Protocol Audit Report

Version 1.0

Maroutis

October 6, 2024

Staking and Reward system Audit Report

Maroutis

October, 2024

Staking and Reward system Audit Report

Prepared by: Maroutis

Table of Contents

  - [M-2] Insufficient gas forwarded in Ether transfers can cause funds to be stuck if user is a contract
- Low
  - [L-1] Use `safeTransfer` instead of `transfer` in `RewardSystem`
- Low
  - [L-2] Disabling ALI's `FEATURE_UNSAFE_TRANSFERS` feature can cause double counting of deposits
- Informational
  - [I-1] Consider only allowing whitelisted tokens to be deposited
- Gas
  - [G-1] **`public`** functions not used internally could be marked `external`

Disclaimer

Maroutis makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

Audit Details

The findings described in this document correspond to the deployed ethereum contracts at the following addresses:

```
RewardSystem : 0x196E9dA436535E352F485c8710f22678816b6f42
ERC1363StakingTrackerV1 : 0xEc3DEE982242359a5c6708c295fFf6B430ED8A91
```

Scope

```
src/
- RewardSystem.sol
- ERC1363StakingTrackerV1.sol

src/utils/
- InitializableAccessControl.sol
- Transfers.sol
- UpgradeableAccessControl.sol
```

Protocol Summary

The project includes two main contracts:

- **RewardSystem:** Distributes rewards to users using a Merkle tree for efficient verification. Supports both ETH and ERC20 token rewards. Only authorized users can update the Merkle root.
- **Staking Contract:** Allows users to lock (stake) ERC20 tokens for a specified period without rewards and withdraw their tokens.

Roles

- **Authorized Users:** Only they can perform sensitive operations like updating the Merkle root or managing deposit settings in the contracts.

Executive Summary

Issues found

Severity	Number of issues found
High	0
Medium	2
Low	2
Info	1
Gas Optimizations	1
Total	6

Findings

Medium

[M-1] Lock-up period unfairly extended on additional deposits in `ERC1363StakingTrackerV1` Contract

Description:

In the `ERC1363StakingTrackerV1` contract, when a user makes an additional deposit of tokens, the entire amount, including the previously deposited tokens, has its lock-up period reset:

```
        // otherwise, if it already exists
        else {
            // update it
            account.lastUpdatedOn = now32();
@>          account.maturesOn = now32() + depositDuration; //@audit the period should be a weighted average
            account.amountLocked += depositAmount;
        }
```

This means that users who add more tokens to their stake will have their entire stake's maturity date extended, penalizing the tokens that were already locked. This behavior is unfair to users, as their earlier deposits are effectively locked for longer than initially agreed.

A fair approach would be to calculate a weighted average of the maturity dates based on the amounts and durations, ensuring that each deposit's lock-up period is respected.

Impact:

- Users are discouraged from adding more tokens to their stake, as it would unfairly extend the lock-up period of their existing stake.
- This will lead to less revenue for the protocol

Proof of Concept:

- A user deposits 100 tokens with a lock-up period of 30 days.
- After 15 days, they deposit an additional 50 tokens.
- According to the current implementation, the entire 150 tokens will have a new lock-up period of 30 days from the second deposit.
- Meaning the initial 100 tokens are locked for a total of 45 days, which is unfair.

Recommended Mitigation:

```
else {
    // update it
    account.lastUpdatedOn = now32();
-   account.maturesOn = now32() + depositDuration;
+   uint256 totalAmount = account.amountLocked + depositAmount;
+   uint32 currentTime = now32();
+   uint32 previousMaturity = account.maturesOn > currentTime ? account
    .maturesOn : currentTime;
+   uint256 weightedMaturity = uint256(account.amountLocked) * uint256(
    previousMaturity) + uint256(depositAmount) * uint256(currentTime +
    depositDuration);
+   account.maturesOn = uint32(weightedMaturity / totalAmount);
    account.amountLocked += depositAmount;
}
```

[M-2] Insufficient gas forwarded in Ether transfers can cause funds to be stuck if user is a contract

Description:

In the `RewardSystem` contract, when distributing rewards in native Ether, the `Transfers` library is used to transfer eth to the recipient:

```
    function claimReward(address payable _to, uint256 _totalReward,
        bytes32[] memory _proof) external {
        // ...
        if (isEthRewardSystem()) {
            // transfer ether to user
@>          Transfers.transfer(_to, claimableAmount);

            // emit an event
            emit EthRewardClaimed(_to, claimableAmount);
        }
```

The `Transfers.transfer` function in the `Transfers` library forwards a **fixed amount of gas (4900)** when making the Ether transfer:

```
function send(address payable to, uint256 value) internal returns(bool)
     {
@>   (bool success, ) = to.call{gas: 4900, value: value}("");
     return success;
}
```

This amount of gas may not be sufficient for certain recipient contracts, such as multisig wallets to execute any non-trivial logic in a receive() or fallback function. For instance, it is not enough for Gnosis Safes Wallets (such as this one) to receive funds, which require > 6k gas for the call to reach the implementation contract and emit an event.

As a result, transfers to such contracts will fail, and funds will be stuck in the `RewardSystem` contract.

Impact:

As we know, today many users prefer using smart contracts as wallet especially multisig wallets.

- Users who use smart contract wallets or multisig wallets like Safe will not be able to receive their eth rewards.

Recommended Mitigation:

There is no risk of reentrancy attacks in the function `claimReward` as it follows CEI. You can forward all remaining gas when sending eth. If you want to avoid all risk, it's better to add a reentrancy guard rather than limiting the amount of gas.

Low

[L-1] Use `safeTransfer` instead of `transfer` in `RewardSystem`

Description:

In the `RewardSystem` contract, the function `claimReward` is used to distribute rewards to users. When the reward system is configured to distribute `ERC20` tokens, the function uses the standard `ERC20` transfer method to send tokens to users:

```
// transfer erc20 reward token to user
erc20RewardToken.transfer(_to, claimableAmount);
```

However, not all ERC20 tokens adhere strictly to the ERC20 standard. Some tokens do not return a boolean value or may revert in unexpected ways. Using the standard `transfer` function without checking the return value or handling exceptions can lead to tokens being lost or the function failing silently.

Impact:

- Users may not receive their rewards in case of silent failure, leading to loss of funds for users who cannot claim again.

Recommended Mitigation:

Use OpenZeppelin's `SafeERC20` library and replace `transfer` with `safeTransfer`.

Low

[L-2] Disabling ALI's FEATURE_UNSAFE_TRANSFERS feature can cause double counting of deposits

Description:

The staking contract `ERC1363StakingTrackerV1` allows users to deposit `ERC20` tokens, specifically the Alethea AI token, which is an `ERC1363-compliant` token. The default depositDuration is set to `2592000` seconds (approximately `30 days`).

The ALI token has a feature flag `FEATURE_UNSAFE_TRANSFERS` that controls how transfers behave:

- When `FEATURE_UNSAFE_TRANSFERS` is enabled, standard `ERC20 transfer and transferFrom` functions behave normally.
- When `FEATURE_UNSAFE_TRANSFERS` is disabled, `transfer` and `transferFrom` functions trigger the `onTransferReceived` callback on the recipient, similar to `transferAndCall` and `transferFromAndCall`.

```
    function transferFrom(address _from, address _to, uint256 _value)
        public override returns (bool success) {
         // depending on `FEATURE_UNSAFE_TRANSFERS` we execute either
             safe (default)
         // or unsafe transfer
         // if `FEATURE_UNSAFE_TRANSFERS` is enabled
         // or receiver has `ROLE_ERC20_RECEIVER` permission
         // or sender has `ROLE_ERC20_SENDER` permission
         if(isFeatureEnabled(FEATURE_UNSAFE_TRANSFERS)
             || isOperatorInRole(_to, ROLE_ERC20_RECEIVER)
```

```
            || isSenderInRole(ROLE_ERC20_SENDER)) {
            // we execute unsafe transfer - delegate call to `
                unsafeTransferFrom`,
            // `FEATURE_TRANSFERS` is verified inside it
            unsafeTransferFrom(_from, _to, _value);
        }
        // otherwise - if `FEATURE_UNSAFE_TRANSFERS` is disabled
        // and receiver doesn't have `ROLE_ERC20_RECEIVER` permission
        else {
            // we execute safe transfer - delegate call to `
                safeTransferFrom`, passing empty `_data`,
            // `FEATURE_TRANSFERS` is verified inside it
            safeTransferFrom(_from, _to, _value, "");
        }
```

