CD SECURITY

AUDIT REPORT

Pegged Farm Keeper
January 2025

Prepared by
GT_GSEC
tsvetanovv
MrPotatoMagic

Introduction

A time-boxed security review of the **Pegged Farm Keeper** 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 **Pegged Farm Keeper**

The PeggedFarmKeeper contract is an advanced addition to the TitanX-Farms protocol, built specifically to manage Uniswap V3 liquidity farms with a focus on stable farms. These farms enable users to provide liquidity within tight price ranges, such as maintaining a 1:1 peg for stablecoin pairs, ensuring efficient liquidity concentration.

Key Features:

- Uniswap V3 Compatibility: Supports precise liquidity management by leveraging Uniswap V3's concentrated liquidity model.
- Flexible Reward Distribution: Allocates incentive tokens to users based on their share of liquidity and time in the farm, using a sub-allocation mechanism from the root FarmKeeper.
- Fee Management: Automatically collects fees from Uniswap V3 positions and:
    - Sends designated tokens to a buy-and-burn mechanism to enhance token value.
    - Distributes non-designated tokens as rewards to liquidity providers.
- Protocol Fee Integration: Allows for customizable protocol fees on farm-generated earnings.
- Sub-Farm Architecture: Acts as a sub-farm keeper under the root FarmKeeper, utilizing proxy pools to manage and distribute rewards seamlessly.

The PeggedFarmKeeper.sol enhances the efficiency of liquidity farming for stable assets, integrates robust fee management systems, and provides added value to the protocol through its buy-and-burn mechanism and reward flexibility.

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* - 5ddbbe2b93fcc402f17cafa1cde2ce5f6acaf756**

Scope

The following smart contracts were in scope of the audit:

- `PeggedFarmKeeper.sol`

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

- Critical & High: 0 issues
- Medium: 2 issues
- Low: 3 issues
- Info: 3 issues

Findings Summary

ID	Title	Severity	Resolution
[M-01]	`farm.lastRewardTime` is incorrectly check-pointed in function `_updateFarm()`	Medium	Fixed
[M-02]	Overestimated accounting of `accIncentiveTokenPerShare` in subsequent farm possible	Medium	Fixed
[L-01]	The slippage is under control of the protocol	Low	Fixed
[L-02]	The pool initialization within `PeggedFarmKeeper`'s constructor can revert	Low	Fixed
[L-03]	The `massUpdateFarms` can be prone to Out of Gas errors	Low	Acknowledged

ID	Title	Severity	Resolution
[I-01]	Improvise redundant `<=< code> check to == in functions enableFarm() and setAllocation()`	Info	Acknowledged
[I-02]	Residual allowance not zeroed for position manager	Info	Fixed
[I-03]	The `enableFarm` allows subsequent farm to have zero allocation	Info	Acknowledged

Detailed Findings

[M-01] `farm.lastRewardTime` is incorrectly check-pointed in function `_updateFarm()`

Severity

Impact: Medium

Likelihood: High

Description

When a farm is enabled, its `lastRewardTime` checkpoint is set based on the latest `_globalAccIncentiveTokenPerShare` at that point in time. But we do not consider that there can be a duration from t_1 to t_2 during which the `_globalAccIncentiveTokenPerShare` increases. Here t_1 refers to the time at which the farm is enabled and t_2 referring to the time at which the first deposit was made to the farm. Due to this, it is possible for the incentiveTokenReward for that particular farm to be higher than intended.

```
    // Pegged farm keeper impl
    if (liquidity == 0 || farm.allocPoints == 0) {
      return;
    }

    // Farm keeper impl
    if (liquidity == 0 || farm.allocPoints == 0) {
      farm.lastRewardTime = block.timestamp;
      return;
    }
```

Recommendations

Make the following update:

```
    // Pegged farm keeper impl
    if (liquidity == 0 || farm.allocPoints == 0) {
      if (liquidity == 0) farm.lastRewardTime =
Math.mulDiv(farm.allocPoints, _globalAccIncentiveTokenPerShare,
Constants.SCALE_FACTOR_1E18);
      return;
    }
```

[M-02] Overestimated accounting of `accIncentiveTokenPerShare` in subsequent farm possible

Severity

Impact: High

Likelihood: Low

Description

Within the PeggedFarmKeeper the `enableFarm` function allows to configure and add new farm to the protocol. The allocation points assertion enforces that first farm must have allocation set, as `totalAllocPoints` is initially zero. However, every subsequent call to the `enableFarm` function allows to add new farm with zero allocation. If such instance occur, then the `lastRewardTime` will be incorrectly set to 0. This impacts every next `accIncentiveTokenPerShare` accounting for this farm, as `lastRewardTime` remains zero. Whenever the allocation points are set and liquidity is provided, the `updateFarm` will account `accIncentiveTokenPerShare` including rewards from the past.

```
  function enableFarm(AddFarmParams calldata params) external restricted {
...
    // Ensure valid allocations points when enabling a farm
    if (totalAllocPoints + params.allocPoints <= 0) revert
InvalidAllocPoints();
...
    lastRewardTime: Math.mulDiv(params.allocPoints,
_globalAccIncentiveTokenPerShare, Constants.SCALE_FACTOR_1E18),
...
```

Recommendations

It is recommended to update the `enableFarm` function, so it requires the allocation to be non-zero in every case.

[L-01] The slippage is under control of the protocol

Severity

Impact: Low

Likelihood: Low

Description

Within the PeggedFarmKeeper protocol each farm can have pre-set slippage as a security control for liquidity operations. However, this pre-set liquidity may be insufficient in volatile pools. An user who would like to overwrite and insert more strict slippage value is devoid of such possibility. This lack could be crucial especially for the `_decreaseLiquidity` operation.

