Blaize

Know the rules.
Create new ones

SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Table of Contents

This document may contain confidential information about IT systems and the intellectual property of the Customer and information about potential vulnerabilities and methods of their exploitation. The report containing confidential information can be used internally by the Customer, or it can be disclosed publicly after all vulnerabilities are fixed - upon a decision of the Customer.

Abstract

In this report, we consider the security of the **Jibrel** contracts for **Staking protocol**. Our task is to find and describe security issues in the smart contracts of the platform. This report presents the findings of the security audit of Customer's smart contracts conducted between **March 10th, 2021 - March 16th, 2021.**

Post-audit validation provided on **March, 17-18th, 2021**.

Disclaimer

The audit does not give any warranties on the security of the code. One audit can not be considered enough. We always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts. Besides, security audits are not investment advice.

Scope

The scope of the audit is the **"staking-protocol"** project at the main branch with commit 1dd05f60e4a3feef937ccd86ea88bbf3ae73cfea.

Post-audit scope for validation includes a **"staking-protocol"** project at the main branch with commit f744c86afb63e823d23f05e6bedce600364e1170.

1. StakingMilestones/IStakingMilestones.sol
2. StakingMilestones/StakingMilestones.sol
3. StakingMilestones/Vault.sol
4. StakingMilestones/YieldFarm.sol
5. StakingMilestones/YieldFarmLP.sol

We have scanned this smart contract for commonly known and more specific vulnerabilities. Here are some of the commonly known

vulnerabilities that are considered (the full list includes them but is not limited to):

- Unsafe type inference;
- Timestamp Dependence;
- Reentrancy;
- Implicit visibility level;
- Gas Limit and Loops;
- Transaction-Ordering Dependence;
- Unchecked external call - Unchecked math;
- DoS with Block Gas Limit;
- DoS with (unexpected) Throw;
- Byte array vulnerabilities;
- Malicious libraries;
- Style guide violation;
- ERC20 API violation;
- Uninitialized state/storage/local variables;
- Compile version not fixed.

Procedure

In our report we checked the contract with the following parameters:

- Whether the contract is secure;
- Whether the contract corresponds to the documentation;
- Whether the contract meets best practices in efficient use of gas, code readability;

We perform our audit according to the following procedure:

1. Automated analysis:
    - Scanning contract by several public available automated analysis tools such as Mythril, Solhint, Slither and Smartdec;
    - Manual verification of all the issues found by tools.
2. Manual audit:
    - Manual analysis of smart contracts for security vulnerabilities;
    - Checking smart contract logic and comparing it with one described in the documentation.

Executive summary

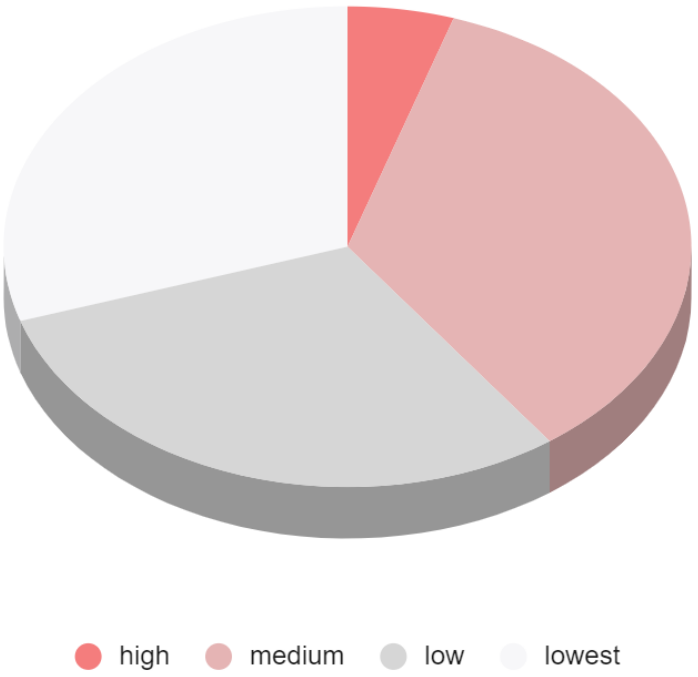
According to the assessment, the Customer's smart contracts required improvements; some functionality works semi-auto and do not follow best practices. We described all the issues and added recommendations for the Customer. Though, the Customer's team has provided **all** necessary improvements according to the Auditor's recommendations. Most of the issues were connected with the checks for constructor and initializers, though they were implemented in the deployment script.

