CD SECURITY

AUDIT REPORT

TitanX - Farms
September 2024

Prepared by
tsvetanovv
minquanym

Introduction

A time-boxed security review of the **TitanX-Farms** protocol was done by **CD Security**, with a focus on the security aspects of the application's implementation.

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.

About **TitanX-Farms**

The TitanX-Farms project is a DeFi protocol focused on liquidity farming and token buy-and-burn mechanisms. The protocol includes two main components:

1. FarmKeeper Contract: Manages Uniswap V3 liquidity farms, allowing users to deposit and withdraw liquidity. Users earn incentive tokens (like TINC) based on the liquidity they provide. The contract also handles fee collection and distribution, ensuring slippage protection and utilizing robust security measures.
2. UniversalBuyAndBurn Contract: Implements a buy-and-burn mechanism for a specific output token using various ERC-20 input tokens. It supports multiple input tokens with configurable parameters and includes features like TWAP (Time-Weighted Average Price) for price quotes, slippage protection, and user incentives.

These components work together to optimize liquidity provision and maintain token value, making TitanX-Farms a comprehensive solution for decentralized finance participants.

Severity classification

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

Impact - the technical, economic, and reputation damage of a successful attack

Likelihood - the chance that a particular vulnerability gets discovered and exploited

Severity - the overall criticality of the risk

Security Assessment Summary

***review commit hash* - bfc117928f8a75e20dc810de867e28193efe9f91**

Scope

Everything in the `./contracts` directory was within the scope of this audit.

The following number of issues were found, categorized by their severity:

- Critical & High: 1 issues
- Medium: 6 issues
- Low: 5 issues

Findings Summary

ID	Title	Severity	Status
[H-01]	`secondsPerLiquidityCumulativesDelta` calculation can overflow	High	Fixed
[M-01]	Fee growth inside could overflow causing a function to revert	Medium	Fixed
[M-02]	One of the intended functionalities of `withdraw()` does not work	Medium	Fixed
[M-03]	Some functions should be able to overflow	Medium	Fixed
[M-04]	`_safeTransferToken()` does not check the returned value of tokens	Medium	Fixed
[M-05]	Some functions don't have deadline check	Medium	Fixed
[L-01]	The protocol cannot remove the input token when needed	Low	Fixed
[L-02]	Incorrect Slippage limit check	Low	Fixed
[L-03]	`setIncentiveFee()` does not check if `incentiveFee` is zero	Low	Fixed
[L-04]	Incomplete validation of TWA period in `setPriceTwa()`	Low	Fixed
[L-05]	`_collectFees()` should be called before changing `protocolFee`	Low	Fixed
[L-06]	Unsafe casting	Low	Fixed

Detailed Findings

[H-01] `secondsPerLiquidityCumulativesDelta` calculation can overflow

Severity

Impact: High

Likelihood: Medium

Description

The function `consult()` depends on the seconds per liquidity cumulative values from the Uniswap V3 Pool to calculate `harmonicMeanLiquidity`. This value represents the amount of second liquidity inside a tick range that is "active". It is noted that they utilize a function similar to the one implemented in Uniswap V3. However, this function implicitly relies on underflow/overflow when calculating `secondsPerLiquidityCumulativesDelta`. If underflow is prevented, certain operations that depend on fee growth may revert.

Due to the use of a different Solidity version, these calculations could revert due to overflow. Although this implementation resembles the library provided by Uniswap, it is important to note that Uniswap uses Solidity 0.6, which does not revert on overflow, whereas this contract is using Solidity 0.8.

