonce,
            "TractorFacet: maxNonce reached"
        );
```

But `publishRequisition` can still be called, emitting it's data on-chain. Canceled/destroyed blueprints should not be interacted with and thus `publishRequisition` should not be used with canceled blueprints.

Recommendation: Add check to ensure that the blueprint was not canceled.

Resolution: Fixed.

6.4 Informational

6.4.1 Consider making `withdrawBeansFromSources` accept `WithdrawalPlan` to safe gas, instead of calling `getWithdrawalPlan` two times

Severity: *Informational*

Resolution: Fixed.

6.4.2 Consider making `SiloHelpers::tip` accept `fromMode` and `toMode` in order to make it more generic.

Severity: *Informational*

Description: Consider making `SiloHelpers::tip` accept `fromMode` and `toMode` in order to make it more generic as `SiloHelpers` is supposed to be a more generic helper contract in order to fit in with future blueprints.

Resolution: Fixed.

6.4.3 Stale approval of beanToken in `SiloHelpers::withdrawBeansFromSources`.

Severity: *Informational*

Description: Stale approval of beanToken in `SiloHelpers::withdrawBeansFromSources`, if `mode == LibTransfer.To.EXTERNAL`. The code does a normal `ERC20.transfer` to account, so no approval is needed. Consider calling `IERC20(beanToken).approve(address(beanstalk), beansOut);` only if `mode == LibTransfer.To.INTERNAL`.

Resolution: Fixed.

6.4.4 Consider implementing QuickSort, instead of BubbleSort to save gas for all sorting functions.

Severity: *Informational*

Resolution: Acknowledged.

6.4.5 Comment above `sendTokenToInternalBalance` is wrong.

Severity: *Informational*

Description: `Operators can check for this function in bytecode via the selector (0x9b8911f6)The actual selector is 0xca1e71ae` - The actual selector is `0xca1e71ae`.

Resolution: Fixed.