dxDAO Staking Rewards Review No. 3

May, 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 source files implementing a "staking rewards" contract that was originally reviewed by Sunfish Technology, LLC. in March of 2021 and a second time in April of 2021.

1.2 Source Files

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

`github.com/luzzif/erc20-staking-rewards-distribution-contracts`
`50fb10b80159b70042d30a8601fe19a6591543f5`

Within the commits listed above, only the following files were reviewed:

- erc20-staking-rewards-distribution-contracts
  - ERC20StakingRewardsDistribution.sol
  - ERC20StakingRewardsDistributionFactory.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 de-

fects. This document is provided expressly for the benefit of dxDAO developers and only under the following terms:

THIS REVIEW IS PROVIDED BY SUNFISH TECHNOLOGY, LLC. “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL SUNFISH TECHNOLOGY, LLC. OR ITS OWNERS OR EMPLOYEES BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS REPORT OR REVIEWED SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Chapter 2

Minor Issues

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

2.1 Staking Cap DoS

If `stakingCap` is set to a non-zero value, the `stake()` function will refuse to allow `totalStakedTokensAmount` to exceed `stakingCap`. Consequently, an attacker can "grief" other stakers by periodically depositing and then quickly removing a quantity of the staking token such that no other accounts can successfully deposit tokens. The cost of such an attack would be the cost of coordination necessary to "sandwich" the victim's calls to `stake()` with calls to `stake()` and `widthdraw()` by the attacker.

A more directed attack can be carried out by the owner of `ERC20StakingRewardsDistributionFactory`: simply calling `pauseStaking()` will cause all rewards contracts to disallow new deposits. In other words, the factory contract's owner has the ability to hoard the rewards distributed by any of the rewards contracts by unilaterally disabling the `stake()` function after depositing their own funds.

2.2 Arithmetic Loss of Precision

The code on lines 408 through 410 of `ERC20StakingRewardsDistribution.sol` performs two division operations where one could be substituted instead. The code currently reads as:

```
return
  ((_reward.recoverableSeconds * _reward.amount) /
    secondsDuration) / MULTIPLIER;
```

But instead it could read as:

```
return (_reward.recoverableSeconds * _reward.amount) /
       (secondsDuration * MULTIPLIER);
```

This transformation is safe because `secondsDuration` must have no more than 64 significant bits, and so the product `secondsDuration * MULTIPLIER` will never exceed 176 significant bits. (However, consider explicitly zero-extending `secondsDuration` to a 256-bit integer before the multiplication to indicate that the expression is performing 256-bit multiplication rather than 64-bit.)

2.3 Code Duplication

The functions `claim()`, `claimAll()`, and `claimableRewards()` are nearly identical; they could probably be refactored to share logic. For example, `claimAll()` could be implemented in terms of `claim()` and `claimableRewards()`. Similarly, `recoverUnassignedRewards()` could be implemented in terms of `recoverableUnassignedReward()`.

2.4 Inconsistent Unassigned Reward Calculation

The return value of `recoverableUnassignedReward()` and the actual quantity of tokens transferred by `recoverUnassignedRewards()` are not identical, even when each of these functions is called as part of a single transaction. The `recoverUnassignedRewards()` function distributes additional tokens sent to the rewards contract in excess of the explicit reward amount, but the `recoverableUnassignedRewards()` function does not take those tokens into account when calculating its return value.

Chapter 3

Economic Issues

3.1 Locked Funds are Still Fungible

Anyone who possesses the "staking" token to be deposited into the `ERC20StakingRewardsDistribution` contract can deposit tokens without bearing the cost-of-carry risk associated with holding the staking token assets to term. The term "staking" implies that there is something "at stake" for the accounts that deposit funds into the contract, but the only risk that they would be exposed to *in principle* is the cost-of-carry of deposited funds. In practice, those funds can be exchanged without ever withdrawing them from the contract.

Below is a proof-of-concept ERC20 token that allows anyone to deposit staking tokens into the `ERC20StakingRewardsDistribution` contract in exchange for another ERC20 token that is one-to-one redeemable for the staking token plus any accrued "rewards."

```
contract StakingERC20Wrapper is ERC20Burnable {
    IRewards rewards; // rewards contract
    IERC20 token;     // staking token
    constructor(r address, t address) {
        rewards = IRewards(r);
        token = IERC20(t);
    }

    // deposit 'c' staking tokens and receive
    // the same quantity of this token in return
    function deposit(c uint256) external {
        token.safeTransferFrom(msg.sender, address(this), c);
        token.approve(address(rewards), c);
        rewards.deposit(c);
        _mint(msg.sender, c);
    }
```

```
    // withdraw 'c' staking tokens and associated rewards
    // by burning the equivalent amount of this token
    function withdraw(c uint256) external {
        // compute the pro-rata share of rewards
        uint256[] memory amounts = rewards.claimableRewards(address(this));
        for uint i = 0; i < amounts.length; i++ {
            amounts[i] *= c;
            amounts[i] /= totalSupply;
        }
        _burn(msg.sender, c);
        rewards.withdraw(c);
        token.safeTransfer(msg.sender, c)
        rewards.claim(amounts, msg.sender);
    }
}
```

In practice we would expect a swap market for the `StakingERC20Wrapper` token to price it similarly to the underlying staking token, since they are exactly one-to-one redeemable. (Consider: in a single transaction, you could swap for the `StakingERC20Wrapper` token, call `withdraw()` to get an equivalent quantity of the original staking token, and then subsequently swap that token for something else.) In short: initializing the `ERC20StakingRewardsDistribution` contract with `_locked = true` does not meaningfully "lock" any funds, and initializing it with `_locked = false` allows users to trade using deposited funds directly (through swaps). (The fact that a proxy contract can coordinate the transfer of "locked" assets is almost immaterial; even if such a contract were impossible to implement, it would still be possible for people to coordinate to exchange "locked" funds by exchanging keys. A proxy contract simply removes a layer of social coordination.)

More abstractly, this issue is a restatement of the "Missing Economic Rationale" discussion from the first audit of this code in March of 2021. The most common use case for these sorts of "staking rewards" contracts involves paying people to deposit assets in a Uniswap-style AMM contract, but an economically equivalent effect can be achieved by using the rewards funds to buy the liquidity and deposit it into the AMM directly. (There are other behavioral reasons why someone would want to arrange to pay other people to buy a particular crypto-asset, but all of them hinge on information asymmetry between the buying and selling parties. Otherwise, the total value of the circulating supply of the "staking" token deposited into the contract should appreciate by exactly the present value of the income stream pledged as rewards. But, that means the value of the rewards could be converted into an equivalent quantity of AMM liquidity, so the only reason to pay other people to provide that liquidity is if you believe they will mis-price the present value of those future cash flows.)