TitanX Farm Security Review

Pashov Audit Group

Conducted by: Said, ast3ros, pontifex

September 28th - October 7th

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **dragon-software-devs/TitanX-Farms** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About TitanX Farm

The TitanX Farm Protocol is enables users to provide liquidity on UniswapV3 and earn rewards in TINC tokens, while implementing a buy-and-burn mechanism to support TINC's value.

5. Risk Classification

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

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - 181af1ffd7a8bcfe57c0bdeadccabf1b709619a9

fixes review commit hash - e7d6476452e2aa4b937f292da906c65e427e0962

Scope

The following smart contracts were in scope of the audit:

- `LiquidityAmounts`
- `Oracle`
- `PathDecoder`
- `PoolAddress`
- `PositionKey`
- `PositionValue`
- `TickMath`
- `Constants`
- `Farms`
- `InputTokens`
- `FarmKeeper`
- `TINC`
- `UniversalBuyAndBurn`

7. Executive Summary

Over the course of the security review, Said, ast3ros, pontifex engaged with TitanX to review TitanX Farm. In this period of time a total of **11** issues were uncovered.

Protocol Summary

Protocol Name	TitanX Farm
Repository	https://github.com/dragon-software-devs/TitanX-Farms
Date	September 28th - October 7th
Protocol Type	Farming protocol

Findings Count

Severity	**Amount**
High	1
Medium	1
Low	9
Total Findings	**11**

Summary of Findings

ID	Title	Severity	Status
[H-01]	Loss of fees due to rounding down	High	Resolved
[M-01]	TWAP price manipulation	Medium	Acknowledged
[L-01]	Potential out of gas	Low	Acknowledged
[L-02]	Potential division by zero	Low	Resolved
[L-03]	Detailed information about a specific farm can be outdated	Low	Resolved
[L-04]	Initial last reward time can be changed even if farming is not started	Low	Resolved
[L-05]	Permit signatures cannot be canceled before deadlines	Low	Resolved
[L-06]	Not checking if the value is 0	Low	Resolved
[L-07]	Unnecessary query the TWAP price	Low	Resolved
[L-08]	Fees may be incorrectly handled	Low	Resolved
[L-09]	Lack of min toBuy and toBurn for buyAndBurn	Low	Acknowledged

8. Findings

8.1. High Findings

[H-01] Loss of fees due to rounding down

Severity

Impact: High

Likelihood: Medium

Description

The `FarmKeeper` contract uses a `SCALE_FACTOR` of `1e12` when accumulating fees per share of liquidity. For tokens with low decimals (e.g. USDC with 6 decimals) and large liquidity amounts, this can lead to precision loss and rounding down to 0, resulting in permanent loss of fees for users.

Let's consider a token with 6 decimals such as USDC.

The issue occurs because:

- amount0/amount1 can be as low as 1 unit (e.g. 1e6 for USDC)
- liquidity is a uint128, so can be up to ~3.4e38
- SCALE_FACTOR is only 1e12

This means the calculation can result in: `1e6 * 1e12 / 1e38 = 1e-20` which rounds down to 0 due to integer division.

We can see that if the token is in 6 decimals, with the liquidity amount > 1e18 (doesn't need to be in e38), the incremental fee per share for the token is rounded down to 0.

Combined with the `updateFarm`, it can be called by anyone to collect the fees. Malicious actors can exploit this by calling `updateFarm` frequently to force fee collection when amounts are small.

```
function _collectFees(Farm storage farm) private {

    // Handle token0
    if (isInputToken0) {
    ...
    } else {
      farm.accFeePerShareForToken0 += Math.mulDiv
        (amount0, Constants.SCALE_FACTOR, liquidity);
    }

    // Handle token1
    if (isInputToken1) {
    ...
    } else {
      farm.accFeePerShareForToken1 += Math.mulDiv
        (amount1, Constants.SCALE_FACTOR, liquidity);
    }
  }
```

The impact is fees collected when liquidity is high will be permanently lost and unclaimable by users.

Recommendations

Increase the `SCALE_FACTOR` value to 18 to prevent the calculation from rounding down to 0.

8.2. Medium Findings

[M-01] TWAP price manipulation

Severity

Impact: High

Likelihood: Low

Description

The `_getTwaPrice` function in FarmKeeper and `getQuote` function in UniversalBuyAndBurn are vulnerable to price manipulation in Uniswap V3 pools with low observation cardinality. This vulnerability can lead to inaccurate price calculations, potentially resulting in unfair token swaps or liquidity provisions.

