The eurxb.finance liquidity event Smart Contract audit report

By: SFXDX Team
For: EURxb Finance

www.sfxdx.com

EURxb.finance liquidity event: Smart Contracts audit report

The following Audit was prepared by the Sfxdx team and the purpose of this audit is to analyze the codebase of ten smart contracts for quality of the code, security threats and compliance with the business requirements.

Authors:
Maxim Prishchepo,
Stanislav Golovin.

This document describes the compliance of EURxb.finance platform`s Smart Contracts with the logical, architectural and security requirements. The following Audit was prepared by the Sfxdx and the purpose of this audit is to analyze the codebase of ten smart contracts for quality of the code, security threats and compliance with the business requirements.

During the audit, we classified the shortcomings into four categories of severit

- Informational - shortcomings that are described in the document for informational purposes only. They are not required for correction, do not violate the intended architecture and business logic.

- Low - shortcomings that do not violate the requirements, but can be eliminated to improve the quality of the code. Their elimination can be considered as a recommendation.

- Medium - shortcomings that are not serious, but should be better eliminated. Their elimination can be considered as a recommendation.

- High - shortcomings that lead to the fact that the smart contract does not meet the requirements. For example, business logic violation, inconsistency with the planned architecture etc. This type of shortcomings must be eliminated from the smart contract code.

Based on the results of the audit, the following conclusion can be made:

Generally the implemented business logic and architecture of smart contracts meets initial requirements. All smart contracts do not have security or other critical issues.
Mostly informational - severity shortcomings were found. For example, lack or extra checks.
No medium or high - severity issues were found.
Also few low-severity shortcomings were detected and eliminating them can improve the code' quality.

A more detailed description of the founded issues is provided in the document. The document attached.

Introduction

Git repository with contracts:

https://github.com/EURxbfinance/LPE

Audited contracts:

Router
Incentivizer

Below are summary conclusions regarding the contracts codebase considering the business logic analysis, correctness of logic implementation, security analysis, gas efficiency analysis and style guide principles.

.

Router

30.03.2021

https://github.com/EURxbfinance/LPE/commit/752bcc780b70c4d1dc51cb496e1caa9621b9c24d

The “Router” contract audit results includes the defined shortcomings of the following categories:

Severity		Count
Informational	-	1
Low	-	3
Medium	-	0
High	-	0

The shortcomings are detailed below.

Inheriting from Initializable contracts, without using the logic of upgradeable contracts or proxy contracts

Severity - Low

The Router contract is a derived from the Initializable contract of the OpenZeppelin library https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/proxy/utils/Initializable.so,which is primarily designed to bring constructor logic to the initialization function. In the OpenZeppelin library, this contract is part of the concept of proxy and upgradable contracts. In this implementation, however, this contract is just for delayed initialization, which can be misleading.
Deferred initialization gives us some flexibility, allowing us to initialize the contract later than when it is deployed. However, in this case, there is a risk that a contract that has not yet been initialized will be mishandled.

Initialization of the contract

The primary initialization of the "Router" contract occurs in three functions: "constructor", "configure" and the "setStartWeights" function. Only after all three functions have been executed the contract can be considered started, configured and ready for use.

There is no check for null values of arguments in the configure function

Severity - Informational

The configure function does not check that the passed address arguments are not null.
Possible implementation of the check:

```
require(_uniswapLikeRouter != address(0), "uniswap router address is invalid");
    require(_eurxb != address(0), "eurxb address is invalid");
    require(_token != address(0), "token address is invalid");
    require(_teamAddress != address(0), "team address is invalid");
```

There is no check for null values of arguments in the setStartWeights function

Severity - Low

The setStartWeights function does not check that the passed address arguments are not null.
Possible implementation of the check:

```
require(tokenWeight != 0, "zero token weight");
require(eurxbWeight != 0, "zero eurxb weight");
```

The "setStartWeights" function can only be executed by the Owner of the contract, which minimizes the risk of entering incorrect data, but if an incorrect (zero) value is entered as one of the parameters, the contract will be disrupted.

No check for completion of contract initialization before calling writable functions

Severity - Low

Since the "configure" function is responsible for initializing the "Router" contract, all writable contract functions must be executed after it is executed. However, all functions do not check if the contract initialization is complete.

Privileged operations

The following requirements are met in the Router contract:

- only the contract owner can set the initial ratio of weights in the uniswap pair, if it has not already been created;
- only the contract owner may close the contract by terminating the addLiquidity function and transferring the balance of the eurxb token to his address;
- only the contract owner can reopen the contract, returning the addLiquidity function to work;

To implement the above requirements, the "Router" contract inherits from the "Ownable" contract template from OpenZeppelin (https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/access/Ownable.sol), setting the contract deployer address by the contract administrator (in the private variable "_owner") when the contract is deployed.
The external functions setStartWeights, closeContract, reOpenContract use the modifier "onlyOwner" to check the caller against the address of the set contract owner.

Here it is important to note that the current owner of the contract can transfer the privileged right to the new owner, if necessary - this logic is defined in the "Ownable" contract template from OpenZeppelin.

No bugs were found in the implementation of this functionality.

Functionality specifics: the "addLiquidity" function

The "addLiquidity" function allows any user to execute an operation of adding liquidity to the pool of the uniswap protocol predefined in the "configure" function.

