dxDAO Carrot KPI Token Review

June, 2021

Chapter 1

Introduction

1.1 Scope of Work

This code review was prepared by Sunfish Technology, LLC at the request of members of dxDAO, an organization governed by a smart contract on the Ethereum blockchain.

This review covers a "KPI token" contract, which is an ERC20 token that is conditionally redeemable for another ERC20 token based on the result of a call to an external "oracle" contract.

Additionally, this review covers a deployment contract unrelated to the KPI token code.

1.2 Source Files

This review covers code from the following public git repositories and commits:

```
https://github.com/Carrot-KPIs/carrot-contracts
5ded5a893c244a2836eac1a55ddb05bce9e03902
```

```
https://github.com/luzzif/dxdao-liquidity-mining-relayer-contracts
079af3e03ddc981e50bdb69d446d6e4e4f7143a7
```

Within those commit ranges, only the following files were reviewed:

- carrot-contracts/contracts/KPIToken.sol
- carrot-contracts/contracts/KPITokensFactory.sol
- dxdao-liquidity-mining-relayer-contracts/contracts/DXdaoLiquidityMiningRelayer.sol

This review was conducted under the optimistic assumption that all of the supporting software infrastructure necessary for the deployment and operation of the reviewed code works as intended. There may be critical defects in code outside of the scope of this review that could render deployed smart contracts inoperable or exploitable.

1.3 License and Disclaimer of Warranty

This source code review is not an endorsement of the code or its suitability for any legal/regulatory regime, and it is not intended as a definitive or exhaustive list of defects. This document is provided expressly for the benefit of dxDAO developers and only under the following terms:

Chapter 2

Moderate Issues

Issues discussed in this section are code defects that may lead to unintended deviations in behavior. It may be possible to chain multiple moderate defects into a working exploit.

2.1 Reentrancy in Finalization

The `KPIToken.finalized` variable that is checked at the beginning of `KPIToken.finalize()` is only set to `true` at the end of the `finalize()` function. Consequently, the calls to `collateralToken.safeTransfer()` on lines 85 and 94 of `KPIToken.sol` may allow the the `finalize()` function to be re-entered. (Some ERC20 tokens implement features that allow the recipient of a token transfer to execute an ERC20 "fallback" function.)

This reentrancy is exploitable by the `creator` address insofar as it allows for `collateralAmount` tokens to be transferred each time the function is re-entered. However, under most circumstances the balance of the `KPIToken` contract will be just `collateralAmount`, in which case the subsequent calls to `finalize()` will revert.

The fix for this issue is to move the `finalize = true;` statement on line 101 to the top of the function just after the `require(!finalized);` statement.

Interestingly, though, this bug allows the `creator` to work around the issue described in section 3.1.

Chapter 3

Minor Issues

Issues discussed in this sections are subjective code defects that affect readability, reliability, or performance.

3.1 Lost Collateral Tokens

If `collateralToken.balanceOf(KPIToken) > collateralAmount`, then all of the additional tokens in the contract will be stuck indefinitely. (The `creator` of the `KPIToken` contract may inadvertently send tokens to the `KPIToken` contract.)

3.2 Useless Constant

The private constant `_10000` in `KPITokensFactory` can just be replaced with the integer contant `10000`. (It wouldn't make sense for the constant `_10000` to be any other value except `1000`, and having the constant declared at the top level of the contract doesn't make the code any clearer.)

3.3 Integer Rounding Error

When the difference between the upper and lower bounds for the KPI range used in `KPIToken.finalize()` is small **and** the number of decimals in `collateralToken` is also small, then the integer truncation that occurs in lines 96 and 97 due to the division operation may cause a non-trivial number of tokens to be stranded in the contract. As an extreme example, if there is only 1 zero-decimal `collateralToken` in the contract and the final KPI progress is more than zero but less than the upper bound (for example, 1 of 3), then that single token will never be distributed because any fractional quantity of that token will round to zero.

Although this bug is unlikely to occur in practice, it may make sense to restrict the `collateralAmount` so that the number of tokens (independent of decimals) is significantly larger than magnitude of the KPI range. For example, `require(collateralAmount`

> `_kpiFullRange * 100)` would guarantee that redemption and finalization never yield more than a one percent round-off error.

3.4 Suspicious Ownership Transfer

`DXdaoLiquidityMiningRelayer.createDistribution()` calls `transferOwnership(msg.sender)` on the newly-created contract. However, the `createDistribution()` function is labelled as `onlyOwner`, so presumably the only possible value of `msg.sender` is `owner`. Consider explicitly calling `transferOwnership(owner)` to make it clear that this is the expected behavior. (Conversely, if this is not the intended behavior, then either the `onlyOwner` modifier should be removed or the argument to `transferOwnership()` should be changed!)