SFXDX

VeXBE smart-contract audit report

By: SFXDX Team
For: XBE Finance

www.sfxdx.com

The eurxb.finance VeXBE smart-contract audit report

Introduction **3**

VotingStakingRewards **3**

WithdrawUnbonded function that allow to withdraw an amount in excess of locked funds 4

Redistribution of rewards that have not been distributed 4

Forced setting of the boost value when the tokens blocking for the maximum period 5

The extra variable voting 5

The inheritance from the Initializable contract, without using the logic of upgradeable contracts or proxy contracts 5

The separation of pauser and owner contract logic 5

Outdated version of solidity 6

BaseStrategy **6**

Incomplete support of IStrategy parental interface 6

The excessive support of the meta-transactions 7

The inheritance from the Initializable contract, without using the logic of upgradeable contracts or proxy contracts 7

The fundamental difference between the redefined withdraw(address _token) function and the withdraw (uint256 _amount) function 7

Correctness of the withdraw(uint256 _amount) functions terminates in a case of the setting the parameter that exceeds the Vault and Strategy balance. 8

Including the support of the EnumerableSet library without the using in the contract code. 8

ClaimableStrategy **8**

Incomplete support of IStrategy parental interface 8

SushiStrategy **9**

Redundant structure of Settings 9

SimpleXBEInflation **9**

Excessive zero address check 10

The inheritance from the Initializable contract, without using the logic of upgradeable contracts or proxy contracts 10

The inflation takes place to the future 10

BonusCampaign **11**

Inheritance from the StakingReward contract with fractionally unusable functionality 11

Extra parameters for the processLockEvent function 11

Extra onlyOwner modificator of the startMint function 12

VeXBE **12**

Complicated logic of the _checkpoint function 12

Sub-optimal structures 12

Inability to add locked tokens of the same address to the bonus campaign twice 13

BaseVault **13**

Extra events are generated 13

Partial duplication of function logic 13

Each call to the withdraw function call the claim of all rewards from the strategy 14

Unused return value 14

Partial blocking of functionality by the contract owner 14

VaultWithAutoStake **14**

SushiVault **14**

Introduction

Git repository with contracts:

https://github.com/EURxbfinance/vaults

Audited contracts:

https://github.com/EURxbfinance/vaults/blob/audit_VeXBE/contracts/governance/VotingStakingRewards.sol
https://github.com/EURxbfinance/vaults/blob/audit_VeXBE/contracts/main/strategies/base/BaseStrategy.sol
https://github.com/EURxbfinance/vaults/blob/audit_VeXBE/contracts/main/strategies/base/ClaimableStrategy.sol
https://github.com/EURxbfinance/vaults/blob/audit_VeXBE/contracts/main/strategies/SushiStrategy.sol
https://github.com/EURxbfinance/vaults/blob/audit_VeXBE/contracts/main/SimpleXBEInflation.sol
https://github.com/EURxbfinance/vaults/blob/audit_VeXBE/contracts/main/BonusCampaign.sol
https://github.com/EURxbfinance/vaults/blob/audit_VeXBE/contracts/main/VeXBE.sol
https://github.com/EURxbfinance/vaults/blob/audit_VeXBE/contracts/main/vaults/base/BaseVault.sol
https://github.com/EURxbfinance/vaults/blob/audit_VeXBE/contracts/main/vaults/base/VaultWithAutoStake.sol
https://github.com/EURxbfinance/vaults/blob/audit_VeXBE/contracts/main/vaults/SushiVault.sol

Audited commit:
https://github.com/EURxbfinance/vaults/commit/d979d6ecba063e2eb873f2a886d2fff491bb00ff

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.