In this case, if there is a sufficient amount of EURxb tokens on the balance of the "Router" contract, then:
- first, half of the amount of contributed tokens will be exchanged for EURxb tokens at the current rate of the required pool / pair;
- Then liquidity in the form of both tokens will be added to the required pool / pair in appropriate proportions. The pool's LP tokens for the contributed liquidity will be sent to the user.
- The rest of the contributed token will be sent to the team address.

No bugs were found in the implementation of this functionality.

Variables and data structures used

The following variables and data structures are used in the "Router" contract:

```
address public eurxb;
        address public token;
        IUniswapV2Router02 public uniswapLikeRouter;
        address public uniswapLikeFactory;

        address public teamAddress;

        uint256 public startWeightToken;
        uint256 public startWeightEurxb;

        bool public isClosedContract;
```

All structures are described as public variables, so they do not require additional getters functions.

Incentivizer

30.03.2021

https://github.com/EURxbfinance/LPE/commit/752bcc780b70c4d1dc51cb496e1caa9621b9c24d

The “Incentivizer” contract audit results includes the defined shortcomings of the following categories:

Severity		Count
Informational	-	2
Low	-	3
Medium	-	0
High	-	0

The shortcomings are detailed below.

Inheriting from Initializable contracts, without using the logic of upgradeable contracts or proxy contracts

Severity - Low

The Incentivizer contract inherits from the "Initializable" contract of the OpenZeppelin library https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/proxy/utils/Initializable.sol, which is primarily designed to bring constructor logic to the initialization function. In the OpenZeppelin library, this contract is part of the concept of proxy and upgradable contracts. In this implementation, however, this contract is just for delayed initialization, which can be misleading.
The delayed initialization gives some flexibility, allowing to initialize the contract at a later time instead of the time of deployment. However, in this case, there is a risk that a contract that has not yet been initialized will be used inappropriately.

There is no interface that supports all contract functions

Severity - Informational

The contract is almost a full copy of the "StakingReward" contract, with which the Uniswap protocol distributed UNI tokens. The same principle of the staking-contract is used, for example, in yearn governance and other contracts of known protocols. All the public functions of the main contract are described in the "IStakingRewards" (except for notifyRewardAmount which is described in the "RewardsDistributionRecipient" contract), the same interface is supported by the "Incentivizer" contract. However, there is a new function added to "Incentivizer": "setRewardsDuration", which has not been added to any interface from which "Incentivizer" contract is inherited. Thus, access to this function may be complicated for other contracts that want to interact with this contract.

Initialization of the contract

The primary initialization of the "Incentivizer" contract is performed in the "configure" function. Only after executing this function the contract can be considered launched, configured and ready for use.

There is no check for null values of arguments in the configure function

Severity - Informational

The configure function does not check that the passed address arguments are not null. Possible implementation of the check:

```
require(_rewardsDistribution != address(0), "rewards distribution address is invalid");
          require(_rewardsToken != address(0), "rewards token address is invalid");
          require(_stakingToken != address(0), "staking token address is invalid");
          require(_rewardsDuration !=0, "incorrect reward duration");
```

No check for completion of contract initialization before calling writable functions

Severity - Low

Since the "configure" function is responsible for initializing the "Incentivizer" contract, all writable contract functions must be executed after it is executed. However, all functions do not check if the contract initialization is complete.

Impossible to change the rewardsDistribution address

Severity - Low

After deploying and configuring the Incentivizer contract, it is not possible to change the address of the user (or contract) providing the reward, which will then be distributed among the stakers. In the basic contract, which implements the same logic, this is not a problem, because stackers can withdraw funds whenever they want. In this implementation, however, all user funds are locked for as long as the reward is being distributed. If the private key of the "rewardsDistribution" address will be compromised, there is a possibility of an exploit, when from that address will be added a small amount of reward (eg 1wei), and this will time after time block the users' funds on the contract. Since they can withdraw funds only after the "rewardsDuration" seconds after the last addition of rewards.

Privileged operations

The Incentivizer contract meets the following requirements:
- only the Owner of the contract can set the reward period, and only if the current reward period has already expired;

To implement the above requirements, the "Incentivizer" contract inherits from the "Ownable" contract template from OpenZeppelin (https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/access/Ownable.sol), setting the contract deployer address by the contract administrator (in the private variable "_owner") when the contract is deployed.
The external function setRewardsDuration uses the modifier "onlyOwner" to check the caller against the address of the set contract owner.

Here it is important to note that the current contract owner can transfer the privileged right to the new owner, if necessary - this logic is defined in the contract template "Ownable" from OpenZeppelin.

No bugs were found in the implementation of this functionality.

Variables and data structures used

The following variables and data structures are used in the "Incentivizer" contract:

```
IERC20 public rewardsToken;
        IERC20 public stakingToken;
        uint256 public periodFinish;
        uint256 public rewardRate;
        uint256 public rewardsDuration;
        uint256 public lastUpdateTime;
        uint256 public rewardPerTokenStored;

        mapping(address => uint256) public userRewardPerTokenPaid;
        mapping(address => uint256) public rewards;
        uint256 private _totalSupply;
        mapping(address => uint256) private _balances;
```

All structures are described as public variables, so they do not require additional getters functions.