Reference: https://github.com/code-423n4/2023-05-maia-findings/issues/505

```
(int56[] memory tickCumulatives, uint160[] memory
secondsPerLiquidityCumulativeX128s) = IUniswapV3Pool(pool)
  .observe(secondsAgos);

int56 tickCumulativesDelta = tickCumulatives[1] - tickCumulatives[0];
uint160 secondsPerLiquidityCumulativesDelta =
secondsPerLiquidityCumulativeX128s[1] - // @audit should use unchecked
  secondsPerLiquidityCumulativeX128s[0];
```

The similar issue also exists when subtracting `poolFeeGrowthInside` in the function `_fees()`.

```
amount0 =
  Math.mulDiv(
    poolFeeGrowthInside0LastX128 -
feeParams.positionFeeGrowthInside0LastX128,
    feeParams.liquidity,
    FixedPoint128.Q128
  ) +
  feeParams.tokensOwed0;

amount1 =
  Math.mulDiv(
    poolFeeGrowthInside1LastX128 -
feeParams.positionFeeGrowthInside1LastX128,
    feeParams.liquidity,
    FixedPoint128.Q128
  ) +
  feeParams.tokensOwed1;
```

Recommendations

Consider using `unchecked` when calculating `secondsPerLiquidityCumulativesDelta` or using Solidity version smaller than 0.8.

[M-01] Fee growth inside could overflow causing a function to revert

Severity

Impact: Medium

Likelihood: Medium

Description

When operations need to calculate the fee growth of a Uniswap V3 position, they utilize a function similar to the one implemented in Uniswap V3. However, as noted in this known issue, the contract implicitly relies on underflow/overflow when calculating fee growth. If underflow is prevented, certain operations that depend on fee growth may revert.

Due to the use of a different Solidity version, these calculations could revert due to overflow. Although this implementation resembles the library provided by Uniswap, it is important to note that Uniswap uses Solidity 0.6, which does not revert on overflow, whereas this contract is using Solidity 0.8.

```
if (tickCurrent < tickLower) {
  feeGrowthInside0X128 = lowerFeeGrowthOutside0X128 -
upperFeeGrowthOutside0X128;
  feeGrowthInside1X128 = lowerFeeGrowthOutside1X128 -
upperFeeGrowthOutside1X128;
} else if (tickCurrent < tickUpper) {
  uint256 feeGrowthGlobal0X128 = pool.feeGrowthGlobal0X128();
  uint256 feeGrowthGlobal1X128 = pool.feeGrowthGlobal1X128();
  feeGrowthInside0X128 = feeGrowthGlobal0X128 - lowerFeeGrowthOutside0X128
- upperFeeGrowthOutside0X128;
  feeGrowthInside1X128 = feeGrowthGlobal1X128 - lowerFeeGrowthOutside1X128
- upperFeeGrowthOutside1X128;
} else {
  feeGrowthInside0X128 = upperFeeGrowthOutside0X128 -
lowerFeeGrowthOutside0X128;
  feeGrowthInside1X128 = upperFeeGrowthOutside1X128 -
lowerFeeGrowthOutside1X128;
}
```

Recommendations

Consider using `unchecked` when calculating `feeGrowthInside0X128` and `feeGrowthInside1X128`.

