



ICO Audit

Audited Material Summary

The audit consists of the following contracts:

### <https://github.com/leverj/staking/tree/e8716e4a11881fad181b5330206d8b0c27a58510/contracts>

* Stake.sol
* Fee.sol
* HumanStandardToken.sol
* StandardToken.sol
* Token.sol
* Owned.sol
* Validating.sol
* SafeMath.sol

# 

## Summary

The contracts under audit implement an ERC20 token based ICO (LEV) with an special staking contract that allows the user to lock (stake) LEV tokens and earn FEE tokens. LEV tokens staked are released and Fee tokens dispersed at the culmination of a staking period.

## Security

The contracts have no critical security issues however we have identified many issues ranging from stylistic conventions to potentially confusing contradictions or omissions between the white paper and the contract code.

An owner of the Stake contract appears to have an extraordinarily high level of control. We are unsure if this is design choice. It is not generally considered to be good practice and could lead to unexpected behaviour due to multiple ownership.

There is also a possibility to prevent staking for one staking period if the end block is lesser than start block. For that period stakers will become unable to stake.

Token.sol, StandardToken.sol, HumanStandardToken.sol

All of these contracts are simple instantiations of ERC20 base standard except in Token.sol:

function totalSupply() constant returns (uint256 supply);

is replaced with:

uint256 public totalSupply;

## Security

No issues in these contracts.

SafeMath.sol

Zeppelin’s widely implemented library. Used to prevent uint overflow.

It is used/called by *Owned*, *Fee* and *Stake*.

## Security

No issues in this contract.

Fee.sol

Fee inherits from *Owned*, *Validating* and *StandardToken*.

The primary function in Fee is *sendTokens()* which generates FEE tokens and transfers them to the staker. A new minter can be set by the owner using *setMinter()* and the ability to eliminate tokens is provided by *burnTokens()*.

## Functional Breakdown

### setMinter

Only Owner can call this function. It requires valid address to set as a new minter.

### burnTokens

It requires caller’s balance in FEE greater than amount to burn and amount to burn greater than zero.

The sender’s balance is decreasing by the specified amount, the *feeInCirculation* is also decreasing accordingly. Then event *Burn()* fired.

### sendTokens

The Minter can create specified amount of FEE tokens and send it to the specified address. It requires: *validAddress* and *notZero* amount.

The balance of recipient is increasing to the specified amount, the *feeInCirculation* is also increasing accordingly. Then event *Transfer()* fired.

## Security

No issues in this contract except confusing names of events.

Stake.sol

The Stake contracts allow users to lock their LEV tokens for a while and get a FEE tokes as a benefit of that. Staking tokens require running staking period, which could be initiated by the Operator, by calling *startNewStakingPeriod()* function. All funds sent to the Stake contract are still locked until the end of staking period. The lockage time and the amount of locked LEV tokens define the FEE amount paid to the user.

After staking period is over, users can redeem their funds and FEE tokens they earned. The redeeming to the stakers could be also initiated by the Operator.

The FEE paid to the stakers could be increased by sending extra FEE tokens to the Stake contract.

## Functional Breakdown

### Stake

Constructor, deploy the contract *Stake.sol* and assign owner(s), operator, Fee token price. It also connects LEV contract to the *Stake*.

### version

Always return “1.0.0”. Probably because the project is upgradable and different contracts return the version from the same function. (shall we guess?)

### setLevToken

To set the the address of the LEV token. Could be changed by the owner.

### setFeeToken

To set the the address of the FEE token. Could be changed by the owner.

### setWallet

To set the the address of the Wallet. Could be changed by the owner.

### stakeToken

Public function to stake tokens executable by any user. It requires:

### Staking period is not over;

### Amount of LEV tokens on the balance is greater or equal than amount to stake

### User approved amount of tokens to transfer from his account to the *Stake* contract.

By staking tokens, the amount of *LevBlocks* is calculating: number of LEV tokens from user multiplied by number of blocks in the current staking period + amount of *LevBlocks* from the stakes made earlier.

When staking is done, event *StakeEvent()* is triggered.

### revertFeeCalculatedFlag

It changes the flag to the true or false (which is defined by parameter).

### updateFeeForCurrentStakingInterval

It requires that staking interval is over. Function check if there are any FEE tokens on the balance of the stake contract and then calculate the amount of the FEE price for the current staking interval: FEE price from the previous interval + (FEE tokens on the balance divided to the *WeiPerFee* variable). After that function burns any tokens on the balance of the Stake contract and remove that amount from the tracked FEE in circulation.

### redeemLevAndFeeByStaker

The public gate to the *redemLevAndFee()* function.

### redeemLevAndFeeToStakers

The only operator could call this function which will loop trough all stakers sent in the parameter and call *redeemLevAndFee()* function for each of them.

### redeemLevAndFee

Requires: valid address of staker, FEE price is calculated, the amount of locked LEV tokens is more than 0, stakes of the staker is greater than zero.

Function calculates the amount of FEE to send to the staker: *feeEarned* = the amount of LEVs locked is multiplied by *feeForStakingInterval* (calculated earlier) and divided by total LEV tokens staked by all stakers. This amount is subtracting from the *totalLevs*, and deleting from the balance of the contract. If it greater than a zero, FEE contract’s *sendTokens()* is called to send *feeEarned* to the staker.

### startNewStakingInterval

Operator can start new staking interval by calling this function with start and end blocks defined. It requires that start and and blocks are non-zero. It also requires that *isDoneStaking* is true, which means that there is no staking period in progress now.

NB: There is no check if the end block is greater than the start one.

## Security

An owner of the Stake contract appears to have an extraordinarily high level of control. We are unsure if this is design choice. It is not generally considered to be good practice and could lead to unexpected behaviour due to multiple ownership.

There is also a possibility to start new staking period if the end block is lesser than start block. That will make the staking process impossible during this staking period because modifier *isStaking()* requires that *endBlock* should be greater than *startBlock*.

After *endBlock* is in the past, the new staking period is possible.