When getting the twap price to prevent slippage in liquidity providing and swapping, the `_getTwaPrice` and `getQuote` functions fallback to the oldest observation if the requested time window is greater than the available price history.

It's a problem because it doesn't consider the pool cardinality. In newly created pools with cardinality initialized to 1, the oldest observation can be manipulated to be the current `block.timestamp`, setting the TWAP price to the current (potentially manipulated) price. Because in uniswap v3 pool, an oracle observation is written by the first swap or liquidity provision in the block. Let's say before the UniversalBuyAndBurn contract performs the swap, a malicious actor front-runs and swaps to manipulate the price because the cardinality is 1, and the oldest observation is 0.

Another consequence is that the `oldestObservation` can be equal to `cardianility * 12 seconds (time per block)` if liquidity changes each block. It means that the TWAP period can be shorter than intended if `oldestObservation < secondsAgo`, potentially increasing slippage risk.

```
function _getTwaPrice(address id, uint32 priceTwa) private view returns
    (uint160 slotPrice, uint160 twaPrice) {
    ...
    // Default TWA price to slot
    twaPrice = slotPrice;

    uint32 secondsAgo = uint32(priceTwa * 60);
    uint32 oldestObservation = 0;

    // Load oldest observation if cardinality greater than zero
    oldestObservation = OracleLibrary.getOldestObservationSecondsAgo(id);

    // Limit to oldest observation (fallback)
    if (oldestObservation < secondsAgo) {
      secondsAgo = oldestObservation; // @audit fallback to oldest observation
    }
    ...
  }
```

```
function getQuote(
    address inputTokenAddress,
    address outputTokenAddress,
    uint24 fee,
    uint256 twap,
    uint256 inputTokenAmount
  ) public view returns (uint256 quote, uint32 secondsAgo) {
    ...
    secondsAgo = uint32(twap * 60);
    uint32 oldestObservation = 0;

    // Load oldest observation if cardinality greather than zero
    oldestObservation = OracleLibrary.getOldestObservationSecondsAgo
      (poolAddress);

    // Limit to oldest observation
    if (oldestObservation < secondsAgo) {
      secondsAgo = oldestObservation; // @audit fallback to oldest observation
    }

    // If TWAP is enabled and price history exists, consult oracle
    if (secondsAgo > 0) {
      // Consult the Oracle Library for TWAP
      (int24 arithmeticMeanTick, ) = OracleLibrary.consult
        (poolAddress, secondsAgo);

      // Convert tick to sqrtPriceX96
      sqrtPriceX96 = TickMath.getSqrtRatioAtTick(arithmeticMeanTick);
    }
    ...
  }
```

Recommendations

It's recommended to either:

  - Revert if `oldestObservation < secondsAgo` to ensure the full intended TWAP period is used.
  - Require a minimum cardinality (e.g., 10) for pools to be eligible for farming or buy-and-burn operations. Reference: https://docs-v1.euler.finance/euler-protocol/eulers-default-parameters#uniswap-observation-cardinality

8.3. Low Findings

[L-01] Potential out of gas

The `massUpdateFarms` function iterates over all farms and updates them. This presents a potential risk of reverting due to out-of-gas errors if the number of farms becomes very large. If the function reverts when `collectFees` is set to false, key functionalities such as `enableFarm` and `setAllocation` could be blocked, as they rely on `massUpdateFarms`.

Adding farms to the `_farms` array is restricted to admin, and when `collectFees` is set to `false`, each iteration costs only about 1300 gas. This makes the likelihood of an out-of-gas error occurring very low.

However, the admin should be mindful of this potential issue and consider implementing solutions such as removing inactive farms when they're no longer needed.

```
function massUpdateFarms(bool collectFees) public nonReentrant {
    uint256 length = _farms.length();

    // Iterate all farms and update them
    for (uint256 idx = 0; idx < length; idx++) {
      Farm storage farm = _farms.at(idx);
      _updateFarm
      //(farm, collectFees); // @audit Potential out of gas if the number of farms is
    }
  }
```