[M-02] One of the intended functionalities of `withdraw()` does not work

Severity

Impact: Low

Likelihood: High

Description

The function `withdraw()` in `FarmKeeper.sol` allows a user to withdraw liquidity from a farm:

```
  function withdraw(address id, uint128 liquidity) external nonReentrant {
    if (!_farms.contains(id)) revert InvalidFarmId();

    Farm storage farm = _farms.get(id);
    User storage user = _farms.user(id, msg.sender);

    if (user.liquidity < liquidity || liquidity == 0) {
      revert InvalidLiquidityAmount();
    }

    // Update farms and collect fees
    _updateFarm(farm, true);

    // Calculate pending rewards and fees
    uint256 pendingIncentiveTokens = Math.mulDiv(
      user.liquidity,
      farm.accIncentiveTokenPerShare,
      Constants.SCALE_FACTOR
    ) - user.rewardCheckpoint;
    uint256 pendingFeeToken0 = Math.mulDiv(user.liquidity,
farm.accFeePerShareForToken0, Constants.SCALE_FACTOR) -
      user.feeCheckpointToken0;
    uint256 pendingFeeToken1 = Math.mulDiv(user.liquidity,
farm.accFeePerShareForToken1, Constants.SCALE_FACTOR) -
      user.feeCheckpointToken1;

    // Update state
    user.liquidity -= liquidity;
    user.rewardCheckpoint = Math.mulDiv(user.liquidity,
farm.accIncentiveTokenPerShare, Constants.SCALE_FACTOR);
    user.feeCheckpointToken0 = Math.mulDiv(user.liquidity,
farm.accFeePerShareForToken0, Constants.SCALE_FACTOR);
    user.feeCheckpointToken1 = Math.mulDiv(user.liquidity,
farm.accFeePerShareForToken1, Constants.SCALE_FACTOR);


    // Decrease liquidity
    (uint256 amountToken0, uint256 amountToken1) =
_decreaseLiquidity(farm, liquidity, msg.sender);


    // Payout pending tokens
    if (pendingIncentiveTokens > 0) {
      // Mint Incentive Tokens to user
      incentiveToken.mint(msg.sender, pendingIncentiveTokens);
      emit IncentiveTokenDistributed(id, msg.sender,
pendingIncentiveTokens);
    }
    if (pendingFeeToken0 > 0) {
      _safeTransferToken(farm.poolKey.token0, msg.sender,
pendingFeeToken0);
      emit FeeDistributed(id, msg.sender, farm.poolKey.token0,
pendingFeeToken0);
    }
    if (pendingFeeToken1 > 0) {
      _safeTransferToken(farm.poolKey.token1, msg.sender,
pendingFeeToken1);
      emit FeeDistributed(id, msg.sender, farm.poolKey.token1,
pendingFeeToken1);
    }

    emit Withdraw(id, msg.sender, liquidity, amountToken0, amountToken1);
  }
```

The intended behavior, as described in the NatSpec comments for the `withdraw()` function, states that:

> Setting liquidity to zero allows to pull fees and incentive tokens without modifying the liquidity position by the user.

This implies that a user should be able to set the `liquidity` parameter to `0` to claim accumulated fees and incentive tokens without changing their liquidity position in the farm.

However, the following check in the function prevents this behavior:

```
if (user.liquidity < liquidity || liquidity == 0) {
    revert InvalidLiquidityAmount();
}
```

If a user passes `0` as the liquidity parameter, the function reverts with an `InvalidLiquidityAmount` error. This prevents users from pulling their fees and incentive tokens without modifying their liquidity, contradicting the intended behavior stated in the NatSpec.

As a result, users are unable to claim their accumulated rewards without modifying their liquidity, which can create friction, particularly for users who wish to claim rewards without adjusting their position in the farm.

Recommendations

Modify the check to allow `liquidity == 0` as a valid input when a user only wants to pull fees and incentive tokens without modifying their liquidity:

```
if (user.liquidity < liquidity) {
    revert InvalidLiquidityAmount();
}
```

[M-03] Some functions should be able to overflow

Severity

Impact: Medium

Likelihood: Medium

Description

`TickMath.sol` library is missing unchecked blocks, which causes it to incorrectly revert on phantom overflow. This library was taken from https://github.com/Uniswap/v3-core/blob/main/contracts/libraries/TickMath.sol, but you can see solidity version is < 0.8.0, meaning that the execution didn't revert when the overflow was reached.

This library is supposed to handle "phantom overflow" by allowing multiplication and division even when the intermediate value overflows 256 bits as documented by UniswapV3.

In the original UniswapV3 code, unchecked is not used as solidity version is < 0.8.0, which does not revert on overflow.

This library is used in many places in the protocol, so it is important that it is properly implemented, or it will lead to unintended results.

Recommendations

Add the entire functions bodies of `getTickAtSqrtRatio()` and `getTickAtSqrtRatio()` to an unchecked block.

Take a look at how Uniswap has implemented it: https://github.com/Uniswap/v3-core/blob/0.8/contracts/libraries/TickMath.sol

[M-04] `_safeTransferToken()` does not check the returned value of tokens

Severity

Impact: Medium

Likelihood: Medium

Description

The `_safeTransferToken` function in the `FarmKeeper.sol` uses the standard `.transfer` method to move ERC-20 tokens:

```
  function _safeTransferToken(address token, address to, uint256 amount)
private {
    uint256 balanace = IERC20(token).balanceOf(address(this));

    // In case if rounding error causes farm keeper to not have enough
tokens.
    if (amount > balanace) {
      IERC20(token).transfer(to, balanace);
    } else {
      IERC20(token).transfer(to, amount);
    }
  }
```

However, it does not check the return value of the `.transfer` function, which can lead to issues if the token contract does not behave as expected.

