SFXDX

# The eurxb.finance: Governance Smart Contract audit report

By: SFXDX Team
For: EURxb Finance

www.sfxdx.com

# EURxb.finance: Governance: 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  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/vaults

Audited contracts:

**Governance.sol**
**Governable.sol**
**LPTokenWrapper.sol**
**RewardDistributionRecipient.sol**

Below are summary conclusions regarding the contract codebase considering the business logic analysis, correctness of logic implementation, security analysis, gas efficiency analysis and style guide principles.
As an appendix to this report the “Test Coverage” report and “Static Analysis” report are also attached.

# Governance

05.04.2021

https://github.com/EURxbfinance/vaults/commit/5c1409aacf7a046f504bfa68dc936c70fa0c5c60

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

| Severity | | Count |
|---|---|---|
| Informational | - | 5 |
| Low | - | 4 |
| Medium | - | 0 |
| High | - | 0 |

The shortcomings are detailed below.

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

Low

The "Governance" contract is inherited 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 into 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.

Delayed initialization gives some flexibility in use, allowing you to initialize the contract at a later time, rather than at the time of deployment. However, in this case, there is a risk that a contract that has not yet been initialized will be misused.

## Inheritance from Ownable and Governable contracts simultaneously

Informational

The "Governance" contract clearly inherits from the "Governable" contract and indirectly from the "Ownable" contract ("Governance" inherits from the abstract IRewardDistributionRecipient contract, which in turn inherits from "Ownable"). Both of these contracts serve the same logic - unique rights to use certain smart contract features. However, the "Ownable" contract is only used to call the setRewardDistribution function.

Recommendation:

Drop on one of the two contracts by combining all functions available only to the contract administrator with one modifier (either onlyOwner or onlyGovernance)

## Incomplete GSN logic support

Low

The underlying contract "Ownable", of the "Governable" contract, is inherited from the Context contract, this is done to support gas stations (so that operations can be done by paying for gas with tokens). However, the other underlying contracts ("LPTokenWrapper" and "Governable"), from which "Governance" inherits, do not support gas stations, so that calling some functions cannot be paid in tokens.

Recommendation:

Either drop support for GSN completely, this would make the functions a little cheaper to use. Or make GSN support complete - that is, inherit "LPTokenWrapper" and "Governable" contracts from Context.

## Variables and data structures

### Proposal's non-optimal structure

Informational

In the "Governance" contract, the "Proposal" structure is declared. Variables in this structure are not ordered by data type, so they cannot be efficiently optimized by the compiler. This does not affect the logic of the contracts, but using the contract functions associated with the "Proposal" structure is slightly more expensive in terms of gas.

Recommendation:

Change the sequence of data declaration in the structure by grouping variables with the same data type together.

### The DURATION constant is invariable

Informational

The DURATION public constant is responsible for the distribution period of the reward among all the governance token stakers. This constant is equal to 7 days (in seconds) and cannot be changed in the future. It is the only parameter of all that cannot be changed in the contract. It is not clear for what purpose this constant was made immutable.

## Initializing a contract

The initial initialization of the "Governance" contract takes place in the "configure" function. Only after this function has been executed can the contract be considered running, configured and ready for use.

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

Informational

The configure function does not check that passed arguments are non-null.
A possible implementation of the check:

```
require(_rewardsTokenAddress != address(0), "rewards token address is invalid");
require(_governanceToken != address(0), "governance token address is invalid");
require(_governance != address(0), "governance address is invalid");
require(_rewardDistribution != address(0), "reward distribution address is invalid");
```

### No check for completeness of contract initialization before calling writable functions

Low

Since the "configure" function is responsible for initializing the "Governance" contract, all writable contract functions must be executed after it is executed. However, all functions do not have a check for the completion of contract initialization.

## Privileged operations

The "Governance" contract has the following requirements:

- only the contract owner can set the rewardDistribution address, which is responsible for starting the reward distribution period for staking governance tokens.
- only the Governance address can call the seize function to transfer tokens (not governance tokens or tokens that are paid out as rewards) from the "Governance" address of the contract to the Governance address.