For the overall security of the smart-contracts system can not be evaluated as sufficiently secure and has **97** out of **100**.

Findings:

	Found	Fixed
Critical	**0**	**0**
High	**1**	**1 (marked as resolved)**
Medium	**7**	**6 = 5 + 1 (marked as resolved)**
Low	**6**	**6 = 1 + 5 (marked as resolved)**
Lowest	**6**	**5 = 2 + 3 (marked as resolved)**

The graph of vulnerabilities distribution:



Severity Definition

Critical	A system contains several issues ranked as very serious and dangerous for users and the secure work of the system. Needs immediate improvements and further checking.
High	A system contains a couple of serious issues, which lead to unreliable work of the system and might cause a huge information or financial leak. Needs immediate improvements and further checking.
Medium	A system contains issues which may lead to medium financial loss or users' private information leak. Needs immediate improvements and further checking.
Low	A system contains several risks ranked as relatively small with the low impact on the users' information and financial security. Needs improvements.

Lowest	A system does not contain any issue critical to the secure work of the system, yet is relevant for best software defensive practices implementations.

AS-IS overview

StakingMilestones contract overview

StakingMilestones contract inherits from the ReentrancyGuardUpgradeSafe and OwnableUpgradeSafe contracts. Initialize function sets Unix timestamp.

- From Initializable
  - isConstructor() (private)
- From ContextUpgradeSafe
  - __Context_init() (internal)
  - __Context_init_unchained() (internal)
  - _msgData() (internal)
  - _msgSender() (internal)
- From OwnableUpgradeSafe
  - __Ownable_init() (internal)
  - __Ownable_init_unchained() (internal)
  - owner() (public)
  - renounceOwnership() (public)
  - transferOwnership(address) (public)
- From ReentrancyGuardUpgradeSafe
  - __ReentrancyGuard_init() (internal)
  - __ReentrancyGuard_init_unchained() (internal)
- Native functions
  - initialize(uint256, uint256) (public) - sets Unix timestamp and initializes ReentrancyGuardUpgradeSafe and OwnableUpgradeSafe
  - deposit(address, uint256) (public) - deposits a certain amount of tokens to the StakingMilestones contract; stores new balance. Executes the epoch logic
  - withdraw(address, uint256) (public) - withdraws a certain amount of tokens to the sender; stores new balance. Executes the epoch logic
  - manualEpochInit(address[], uint128) (public) - initializes the current and next epoch.

- emergencyWithdraw(address) (public) - withdraws a total balance of tokens to the sender; stores new balance.
- getEpochUserBalance(address, address, uint128) → uint256 (public) - returns the valid balance of a user that was taken into consideration in the total pool size for the epoch
- balanceOf(address, address) → uint256 (public) - returns the amount of tokens that the user has currently staked
- getCurrentEpoch() → uint128 (public) - returns the id of the current epoch derived from block.timestamp
- getEpochPoolSize(address, uint128) → uint256 (public) - returns the total amount of tokens that was locked from beginning to end of epoch identified by `epochId`
- currentEpochMultiplier() → uint128 (public) - returns the percentage of time left in the current epoch
- computeNewMultiplier(uint256, uint128, uint256, uint128) → uint128 (public) - returns computed `NewMultiplier`
- epochIsInitialized(address, uint128) → bool (public) - checks if an epoch is initialized, meaning we have a pool size set for it
- getCheckpointBalance(StakingMilestones.Checkpoint) → uint256 (internal) - returns checkpoint balance
- getCheckpointEffectiveBalance(StakingMilestones.Checkpoint) → uint256 (internal) - returns checkpoint effective balance

Vault contract overview

Inherits from the OwnableUpgradeSafe. Initializing function sets slice address.