In the ERC-20 standard, the `transfer` function returns a boolean value indicating success (`true`) or failure (`false`). If this return value is not checked, the function could assume the transfer succeeded when it may have failed, potentially leading to inconsistencies in the contract's state, unclaimed tokens, or loss of funds.

Note: The `transfer()` function is also used in other places, not just in `_safeTransferToken()`. It is also recommended to replace it with `safeTrasnfer()`.

Recommendations

Replace the current `transfer()` with `safeTransfer()` from OpenZeppelin.

[M-05] Some functions don't have deadline check

Severity

Impact: Medium

Likelihood: Medium

Description

The `withdraw()` function calls `_decreaseLiquidity()`, and the `deposit()` function calls `_addLiquidity()` and `_createLiquidityPosition()`. These internal functions set the `deadline` parameter to `block.timestamp`:

```
        INonfungiblePositionManager.DecreaseLiquidityParams({
          tokenId: farm.lp.tokenId,
          liquidity: liquidity,
          amount0Min: minAmount0,
          amount1Min: minAmount1,
          deadline: block.timestamp
        })
```

Without a deadline, pending transactions could be left open and later executed under conditions that are unfavorable to the user.

Users have no way to enforce a time limit on their transactions, meaning their swap could be executed at an unexpected time, which may no longer be beneficial.

Recommendations

Introduce a `deadline` parameter in these functions.

[L-01] The protocol cannot remove the input token when needed

The `InputTokens.sol` library includes a `remove()` function designed to remove an `InputToken` from the map:

```
  function remove(Map storage map, InputToken calldata token) internal
returns (bool) {
    uint256 position = map._positions[token.id];

    if (position != 0) {
      uint256 valueIndex = position - 1;
      uint256 lastIndex = map._tokens.length - 1;

      if (valueIndex != lastIndex) {
        InputToken storage lastElement = map._tokens[lastIndex];
        map._tokens[valueIndex] = lastElement;
        map._positions[lastElement.id] = position;
      }

      map._tokens.pop();
      delete map._positions[token.id];

      return true;
    } else {
      return false;
    }
  }
```

The protocol includes functionality to add input tokens via the `add()` function, as seen in the `enableFarm()` function in `FarmKeeper.sol` and the `enableInputToken()` function in `UniversalBuyAndBurn.sol`. Despite this, there is no corresponding functionality to remove an input token from the system.

The lack of a removal mechanism might lead to operational rigidness, as there is no way to remove tokens that become obsolete, misconfigured, or otherwise need to be replaced.

Recommendations

You can add such functionality that can remove input tokens.

[L-02] Incorrect Slippage limit check

In the `setSlippage()` function, there is a discrepancy between the NatSpec comment and the actual validation logic. The comment states that the slippage value should be between 0% and 15%, but the `_validateSlippage()` function actually allows slippage values up to 25% (2500 basis points).

`setSlippage()`:

```
  /**
   * @notice Sets the slippage percentage for buy and burn minimum
received amount
   * @param id The ID of the farm to update
   * @param slippage The new slippage value (from 0% to 15%)
   */
  function setSlippage(address id, uint256 slippage) external restricted {
    if (!_farms.contains(id)) revert InvalidFarmId();
    _validateSlippage(slippage);
    _farms.get(id).slippage = slippage;
    emit SlippageUpdated(id, slippage);
  }
```

The actual validation logic in `_validateSlippage` is as follows:

```
  function _validateSlippage(uint256 slippage) private pure {
    if (slippage < 1 || slippage > 2500) revert InvalidSlippage();
  }
```

Recommendations

If the slippage is intended to be up to 25%, then we only have a typo in NatSpec.