- only the Governance address can call the setBreaker function to set the value of the corresponding parameter.
- only the Governance address can call the setQuorum function to set the value of the corresponding parameter.
- Only the Governance address can call the setMinimum function to set the value of the corresponding parameter.
- Only the Governance address can call the setPeriod function to set the value of the corresponding parameter.
- only Governance address can call the setLock function to set the value of the corresponding parameter.

No bugs were detected in the implementation of this functionality.

## Features functionality: Token Staking

The "Governance" contract supports ERC20 token staking of one type, designated as "_governanceToken" in the configure function.

To stake a token, the user must approve tokens for the "Governance" contract and then call the "stake" function.

To withdraw tokens, the user must call the "withdraw" function.

The user can call the "balanceOf" function to see the number of tokens he has staked.

The user can call the "totalSupply" function to see the number of all tokens deposited on the contract.

Although the described functionality looks like LP tokens, the "Governance" contract does not support the entire IERC20 interface, because the tokens that are secured cannot be forwarded to other users.

No bugs were found in the implementation of this functionality.

## Features functionality: voting

The "Governance" contract supports the voting functionality for proposals offered by users.

To be able to vote, the user must stake governance tokens and call the "register" function (in any sequence).

To be able to propose a new proposal, the user must have at least 1 governance token (1018token wei. The contract only supports governance tokens with decimals of 18). In addition, the user must publish the text of the proposal in ipfs in advance and publish a contract on the ethereum network which will implement the logic described in the proposal if the decision is accepted or rejected by the community. The address of this contract must be passed by the user to the "propose" function as the "_executor" parameter, and the hash of the ipfs document as the "_hash" parameter of the same function.

Initially, the voting period for any proposal is 17280 blocks. Taking into account the current block production speed (average 13 sec), this period equals 2 days 14 hours. During this time, any user who has the ability to vote, may vote for or against this proposal. The vote will be considered valid if the quorum of votes for the allotted time is equal to 20% of all tokens. The "weight" of the vote is proportional to the size of the user's stake, relative to all tokens.

If a quorum is gathered, the "execute" function of the "_executor" contract that was specified in the "propose" function will execute anyway. So the logic of this function should take into account if the decision is accepted (i.e., a majority of those who voted, vote "for" the proposal), or rejected (i.e., a majority of those who voted, vote "against" the proposal).

In order to vote ("for" or "against" the proposal), the user must be able to vote, and within the above period of time from the moment of publication of the proposal, perform the function "voteFor" (to vote "for") or the function "voteAgainst" (to vote "against"), specifying as parameter "id", the proposal id for which they are voting.

The user can re-vote at any time by calling the corresponding function. The weight of his vote will be subtracted from the previous vote and re-counted.

The user may NOT "abstain" if they have already voted for or against. His vote will be counted anyway.

The user can see the number of for and against votes at any time by calling the getStats function, with the id parameter of the proposal they are interested in.

After the end of the voting period, if there is a quorum of votes for the proposal, the execute function becomes available, which executes the "executor" function of the same name of the contract specified in the proposal.

No bugs were found in the implementation of this functionality, but there is one point of an informational nature.

### Redundant revoke function

Informational

The "revoke" function is the inverse of the "register" function; it stops counting the tokens the user has staked. However, there are no functions in the contract for which this may be required. This function looks redundant, but it does not affect the correct functioning of the contract.

## Features functionality: payout of rewards

Governance contract" supports the functionality of paying out rewards to users who stake governance tokens. This functionality is identical to the functionality of the "StakingReward" contract, through which the Uniswap protocol distributed UNI tokens, and the "Incentivizer" contract, through which XBE tokens were distributed in a liquidity event.

The main difference from the mentioned contracts is that a staker can collect the reward only during 17280 blocks (about 2 days 14 hours) after voting for some proposal or depositing a new proposal. At the same time, the staker cannot withdraw his governance tokens during the same period of time.

The user can see the block number up to which he can receive a reward, and after which he can withdraw his governance tokens by calling the public voteLock field.

The Governance address can cancel the token lock described above by calling the setBreaker function with the parameter true. By default, this parameter is set to false.

No bugs were detected in the implementation of this functionality.

## Redundant _setGovernanceToken function

Low

The function "_setGovernanceToken" described in the base contract "LPTokenWrapper" is not needed because it changes the public variable of the contract. This can be done without using the function, which would be a little cheaper by gas.