[L-02] Potential division by zero

In the `enableFarm` function, the `params.allocPoints` is validated to be less than `MAX_ALLOCATION_POINTS`, but there's no check to ensure `totalAllocPoints` remains above zero after adding a new farm. This could theoretically lead to a division by zero error in the `_updateFarm` function when calculating rewards, if `totalAllocPoints` somehow becomes zero and the farm has non-zero liquidity.

```
function enableFarm(AddFarmParams calldata params) external restricted {
    ...
    _validateAllocPoints
    //(params.allocPoints); // @audit only verify that allocPoints is not greater than
    ...
  }
```

```
function _updateFarm(Farm storage farm, bool collectFees) private {
    ...
    uint256 incentiveTokenReward = Math.mulDiv
    //( // @audit Potential division by zero if totalAllocPoints is 0
      timeMultiplier * Constants.INCENTIVE_TOKEN_PER_SECOND,
      farm.allocPoints,
      totalAllocPoints
    );
    ...
  }
```

It's recommended to add a check to ensure `totalAllocPoints` will be greater than zero after adding the new farm in the `enableFarm` function.

```
function enableFarm(AddFarmParams calldata params) external restricted {
  // ...
  _validateAllocPoints(params.allocPoints);
+ require
+ (totalAllocPoints + params.allocPoints > 0, "Total allocation points must be greater
  // ...
  totalAllocPoints += params.allocPoints;
}
```

[L-03] Detailed information about a specific farm can be outdated

The `farmView` function can return outdated information because it contains storage variables whose values depend on the time period since the last update and does not include pending pool fees. Consider implementing a view functionality that returns actual information similar to the `userView` function.

```
function farmView(address id) public view returns (FarmView memory) {
    if (!_farms.contains(id)) revert InvalidFarmId();

    Farm storage farm = _farms.get(id);

    (uint160 slotPrice, ) = _getTwaPrice(farm.id, 0);
    (
      uint256balanceToken0,
      uint256balanceToken1
    ) = LiquidityAmounts.getAmountsForLiquidity(
      slotPrice,
      TickMath.getSqrtRatioAtTick(Constants.MIN_TICK),
      TickMath.getSqrtRatioAtTick(Constants.MAX_TICK),
      farm.lp.liquidity
    );

    return
      FarmView({
        id: farm.id,
        poolKey: farm.poolKey,
        lp: farm.lp,
        allocPoints: farm.allocPoints,
        lastRewardTime: farm.lastRewardTime,
>>      accIncentiveTokenPerShare: farm.accIncentiveTokenPerShare,
>>      accFeePerShareForToken0: farm.accFeePerShareForToken0,
>>      accFeePerShareForToken1: farm.accFeePerShareForToken1,
>>      protocolFee: farm.protocolFee,
        priceTwa: farm.priceTwa,
        slippage: farm.slippage,
        balanceToken0: balanceToken0,
        balanceToken1: balanceToken1
      });
  }
```

[L-04] Initial last reward time can be changed even if farming is not started

The initial value of the `farm.lastRewardTime` variable can be set to `startTime` and then changed to `block.timestamp` even if `block.timestamp < startTime` by the `_updateFarm` function. Consider not changing `farm.lastRewardTime` to `block.timestamp` value when it is less than `startTime`.

```
function enableFarm(AddFarmParams calldata params) external restricted {
    PoolAddress.PoolKey memory poolKey = PoolAddress.getPoolKey
      (params.tokenA, params.tokenB, params.fee);

    // Compute pool address to check for duplicates
    address id = PoolAddress.computeAddress(Constants.FACTORY, poolKey);

    // Check for duplicates
    if (_farms.contains(id)) revert DuplicatedFarm();

    _validateAllocPoints(params.allocPoints);
    _validatePriceTwa(params.priceTwa);
    _validateSlippage(params.slippage);
    _validateProtocolFee(params.protocolFee);

    // Update all farms but do not collect fees as only
    // the incentive token allocations are affected by enabling a new farm
>>  massUpdateFarms(false);

    // Append new farm
    _farms.add(
      Farm({
        id: id,
        poolKey: poolKey,
        lp: LP({tokenId: 0, liquidity: 0}),
        allocPoints: params.allocPoints,
>>
        lastRewardTime: block.timestamp > startTime ? block.timestamp : startTime,
        accIncentiveTokenPerShare: 0,
        accFeePerShareForToken0: 0,
        accFeePerShareForToken1: 0,
        protocolFee: params.protocolFee,
        priceTwa: params.priceTwa,
        slippage: params.slippage
      })
    );

    totalAllocPoints += params.allocPoints;
    emit FarmEnabled(id, params);
  }
<...>
  function massUpdateFarms(bool collectFees) public nonReentrant {
    uint256 length = _farms.length();

    // Iterate all farms and update them
    for (uint256 idx = 0; idx < length; idx++) {
      Farm storage farm = _farms.at(idx);
>>    _updateFarm(farm, collectFees);
    }
  }
<...>
  function _updateFarm(Farm storage farm, bool collectFees) private {
    // Total liquidity
    uint256 liquidity = farm.lp.liquidity;
    if (liquidity == 0) {
>>    farm.lastRewardTime = block.timestamp;
      return;
    }
```