Recommendations

It is recommended to allow users providing custom slippage that overwrites the protocol's slippage whenever a stricter requirement is provided.

[L-02] The pool initialization within `PeggedFarmKeeper`'s constructor can revert

Severity

Impact: Low

Likelihood: Medium

Description

Within the PeggedFarmKeeper constructor the Uniswapv3's pair is being created with the `createAndInitializePoolIfNecessary` function. However, this function requires that tokens provided as input are sorted: `require(token0 < token1)`. Otherwise, the function reverts with default EVM's error. As an impact this vulnerability can results in wasting Gas and overall may delay deployment, as revert investigation can be time consuming.

```
  constructor(
    address incentiveTokenAddress,
    address universalBuyAndBurnAddress,
    address rootKeeperAddress,
    address manager
  ) AccessManaged(manager) {
    incentiveToken = IIncentiveToken(incentiveTokenAddress);
    buyAndBurn = UniversalBuyAndBurn(universalBuyAndBurnAddress);
    rootKeeper = IFarmKeeper(rootKeeperAddress);

    // Deploy two helper tokens, mint supply, deploy a pool
    _proxyTokenA = new ProxyToken();
    _proxyTokenB = new ProxyToken(); //@audit can revert due to
createAndInitializePoolIfNecessary: https://github.com/Uniswap/v3-
```

```
periphery/blob/main/contracts/base/PoolInitializer.sol

    // 1:1 initial price
    rootKeeperFarmId =
INonfungiblePositionManager(Constants.NON_FUNGIBLE_POSITION_MANAGER)
      .createAndInitializePoolIfNecessary(
        address(_proxyTokenA),
        address(_proxyTokenB),
        Constants.FEE_TIER_1_PERCENT,
        792281625142643375935439503 36
      );
  }
```

```
abstract contract PoolInitializer is IPoolInitializer,
PeripheryImmutableState {
    /// @inheritdoc IPoolInitializer
    function createAndInitializePoolIfNecessary(
        address token0,
        address token1,
        uint24 fee,
        uint160 sqrtPriceX96
    ) external payable override returns (address pool) {
        require(token0 < token1);
        pool = IUniswapV3Factory(factory).getPool(token0, token1, fee);
...
```

Recommendations

It is recommended to sort tokens before providing them as an input in the `createAndInitializePoolIfNecessary` function.

[L-03] The `massUpdateFarms` can be prone to Out of Gas errors

Severity

Impact: Medium

Likelihood: Low

Description

Within the PeggedFarmKeeper the `massUpdateFarms` function allows user co manually update all farms at once. Firstly, this process includes rewards collection which calls the `deposit` in the external contract: `Keeper`. Then, it updates each farm within a loop. When the `collectFees` is enabled then Uniswapv3's position manager is called to collect fees. On top of that, whenever any farm's token is `isInputToken0` additional transfer call is made to transfer tokens to the `buyAndBurn` instance. Eventually, the sum of

external calls done within a loop may lead to Gas exhaustion and revert the transaction with the Out of Gas error. However, to trigger such instance multiple farms must be added and enabled in the protocol. The finding is reported as a deviation from leading security practices.

Recommendations

It is recommended to consider enabling user to call the `massUpdateFarms` function with input parameters that defines the range of farms that can be updated within this single call, such as start and end positions.

[I-01] Improvise redundant `<=` check to `==` in functions `enableFarm()` and `setAllocation()`

Description

The checks below ensure that the total allocation points of all farms, including the farm being added or farm being modified for allocation points, is less than equal to 0. But since the sum of two unsigned integers cannot be negative, evaluating it to be less than 0 is redundant.

```
// In function enableFarm()
if (totalAllocPoints + params.allocPoints <= 0) revert
InvalidAllocPoints();

... ...

// In function setAllocation()
if (totalAllocPoints - (farm.allocPoints - allocPoints) <= 0) revert
TotalAllocationCannotBeZero();
```

Recommendations

Consider using `==` instead of `<=`

[I-02] Residual allowance not zeroed for position manager

Description

Within the PeggedFarmKeeper the `_createLiquidityPosition` and `_addLiquidity` functions set allowances for input tokens for Uniswapv3's position manager to mint or increase liquidity. However, there is no guarantee that all allowance is consumed within these processes. Thus, a residual allowance can be left for one or both tokens. Ultimately, the residual allowance can accumulate over time to significant value. Leaving positive value for allowance is considered as a deviation from leading security practices.

Recommendations

It is recommended to set allowances to zero after finishing each `_createLiquidityPosition` and `_addLiquidity` operations.

[I-03] The `enableFarm` allows subsequent farm to have zero allocation

Description

Within the PeggedFarmKeeper the `enableFarm` function allows to configure and add new farm to the protocol. The allocation points assertion enforces that first farm must have allocation set, as `totalAllocPoints` is initially zero. However, every subsequent call to the `enableFarm` function allows to add new farm with zero allocation. This can be updated with the `setAllocation` function, however, it requires additional transaction and Gas consumption.

```
  function enableFarm(AddFarmParams calldata params) external restricted {
...
    // Ensure valid allocations points when enabling a farm
    if (totalAllocPoints + params.allocPoints <= 0) revert
InvalidAllocPoints();
...
```

Recommendations

It is recommended to update the `enableFarm` function, so it requires the allocation to be non-zero in every case.

```
  function enableFarm(AddFarmParams calldata params) external restricted {
...
    // Ensure valid allocations points when enabling a farm
    if (params.allocPoints == 0) revert InvalidAllocPoints();
...
```