- From Initializable
  - - isConstructor() (private)
- From ContextUpgradeSafe
  - - __Context_init() (internal)
  - - __Context_init_unchained() (internal)
  - - _msgData() (internal)
  - - _msgSender() (internal)
- From OwnableUpgradeSafe
  - __Ownable_init() (internal)
  - __Ownable_init_unchained() (internal)
  - owner() (public)
  - renounceOwnership() (public)
  - transferOwnership(address) (public)
- Native functions

- setAllowance(address, uint256) (public) - Sets amount as the allowance of spender over the caller's tokens
- constructor(address) (public) - sets slice address

YieldFarm contract overview

Inherits from the OwnableUpgradeSafe.

- From Initializable
  - isConstructor() (private)
- From ContextUpgradeSafe
  - __Context_init() (internal)
  - __Context_init_unchained() (internal)
  - _msgData() (internal)
  - _msgSender() (internal)
- From OwnableUpgradeSafe
  - __Ownable_init() (internal)
  - __Ownable_init_unchained() (internal)
  - owner() (public)
  - renounceOwnership() (public)
  - transferOwnership(address) (public)
- Native functions
  - initialize(address,address,address,address,uint256)(public) - sets Ownable, sliceAddress, stakableToken, totalRewardInEpoch, stackContract, vault, epochStart, epochDuration
  - setTotalRewardInParticularEpoch(uint128,uint256)(external) - sets total reward in particular epoch
  - massHarvest() → uint256 (external) - method to harvest all the unharvested epochs until current epoch - 1
  - harvest(uint128) → uint256 (external) - credits the sender with Reward and returns the amount Reward
  - getPoolSize(uint128) → uint256 (external) - calls to the staking smart contract to retrieve the epoch total pool size
  - getCurrentEpoch() → uint256 (external) - calls to the staking smart contract to retrieve the current epoch id
  - getEpochStake(address,uint128) → uint256 (external) - calls to the staking smart contract to retrieve user balance for an epoch

- userLastEpochIdHarvested() → uint256 (external) - returns user last Epoch Id Harvested
- getTotalAccruedRewards(address) → uint256 (external) - gets total accrued rewards
- _initEpoch(uint128) (internal) - sets epoch Id
- _harvest(uint128) → uint256 (internal) -returns the amount of reward
- _getPoolSize(uint128) → uint256 (internal) - returns the pool size
- _getUserBalancePerEpoch(address,uint128) → uint256 (internal) - returns the user balance per epoch
- _getEpochId() → uint128 (internal) - returns the epoch id

YieldFarmLP contract overview

Inherits from the OwnableUpgradeSafe.

- From Initializable
  - isConstructor() (private)
- From ContextUpgradeSafe
  - __Context_init() (internal)
  - __Context_init_unchained() (internal)
  - _msgData() (internal)
  - _msgSender() (internal)
- From OwnableUpgradeSafe
  - __Ownable_init() (internal)
  - __Ownable_init_unchained() (internal)
  - owner() (public)
  - renounceOwnership() (public)
  - transferOwnership(address) (public)
- Native functions
  - initialize(address,address,address,address,uint256)(public) - sets Ownable, sliceAddress, stakableToken, totalRewardInEpoch, stackContract, vault, epochStart, epochDuration
  - addStakableToken(address, uint) (external) - adds a stackable token contract address, along with its weight
  - removeStakableToken(address) (external) - removes a stackable token contract address, along with its weight