[L-05] Permit signatures cannot be canceled before deadlines

TINC permit signature offers the signer the option to create an EIP-712 signature. After signing this signature, a signer might want to cancel it, but will not be able to do so. This is because the contract is based on OpenZeppelin's `ERC20Permit.sol` in which the function to increase nonce does not exist and the _useNonce function within `Nonces` is marked internal.

Consider introducing a public function that signers can directly use to consume their nonce, thereby canceling the signatures.

```
function useNonce() external returns (uint256) {
        return _useNonce(msg.sender);
    }
```

[L-06] Not checking if the value is 0

When `_collectFees` is triggered and the collected fees are 0, and the tokens are the input token inside `buyAndBurn`, it still triggers `_safeTransferToken` with providing 0 value.

```
function _collectFees(Farm storage farm) private {
    // Cache State Variables
    uint256 liquidity = farm.lp.liquidity;
    INonfungiblePositionManager manager = INonfungiblePositionManager
      (Constants.NON_FUNGIBLE_POSITION_MANAGER);

    // Collect the maximum amount possible of both tokens

      farm.lp.tokenId,
      address(this),
      type(uint128).max,
      type(uint128).max
    );
    (uint256 amount0, uint256 amount1) = manager.collect(params);

    // Identify tokens which are accepted as input for buy and burn
    bool isInputToken0 = buyAndBurn.isInputToken(farm.poolKey.token0);
    bool isInputToken1 = buyAndBurn.isInputToken(farm.poolKey.token1);

    // Handle token0
    if (isInputToken0) {
      uint256 protocolFee = 0;

      if (farm.protocolFee > 0) {
        protocolFee = Math.mulDiv(amount0, farm.protocolFee, Constants.BASIS);
        protocolFees[farm.poolKey.token0] += protocolFee;
      }

      // Send core tokens to the buy and burn contract
>>>   _safeTransferToken(farm.poolKey.token0, address
  (buyAndBurn), amount0 - protocolFee);
    } else {
      farm.accFeePerShareForToken0 += Math.mulDiv
        (amount0, Constants.SCALE_FACTOR, liquidity);
    }

    // Handle token1
    if (isInputToken1) {
      uint256 protocolFee = 0;

      if (farm.protocolFee > 0) {
        protocolFee = Math.mulDiv(amount1, farm.protocolFee, Constants.BASIS);
        protocolFees[farm.poolKey.token1] += protocolFee;
      }

      // Send core tokens to the buy and burn contract
>>>   _safeTransferToken(farm.poolKey.token1, address
  (buyAndBurn), amount1 - protocolFee);
    } else {
      farm.accFeePerShareForToken1 += Math.mulDiv
        (amount1, Constants.SCALE_FACTOR, liquidity);
    }
  }
```

If the tokens revert on a 0 transfer value, the operation could revert and cause operations that depend on `_collectFees` to always revert.

Consider skipping the fee calculation and transfers when the collected amounts are 0.

[L-07] Unnecessary query the TWAP price

when `getAmountsForLiquidity` is called, it will call `_getDesiredAmountsForLiquidity` to get `amount0` and `amount1`.

```
function getAmountsForLiquidity(
    address id,
    uint128 liquidity
  ) external view returns
    (address token0, address token1, uint256 amount0, uint256 amount1) {
    if (!_farms.contains(id)) revert InvalidFarmId();

    Farm storage farm = _farms.get(id);
>>> (amount0, amount1, , ) = _getDesiredAmountsForLiquidity(farm, liquidity);

    token0 = farm.poolKey.token0;
    token1 = farm.poolKey.token1;
  }
```