VotingStakingRewards

25.08.2021

The “VotingStakingRewards” contract audit identified the weaknesses of the following categories:

Severity		Count
Informational	-	1
Low	-	4
Medium	-	2
High	-	0

WithdrawUnbonded function that allow to withdraw an amount in excess of locked funds

Medium

The function *withdrawUnbonded* was created to withdraw the user`s stakingToken that the user stake on the contract. But to withdraw the awards of the autostaked tokens used another function: *withdrawBondedOrWithPenalty*. The *withdrawUnbonded* have two requirements to prevent withdrawal of the excessive amounts of stakingTokens:

require(_balances[msg.sender].sub(amount) >= escrowed, "escrow amount failure");
require(_balances[msg.sender].sub(bondedRewardLocks[msg.sender].amount) >= amount, "cannotWithdrawBondedTokens");

The first condition means that the overall balance (sum of staked and autostaked tokens) does not exceed the funds locked on the contract after the withdrawal. The second condition means that the difference between the stakes and unstaked tokens does not exceed the amount of tokens that user's request for withdrawal. In isolation these conditions are logical but if the user has the 100% of locked staked funds has the ability to withdraw the amount of tokens that does not exceed the autostaked tokens. After this user can wait for an unbond period and call the withdrawBondedOrWithPenalty function. Thereby the user can get the double amount of autostaked tokens.

The possible fix: Change the conditions above to one condition that will take into account all options:

require(

_balances[msg.sender].sub(escrowed).sub(bondedRewardLocks[msg.sender].amount) >= amount, "escrow amount failure");

Redistribution of rewards that have not been distributed

Medium

Because of the ever-decreasing boost level of users, it is difficult to calculate a fair distribution of the reward between all users. That calculation will have the linear increase of gas usage with the increasing of the user's amount. In this case the gas usage becomes unacceptable. This problem was solved by assigning the boost level of users from 0.4 to 1. All users get the amount of rewards related to the ratio of their funds to the total amount of staked funds. This means that users that do not have the maximum boost level that equal to 1 will receive only 40% of possible reward. The remaining 60% of their rewards transfer to the Treasury contract and then again redistribute between the *VotingStakingRewards* contract stakers.

This allow to make the fair distribution between the users only if the new users will not be added during the next periods. Otherwise, the users that will be added to the contract later will be able to claim the unreleased portion of the rewards of those users who do not have the maximum boost.

Forced setting of the boost value when the tokens blocking for the maximum period

Low

The initial logic of the reward distribution that has the Curve protocol envisaged the ever-decrease of the “vote weight” value in the Voting Escrow contract. The user`s reward was calculated in relation to this value. However in the current version this logic was implemented only for locks less than 23 months. The lock to 23 mounts ignores the decreasing reward. The lock for more than 23 months is impossible. The value of the boost reward remains the maximum during all locking period and exceeds the minimum value 2.5 times. In this case the users that block their funds for 23 months have more profits than users that lock their funds to less period.

The extra variable voting

Low

The *VotingStakingRewards* contract initialize the *voting* variable that contains the voting contract. Then this variable is not used. This variable can be used only by the user for easy search of the corresponding contract. But in the contact there is no logic where this variable is used.

The inheritance from the Initializable contract, without using the logic of upgradeable contracts or proxy contracts

Low

The *VotingStakingRewards* contract is inheritance from the *VotingInitializable* contract. The *VotingInitializable* is a copy of the *Initializable* contract of OpenZeppelin library https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/proxy/utils/Initializable.sol. This contract is used to take out the constructor logic to the initialization function. This contract is the main part of the proxy concept and upgradable contract on the OpenZeppelin library. But in this implementation this contract is used only for the delayed initialisation, so this can be misleading.

The delayed initialisation gives the flexibility in use and allows to initialise the contract later rather than at the time of deployment. But in this case there is a danger that the contract that was not initialised will be misused.

The separation of pauser and owner contract logic

Low

The common style of contract`s operation inherited from *Pausable* implies that pause/unpause functions are available to the contract owner. However in the current implementation of the basic *VotingPausable* contract these functions are the privilege to the pauser address. The pauser address is set in the constructor. The problem is that this parameter cannot be changed after the deployment. So the *setPaused* function that allows to pause the *stake* and *stakeFor* functions is available only to the address that deployed the contract.

Possible fix: add a function for the pauser permission`s transfer, or merge the logic of pauser and owner privileged addresses.

Outdated version of solidity

Informational

The *Voting* contract that related with the *VotingStakingRewards* contract inherited from the *AragonApp* contracts that are based on the 0.4.24 Solidity version

Consequently the VotingStakingRewards contract is based on the same Solidity version. The use of the outdated version of Solidity makes it impossible to use the current version of the open source libraries such as OpenZeppelin. This leading to the need to redefine some basic contracts:
VotingPausable instead of *Pausable*, *VotingOwnable* instead of *Ownable*, *VotingNonReentrant* instead of *ReentrancyGuard*, *VotingInitializable* instead of *Initializable*. This generates a higher probability of making a mistake or encountering unverified logic.