- setTotalRewardInParticularEpoch(uint128,uint256)(external) - sets total reward in particular epoch
- massHarvest() → uint256 (external) - method to harvest all the unharvested epochs until current epoch - 1
- harvest(uint128) → uint256 (external) - credits the sender with Reward and returns the amount Reward
- getPoolSize(uint128) → uint256 (external) - calls to the staking smart contract to retrieve the epoch total pool size
- getCurrentEpoch() → uint256 (external) - calls to the staking smart contract to retrieve the current epoch id
- getEpochStake(address,uint128) → uint256 (external) - calls to the staking smart contract to retrieve user balance for an epoch
- userLastEpochIdHarvested() → uint256 (external) - returns user last Epoch Id Harvested
- getTotalAccruedRewards(address) → uint256 (external) - gets total accrued rewards
- _initEpoch(uint128) (internal) - sets epoch Id
- _harvest(uint128) → uint256 (internal) -returns the amount of reward
- _getPoolSize(uint128) → uint256 (internal) - returns the pool size
- _getUserBalancePerEpoch(address,uint128) → uint256 (internal) - returns the user balance per epoch
- _getEpochId() → uint128 (internal) - returns the epoch id

Audit overview

Critical

No critical issues detected.

High

1. **StakingMilestones.sol**. No restrictions for staked token addresses

   Evidence: Contract functionality does not contain any restrictions

about tokens to be used in deposit/withdraw functionalities (and other connected with staking). There are no checks if token is allowed to be staked, or registry of supported tokens, etc. And there is no clear documentation if all possible tokens are allowed or there are restrictions.

Recommendation: Check the rules of tokens participation in staking or implement the “supported tokens” functionality with appropriate checks in crucial places.

Resolution: Marked as resolved after the discussion with the Client.
Comment: The check was purposely skipped because it is performed in YieldFarm. If someone does stake a token which is not stakeable in YieldFarm it won't fetch the user rewards and he/she can withdraw their tokens from StakingMilestones whenever they want.

Medium

1. **Vault.sol, 21**. Use safeERC20 for transfers and approvals

   Evidence: Vault.setAllowance()

   Recommendation: Change *approve()* to *safeApprove()*.

   Resolution: Fixed by the Client.

2. **StakingMilestones.sol**. Use safeERC20 for transfers and approvals

   Evidence:
   StakingMilestones.deposit()
   StakingMilestones.withdraw()
   StakingMilestones.emergencyWithdraw()

   Recommendation: Change *transfer()* to *safeTransfer()*, *transferFrom()* to *safeTransferFrom()*.

   Resolution: Fixed by the Client.

3. **YieldFarm.sol**. Use safeERC20 for transfers and approvals

   Evidence:
   YieldFarm.massHarvest()
   YieldFarm.harvest()

   Recommendation: Change *transfer()* to *safeTransfer()*, *transferFrom()* to *safeTransferFrom()*.

   Resolution: Fixed by the Client.

4. **YieldFarmLP.sol**. Use safeERC20 for transfers and approvals

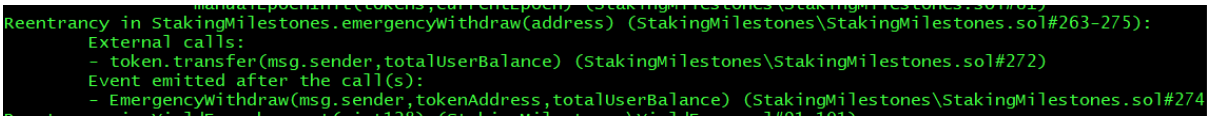
   Evidence:
   YieldFarmLP.massHarvest()
   YieldFarmLP.harvest()

   Recommendation: Change *transfer()* to *safeTransfer()*, *transferFrom()* to *safeTransferFrom()*.

   Resolution: Fixed by the Client.

5. **StakingMilestones.sol, 263**. Add *nonReentrant* for *emergencyWithdraw()*

   Evidence: Add *nonReentrant* modifier for *emergencyWithdraw()* method. It is low risk of reentrant attack, though, since there is no checks for the supported tokens and to keep the same system with *deposit()* and *withdraw()* we recommend to add the check.