Inside `_getDesiredAmountsForLiquidity`, it retrieves the slot0 price and TWAP price, then calculates the amount0/amount1 based on those prices. However, `getAmountsForLiquidity` only requires amounts calculated using the slot0 price.

```
function _getDesiredAmountsForLiquidity(
    Farm storage farm,
    uint128 liquidity
  ) private view returns (
    uint256desiredAmount0,
    uint256desiredAmount1,
    uint256minAmount0,
    uint256minAmount1
  ) private view returns
    (uint256 desiredAmount0, uint256 desiredAmount1, uint256 minAmount0, uint256 minAm
    (uint160 slotPrice, uint160 twaPrice) = _getTwaPrice
      (farm.id, farm.priceTwa);
    // Calculate desired amounts based on current slot price
    (desiredAmount0, desiredAmount1) = LiquidityAmounts.getAmountsForLiquidity(
      slotPrice,
      TickMath.getSqrtRatioAtTick(Constants.MIN_TICK),
      TickMath.getSqrtRatioAtTick(Constants.MAX_TICK),
      liquidity
    );

    // Calculate minimal amounts based on TWA price for slippage protection
    (minAmount0, minAmount1) = LiquidityAmounts.getAmountsForLiquidity(
      twaPrice,
      TickMath.getSqrtRatioAtTick(Constants.MIN_TICK),
      TickMath.getSqrtRatioAtTick(Constants.MAX_TICK),
      liquidity
    );

    // Apply slippage
    minAmount0 = (minAmount0 *
      (Constants.BASIS - farm.slippage)) / Constants.BASIS;
    minAmount1 = (minAmount1 *
      (Constants.BASIS - farm.slippage)) / Constants.BASIS;
  }
```

Consider to modify `getAmountsForLiquidity` to the following, so it doesn't not necessarily query TWAP price :

```
function getAmountsForLiquidity(
    address id,
    uint128 liquidity
  ) external view returns
    (address token0, address token1, uint256 amount0, uint256 amount1) {
    if (!_farms.contains(id)) revert InvalidFarmId();

-    Farm storage farm = _farms.get(id);
-    (amount0, amount1, , ) = _getDesiredAmountsForLiquidity(farm, liquidity);
+    (uint160 slotPrice, ) = _getTwaPrice(id, 0);
+    (amount0, amount1) = LiquidityAmounts.getAmountsForLiquidity(
+      slotPrice,
+      TickMath.getSqrtRatioAtTick(Constants.MIN_TICK),
+      TickMath.getSqrtRatioAtTick(Constants.MAX_TICK),
+      liquidity
+    );

    token0 = farm.poolKey.token0;
    token1 = farm.poolKey.token1;
  }
```

A similar scenario also occurs inside `getLiquidityForAmount`. Update `getLiquidityForAmount` as follows:

```
function getLiquidityForAmount(
    address id,
    address token,
    uint256 amount
  ) external view returns (
    addresstoken0,
    addresstoken1,
    uint128liquidity,
    uint256amount0,
    uint256amount1
  ) external view returns
    (address token0, address token1, uint128 liquidity, uint256 amount0, uint256 amoun
    if (!_farms.contains(id)) revert InvalidFarmId();

    Farm storage farm = _farms.get(id);
    token0 = farm.poolKey.token0;
    token1 = farm.poolKey.token1;

    // Get prices
-    (uint160 slotPrice, ) = _getTwaPrice(farm.id, farm.priceTwa);
+    (uint160 slotPrice, ) = _getTwaPrice(farm.id, 0);
    uint160 sqrtRatioAX96 = TickMath.getSqrtRatioAtTick(Constants.MIN_TICK);
    uint160 sqrtRatioBX96 = TickMath.getSqrtRatioAtTick(Constants.MAX_TICK);

    if (token == token0) {
      liquidity = LiquidityAmounts.getLiquidityForAmount0
        (slotPrice, sqrtRatioBX96, amount);
    } else if (token == token1) {
      liquidity = LiquidityAmounts.getLiquidityForAmount1
        (sqrtRatioAX96, slotPrice, amount);
    } else {
      revert InvalidTokenId();
    }

    // Calculate amounts based on the slot price
    (amount0, amount1) = LiquidityAmounts.getAmountsForLiquidity(
      slotPrice,
      TickMath.getSqrtRatioAtTick(Constants.MIN_TICK),
      TickMath.getSqrtRatioAtTick(Constants.MAX_TICK),
      liquidity
    );
  }
```