BaseStrategy

25.08.2021

The “BaseStrategy” contract audit identified the weaknesses of the following categories:

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

Incomplete support of IStrategy parental interface

Low

The BaseStrategy contract is inherited from the IStrategy interface, but has no redefined methods:

function getRewards() external;
function claim(address) external returns (bool);
function deposit() external;

The redefining of these methods is required in the child contracts. Such implementation makes the contract`s code confusing and leads to the non-obvious compilation errors. In addition there exist the *withdraw* functions without the *deposit* function also there are two functions with similar logic:*claim* and *getRewards*. These functions get the rewards but make it from the potential different sources. All these facts indicate the poor contracts design.

The excessive support of the meta-transactions

Low

The *BaseStrategy* contract implements the support of the meta-transactions by implicit inheritance from the *Context* contract and using of the *_msgSender()* instead of *msg.sender*. However the child and contiguous contracts do not support the meta-transactions. This makes the support of the meta-transactions by the *BaseStrategy* contract make it meaningless and suboptimal in terms of the gas usage.

Possible fix: refuse the support of the meta-transactions in favour of gas usage optimisation.

The inheritance from the Initializable contract, without using the logic of upgradeable contracts or proxy contracts

Low

The *BaseStrategy* contract is inherited from the *Initializable* contract of the OpenZeppelin library https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/proxy/utils/Initializable.sol.This contract is used to take out the constructor logic to the initialization function. This contract is the main part of the proxy concept and upgradable contract on the OpenZeppelin library. But in this implementation this contract is used only for the delayed initialisation, so this can be misleading.

The delayed initialisation gives the flexibility in use and allows to initialise the contract later rather than at the time of deployment. But in this case there is a danger that the contract that was not initialised will be misused.

The fundamental difference between the redefined withdraw(address _token) function and the withdraw (uint256 _amount) function

Informational

The *BaseStrategy* contract has two *withdraw* functions that implement the different logic. The function *withdraw(address _token)* can be called only by the controller contract. This function makes the transfer of all available tokens from the strategy to the calling address.The token address is set in the parameter, so this function can be called multiple times for different tokens.

As opposed to *withdraw(address _token)* function, *withdraw(uint256 _amount)* function can be called by the *controller* contract or the *vault* contract. This function makes the transfer of the specified number of _want tokens to the *vault* address.

Such function calling makes the contract`s code confusing and indicates poor contract design.

Correctness of the termination of the withdraw(uint256 _amount) functions in a case of the setting the parameter that exceeds the Vault and Strategy balance.

Informational

The *withdraw(uint256 _amount)* function of the *BaseStrategy* can be called with the parameter that exceeds the sum of the Vault and Strategy balances.In this case the calling of the virtual function *_withdrawSome* appears. This function withdraws the maximal amount of the _want tokens from the Strategy contract and external protocols in which the strategy has invested tokens. Also the *_withdrawSome* function has to be redefined in the child contracts. The user is not informed that the withdrawal has not been made in full and that the strategy has run out of tokens.

Possible fix: add the event that indicates that Strategy can withdraw the fewer tokens than the requested amount.

Including the support of the EnumerableSet library without the using in the contract code.

Informational

The contract *BaseStrategy* include the support of the EnumerableSet library:
using EnumerableSet for EnumerableSet.AddressSet;
But there are no variables or any method calls to this library in the contract code. This code needs to be deleted to optimise the gas usage during the deployment.

ClaimableStrategy

25.08.2021

The “ClaimableStrategy” contract audit identified the weaknesses of the following categories:

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

Incomplete support of IStrategy parental interface

Low

The *ClaimableStrategy* contract is inherited from the *BaseStratagy* contract that inherited from the *IStrategy* interface. However, like the parent contract, it has no redefined methods:

function getRewards() external;

function deposit() external;

So the *ClaimableStrategy* contract is a pre-defining of an abstract *BaseStratagy* contract by the *claim* method. This method is a function of the rewards transfer to the Vault contract and call the *notifyRewardAmount* and actually become the start of the reward distribution.