If the slippage is intended to be up to 15%, you should do like this:

```
  function _validateSlippage(uint256 slippage) private pure {
    if (slippage < 1 || slippage > 1500) revert InvalidSlippage();
  }
```

[L-03] `setIncentiveFee()` does not check if `incentiveFee` is zero

The `setIncentiveFee()` function allows the contract owner or authorized addresses to set the incentive fee percentage for `buyAndBurn` calls for a specific input token.

```
  function setIncentiveFee(address inputTokenAddress, uint256
incentiveFee) external restricted {
    if (!_inputTokens.contains(inputTokenAddress)) revert
InvalidInputTokenAddress();
    _validateIncentiveFee(incentiveFee);
    _inputTokens.get(inputTokenAddress).incentiveFee = incentiveFee;
    emit IncentiveFeeUpdated(inputTokenAddress, incentiveFee);
  }
```

The function's documentation (NatSpec) suggests that the incentive fee should be between 1 basis point (0.01%) and 1000 basis points (10%).

> @param incentiveFee The incentive fee in basis points (1 = 0.01%, 1000 = 10%).

However, the `_validateIncentiveFee` function only checks that the incentive fee does not exceed 1000 basis points, allowing the incentive fee to be set to 0, which contradicts the NatSpec.

```
  function _validateIncentiveFee(uint256 fee) private pure {
    if (fee > 1000) revert InvalidIncentiveFee();
  }
```

Recommendations

Add check if `incentiveFee` is zero.

[L-04] Incomplete validation of TWA period in `setPriceTwa()`

The `setPriceTwa()` function allows the contract owner or authorized addresses to set the Time-Weighted Average (TWA) period for price quotes used in `buyAndBurn` swaps for a specific input token.

```
  function setPriceTwa(address inputTokenAddress, uint32 mins) external
restricted {
    if (!_inputTokens.contains(inputTokenAddress)) revert
InvalidInputTokenAddress();
    _validatePriceTwa(mins);
    _inputTokens.get(inputTokenAddress).priceTwa = mins;
    emit PriceTwaUpdated(inputTokenAddress, mins);
  }
```

According to the NatSpec documentation, the TWA period should be between 5 minutes and 60 minutes, with an option to disable it by setting it to zero.

> Thrown if the TWA period is not between 5 minutes and 60 minutes (1 hour).

However, the `_validatePriceTwa()` function only checks that the TWA period does not exceed 60 minutes, without enforcing the minimum limit of 5 minutes.

```
  function _validatePriceTwa(uint32 mins) private pure {
    if (mins > 60) revert InvalidPriceTwa();
  }
```

This inconsistency could lead to the TWA being set to values below 5 minutes and above zero. Setting the TWA period to less than 5 minutes could lead to less reliable price quotes.

Recommendations

Update the `_validatePriceTwa()` function to enforce a minimum TWA period of 5 minutes, but also add a check to be able to set it to zero.

[L-05] `_collectFees()` should be called before changing `protocolFee`

In the FarmKeeper contract, the admin can update the `protocolFee` for each farm by calling the `setProtocolFee()` function. The new protocol fee will take effect the next time `_collectFees()` is invoked. However, this means that the new protocol fee will be applied to fees that have not yet been accrued up to the moment `setProtocolFee()` is called.

```
function setProtocolFee(address id, uint256 fee) external restricted {
  if (!_farms.contains(id)) revert InvalidFarmId();

  _validateProtocolFee(fee);
  _farms.get(id).protocolFee = fee;
  emit ProtocolFeeUpdated(id, fee);
}
```

Recommendations

Consider calling `_updateFarm(farm, true)` before updating the `protocolFee` value of a farm.

[L-06] Unsafe casting

In the function `_approveForSwap()`, there is an unsafe cast from `uint256` amount to `uint160` without verifying that the value fits within the range.

```
function _approveForSwap(address token, uint256 amount) private {
  // Approve transfer via permit2
  IERC20(token).safeIncreaseAllowance(Constants.PERMIT2, amount);

  // Give universal router access to tokens via permit2
  IPermit2(Constants.PERMIT2).approve(token, Constants.UNIVERSAL_ROUTER,
uint160(amount), uint48(block.timestamp)); // @audit unsafe casting
`amount`
}
```

Recommendations

Consider using the `SafeCast` library or verifying that `amount` fits in `uint160` before casting.