If `FEATURE_UNSAFE_TRANSFERS` is disabled, calling the `deposit()` function in the staking contract can lead to an unintended behavior where the `__createDeposit` function is called twice for a single token transfer:

1. The `deposit()` function calls `transferFrom(msg.sender, address(this), depositAmount);`.
2. Since `FEATURE_UNSAFE_TRANSFERS` is disabled, this `transferFrom` triggers the `onTransferReceived` callback on the staking contract.
3. The staking contract's `onTransferReceived` function calls `__createDeposit`.
4. After the `transferFrom`, the `deposit()` function also calls `__createDeposit`.

Impact:

- Users can manipulate the contract to record twice the actual amount of tokens deposited, leading to incorrect accounting.
- Malicious users could exploit this behavior to withdraw more tokens than they deposited once the lock-up period expires.
- Loss of funds for the protocol and legitimate users.

Recommended Mitigation:

There are many ways to prevent a user from calling `__createDeposit` twice in the same function :

- Add a state variable to track when a deposit is in progress and prevent `__createDeposit` from processing deposits triggered internally. Example :

```
    function deposit(address depositToken, uint160 depositAmount)
        public {
        require(!_depositInProgress, "Reentrant call detected");

        _depositInProgress = true;

        // transfer the tokens into the contract for locking
        IERC20(depositToken).safeTransferFrom(msg.sender, address(this)
            , depositAmount);

        // delegate to `__createDeposit`
        __createDeposit(depositToken, msg.sender, depositAmount);

        _depositInProgress = false;
    }

    function onTransferReceived(address, address from, uint256 value,
        bytes memory data) external returns (bytes4) {
        if (_depositInProgress) {
            // Ignore the callback if we're already processing a
                deposit
            return this.onTransferReceived.selector;
        }
```

- You can also consider a reentrancy guard on `__createDeposit`.

Informational

[I-1] Consider only allowing whitelisted tokens to be deposited

Description:

As of now, any token can be deposited and withdrawn. However, it is clear that the rewards will only be determined on the whitelisted deposited tokens not on every ERC20. This can be confusing for users and can lead to useless deposits.

Recommended Mitigation:

Only allow whitelisted tokens with a valid `depositDuration` to be deposited.

Gas

[G-1] `public` functions not used internally could be marked `external`

Description:

Instead of marking a function as `public`, consider marking it as `external` if it is not used internally.

Instances :

- Found in src/contracts/ERC1363StakingTrackerV1.sol [Line: 2]

```
function postConstruct(address tokenAddress, uint32
    depositDuration) public initializer {
```

- Found in src/contracts/ERC1363StakingTrackerV1.sol [Line: 175]

```
function getUserAccount(address depositToken, address
    accountAddress) public view returns(UserAccount memory
    account) {
```

- Found in src/contracts/ERC1363StakingTrackerV1.sol [Line: 191]

```
function deposit(address depositToken, uint160 depositAmount)
    public {
```

- Found in src/contracts/ERC1363StakingTrackerV1.sol [Line: 381]

```
function deleteDepositSettings(address depositToken) public {
```

- Found in src/contracts/ERC1363StakingTrackerV1.sol [Line: 405]

```
function updateDepositSettings(address depositToken, uint32
    depositDuration) public {
```

- Found in src/contracts/RewardSystem.sol [Line: 106]

```
  function postConstruct(address _erc20RewardToken) public
      virtual initializer {
```

- Found in src/contracts/RewardSystem.sol [Line: 173]

```
  function resetClaimedRewards() public {
```