Such rewards distribution logic is not "fair" for the user that stake the tokens in the Vault before and earns this reward. In this case the user that stakes the tokens just before the reward distribution has the same reward rate as the user who stakes the token to the Vault before the previous claim function.

However, this problem does not seem significant due to the fact that demotivation of the users do not occur longer than one distribution period and users that receive foreign rewards have to stake the tokens in the Vaults during all reward distribution periods until the next call of claim function. And if the user withdraws their funds from the Vault and their rewards will be redistributed between other stakers.

SushiStrategy

25.08.2021

The “SushiStrategy” contract audit identified the weaknesses of the following categories:

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

Redundant structure of Settings

Low

The structure of the Settings is defined in the *SushiStrategy* contract.This structure contains the fields with the addresses of the contiguous contracts that is necessary for the SushiStrategy functioning. But this structure is used only once over contract configuration and was created only to reduce the number of configuration function variables by the additional gas costs.

Possible fix: Refuse the using of this structure, replace it by 2 variables

SimpleXBEInflation

25.08.2021

The “SimpleXBEInflation” contract audit identified the weaknesses of the following categories:

Severity		Count
Informational	-	2
Low	-	1
Medium	-	0
High	-	0

Excessive zero address check

Low

In the function *mintForContracts* of the *SimpleXBEInflation* implemented the check that it is impossible to transfer the token to the zero address:

require(_to != address(0), "!zeroAddress");

However this checking is redundant. The XBE token is inherited from the ERC-20 token. And the_mint method of the basic ERC20 token contract has it:

require(account != address(0), "ERC20: mint to the zero address");

The additional check of this functionality is not optimal for the gas usage.

The inheritance from the Initializable contract, without using the logic of upgradeable contracts or proxy contracts

Informational

The “SimpleXBEInflation” contract is inherited from the *Initializable* contract of the OpenZeppelin library https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/proxy/utils/Initializable.sol. This contract is used to take out the constructor logic to the initialization function.This contract is the main part of the proxy concept and upgradable contract on the

OpenZeppelin library. But in this implementation this contract is used only for the delayed initialisation, so this can be misleading.

The delayed initialisation gives the flexibility in use and allows to initialise the contract later rather than at the time of deployment. But in this case there is a danger that the contract that was not initialised will be misused.

The inflation takes place to the future

Informational

In the *SimpleXBEInflation* function the *mintForContracts* function mint new tokens gradually once in a specified period. And this does not happen at the end of the period. The token mint happens in advance. This was made to mint the token once and send it to the contracts inherited from *the StakingReward* contract. This contract will gradually distribute the tokens between the stakers of the *VotingStakingRewards* and *SushiVault* contracts. This is not an issue but such implementation have to be taken into account for adding the recipient contract that can potentially be non-inherited from the *StakingReward* contract and receive the rewards from the *SimpleXBEInflation*

BonusCampaign

25.08.2021

The “BonusCampaign” contract audit identified the weaknesses of the following categories:

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

Inheritance from the StakingReward contract with fractionally unusable functionality

Low

The *BonusCampaign* contract is inherited from the *StakingReward* contract. The *StakingReward* contract implements the basic stacking functionality. But the *BonusCampaign* contract doesn't stake the tokens. So the functions that related with the *stake, withdraw notifyRewardAmount* redefined and not supported. This was made to the users have the ability to lock their tokens in other contract *VotingStakingRewards* but receive additional rewards for their lock from the *BonusCampaign* contract. However the excessive logic of the parent contract is a consequence of the insufficient elaboration of the general contracts architecture.
Moreover the excessive functions lead to the excessive gas costs for a contract deployment.

Extra parameters for the processLockEvent function

Low

The *processLockEvent* function implements the automatic registration for the users while users lock their token in the *VotingStakingRewards* contract. The participation in the bonus campaign takes into account only the period of token lock so instead 4 parameters that

the *processLockEvent* takes: *address account, uint256 lockStart, uint256 lockEnd, uint256 amount* could be made 2 parameters: *address account, uint256 duration*.