[L-08] Fees may be incorrectly handled

The `setDisabled` function in the `UniversalBuyAndBurn` contract allows toggling whether an input token is considered active for the buy and burn mechanism. However, this function does not trigger fee collection in the FarmKeeper contract before changing the disabled state. This can lead to incorrect fee handling in the following scenarios:

- When disabling a previously enabled token:
    - Accumulated fees since the last collection up to the disabling time are not sent to the BuyAndBurn contract.
    - Protocol fees are undercalculated for this period.
- When enabling a previously disabled token:
    - Accumulated fees during the disabled period are suddenly sent to the BuyAndBurn contract.
    - Protocol fees are overcalculated, including fees from the disabled period.

```
function setDisabled
    (address inputTokenAddress, bool disabled) external restricted {
    if (!_inputTokens.contains
      (inputTokenAddress)) revert InvalidInputTokenAddress();
    _inputTokens.get
    //(inputTokenAddress).disabled = disabled; // @audit should collect fees in FarmKe
    emit DisabledUpdated(inputTokenAddress, disabled);
  }
```

```
function _collectFees(Farm storage farm) private {
    ...
    // Identify tokens which are accepted as input for buy and burn
    bool isInputToken0 = buyAndBurn.isInputToken(farm.poolKey.token0);
    bool isInputToken1 = buyAndBurn.isInputToken(farm.poolKey.token1);

    ...
  }
```

The impact is inconsistent fee collection and distribution, and inaccurate protocol fee calculations

It's recommended to collect fees in the `setDisabled` function before disabling the token.

[L-09] Lack of min `toBuy` and `toBurn` for `buyAndBurn`

When `buyAndBurn` is called, it calculates the `toBuy` and `toBurn` amounts based on the `burnPercentage` value. As long as one of the values is non-zero, the operation will be processed.

```
function buyAndBurn(address inputTokenAddress) external nonReentrant {
    // Ensure processing a valid input token
    if (!_inputTokens.contains(inputTokenAddress)) {
      revert InvalidInputTokenAddress();
    }
    InputToken storage inputTokenInfo = _inputTokens.get(inputTokenAddress);

    // prevent contract accounts (bots) from calling this function
    // becomes obsolete with EIP-3074, there are other measures in
    // place to make MEV attacks inefficient (cap per swap, interval control)
    if (msg.sender != tx.origin) {
      revert InvalidCaller();
    }

    if (inputTokenInfo.paused) {
      revert InputTokenPaused();
    }

    // keep a minium gap of interval between each call
    // update stored timestamp
    if
      (block.timestamp - inputTokenInfo.lastCallTs <= inputTokenInfo.interval) {
      revert CooldownPeriodActive();
    }
    inputTokenInfo.lastCallTs = block.timestamp;

    // Get the input token amount to buy and incentive fee
    // this call will revert if there are no input tokens left in the contract
    (uint256 toBuy, uint256 toBurn, uint256 incentiveFee) = _getAmounts
      (inputTokenInfo);

>>> if (toBuy == 0 && toBurn == 0) {
      revert NoInputTokenBalance();
    }
    // ...
}
```

This allows griefers to donate a dust amount of the input token to the contract, trigger `buyAndBurn`, and update the token's `lastCallTs`, causing it to wait for the interval, even though an improper `buyAndBurn` was executed.

Consider adding a minimum value for `toBuy` and `toBurn` when `buyAndBurn` is triggered. Also, consider setting `lastCallTs` to `block.timestamp` when a new input token is enabled to minimize griefing when the token is enabled for the first time.

```
function enableInputToken
  (EnableInputToken calldata params) external restricted {
    // ...

    _inputTokens.add(
      InputToken({
        id: params.id,
        totalTokensUsedForBuyAndBurn: 0,
        totalTokensBurned: 0,
        totalIncentiveFee: 0,
-         lastCallTs: 0,
+         lastCallTs: block.timestamp,
        capPerSwap: params.capPerSwap,
        interval: params.interval,
        incentiveFee: params.incentiveFee,
        burnProxy: IBurnProxy(params.burnProxy),
        burnPercentage: params.burnPercentage,
        priceTwa: params.priceTwa,
        slippage: params.slippage,
        path: params.path,
        paused: params.paused,
        disabled: false
      })
    );

    emit InputTokenEnabled(params.id, params);
  }
```