The current parameters set is excessive and not optimal for gas usage. But these parameters have been chosen for the universatility. In this implementation the additional bonus campaign that depends not only on the time lock but on the amount of the locked tokens can be added if it will be required

Extra onlyOwner modificator of the startMint function

Low

The *startMint* function can be called once. This function mint the rewards and establish the time of the token distribution start. In this implementation before the *startMintTime* there is no difference by whom and when this function will be called. Consequently this extra *onlyOwner* modificator only makes the contract less use friendly and don't implement any additional logic.

VeXBE

25.08.2021

The “VeXBE” contract audit identified the weaknesses of the following categories:

Severity		Count
Informational	-	0
Low	-	2
Medium	-	1
High	-	0

Complicated logic of the _checkpoint function

Medium

The _checkpoint function was moved from the *VotingEscrow* contract that was written on Vyper and can not be manually audited in detail. The control of the correspondence of the declared logic is possible only through the tests. But in the init test the testing of this function is not exhaustive. Consequently there is a danger that this function may produce incorrect results in some edge cases.

Sub-optimal structures

Low

The VeXBE contract use two structures for the storage the information about the token lock:

mapping(address => LockedBalance) public locked;

mapping(address => uint256) internal _lockStarts;

The first one store the amount and timestamp of the lock end. The second structure store the timestamp of the lock start. Such storage format is not erroneous, but it makes the code more complicated and results in the use of more variables.

Inability to add locked tokens of the same address to the bonus campaign twice

Low

The VeXBE contract does not store its own data about the staked tokens and receives the balances from the *votingStakingRewards* contract. Moreover it carries out the automatic registration of the users who want to participate in the bonus campaign through the call of the *BonusCampaign* contract. As a result the *VeXBE* contract can not be protected from manipulation with the staking tokens balances.

BaseVault

25.08.2021

The “BaseVault” contract audit identified the weaknesses of the following categories:

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

The basic Vauit contract implement the combination of the ERC-20 logic (Vault`s LP tokens) and StakingReward contract that have been modified for the rewards distribution in the different tokens

Extra events are generated

Low

The *BaseVault* contract is inherited from the ERC-20 implementation contract *ERC20Vault* so when the of _mint and _burn functions are called the following events is generated:

event Transfer(address indexed from, address indexed to, uint256 value);

This event transfers the funds from/to zero address accordingly. However in the *BaseVault* functions that calls the _mint and _burn function the own events are generated:

event Staked(address indexed user, uint256 amount);

event Withdrawn(address indexed user, uint256 amount);

These events do not add any information so it is superfluous.

Partial duplication of function logic

Low

The *potentialRewardReturns* function is duplicate the work of the *earned* function logic with some extension with the *duration* parameters. So the *earned* function is redundant and can be deleted.

Also the *_rewardPerTokenForDuration* function duplicates the work of the *rewardPerToken* function logic with some extension with the *duration* parameters. So the *rewardPerToken* function is redundant and can be deleted.

.

Each call to the withdraw function call the claim of all rewards from the strategy

Low

The *BaseVault* contract has the 3 public withdraw methods:
withdrawAll()
withdraw(uint256 _amount)
withdraw(uint256 _amount, bool _claimUnderlying)
The flirts and second methods calls the third method. And the third method call the *_getRewardAll(_claimUnderlying)* function. This function call the *claim* of the all rewards from the strategy and related protocols. Thus each call of the withdraw functions is gas expensive and potentially can fail if there are any problems at the stage of receiving the rewards. The absence of an emergency token withdrawal function seems potentially dangerous for users.

Unused return value

Informational

The *_withdraw(uint256 _amount)* function return the uint256 value. However the withdraw public function that call it have no the return value. So this logic is useless.

Partial blocking of functionality by the contract owner

Informational

The owner of the *BaseVault* contract can pause only the deposit related functions. the owner can not pause functions related to the withdraw or claim. It prevents new users from using the contract, but does not stop users from withdrawing funds if requested.

VaultWithAutoStake

25.08.2021

The VaultWithAutoStake contract implements the transfer of the reward token to the third-party contract as a stake instead of sending it to the user. According to the audit results of the “VaultWithAutoStake” contract, no errors were found.

SushiVault

25.08.2021

The *SushiVault* contract implements the *BaseVault* and *VaultWithAutoStake* functionality without any additional logic. According to the audit results of the *“SushiVault”* contract, no errors were found.