   Recommendation: Add *nonReentrant* modifier for *emergencyWithdraw()* method.

Resolution: Fixed by the Client.

6. **YieldFarmLP.sol**, **71** addStakableToken() method also allows you to remove a token.

   Evidence: There is no check for 0 value AND for 0 address passed into the method. It can cause incorrect irrevertible storage changes. Since additional irrevertible changes are made in the storage and consequences solution requires a lot of gas to be performed, this is marked as medium

   Recommendation: Add require() statement to check 0 value and/or 0 address passed.

7. **StakingMilestones.sol**, **53** check for 0 value.

   Evidence: There is no check for 0 in initialize function. It has low risk in general, since initialization can be performed only by the owner. Though, since there is no way to change the value and the value participates in division it is better to reduce the risk of zero division.

   Recommendation: Add require() statement to check 0 value.

   Resolution: Marked as resolved after the discussion with the Client.
   Comment: After the discussion with the Client, set up that the value is checked in the deployment script.

Low

1. **Vault.sol**. Missing documentation

   Evidence: Vault.sol methods do not have proper documentation neither in Doxygen format within the code nor in the separate Readme

   Recommendation: Add documentation for the Vault contract methods.

2. **StakingMilestones.sol**, **78** Possible revert in *deposit()*.

   Evidence: deposit() method calls to *manualEpochInit()* in case if the epoch is uninitialized. Also, *manualEpochInit()* reverts if previous epoch is uninitialized. Though, there can be more than 1 epoch uninitialized. In this case the user will get the reverted deposit() transaction with no clear error, so it is not clear that he will need to initialize several epochs as well.

   Recommendation: Add error message, that more than 1 epoch is uninitialized and check the functionality if it is acceptable (user gets an error and needs to initialize several epochs manually).

   Resolution: Marked as resolved after the discussion with the Client.
   Comment: The check was purposely skipped following the established model and userflow. Initialization is required for every epoch either manually or by regular deposit/withdraw. Current behavior is acceptable.

3. **YieldFarm.sol**, **53; YieldFarmLP.sol**, **58**
   Check for the correct *slice* address.

   Evidence: YieldFarm.sol and YieldFarmLP.sol *initializer()* methods get **slice** address and **vault** address as arguments, though check for correct slice address (crucial value in Vault.sol) is missing.

   Recommendation: Add require() statement to check that **slice** is the same as the value kept in the **vault**.

   Resolution: Marked as resolved after the discussion with the Client.
   Comment: After the discussion with the Client, set up that the value is checked in the deployment script.

4. **YieldFarm.sol**, **72** Not optimal failure point.

   Evidence: YieldFarm.sol *massHarvest()* has inappropriate statement to check the epoch is greater than 0 and the transaction will fail with inappropriate error.

```
function massHarvest() external returns (uint){
    uint totalDistributedValue;
    uint epochId = _getEpochId().sub(1); // fails in epoch 0
```

Recommendation: Add require() statement to check that **epoch** is greater than 0.

Resolution: Marked as resolved after the discussion with the Client.

5. **YieldFarm.sol**, **182** Not optimal calculation.

Evidence: YieldFarm.sol *getTotalAccruedRewardsForToken()* has overcomplicated calculation.

Expression `((_staking.getEpochUserBalance(userAddress, token, epochId)).mul(weightOfStakableToken[token])).div(100)` can be encapsulated into a separate function to simplify the calculation.
The same applies to *_getUserBalancePerEpoch()*.
Also, check the calculation correctness based on:

```
YieldFarmLP.getTotalAccruedRewardsForToken(address,address) (StakingMilestones\YieldFarmLP.sol#182-193) performs a
plication on the result of a division:
        -rewards = rewards.add(totalRewardInEpoch[i].mul(((_staking.getEpochUserBalance(user,token,i)).mul(weightO
bleToken[token])).div(100)).div(_getPoolSize(i))) (StakingMilestones\YieldFarmLP.sol#189-191)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#divide-before-multiply
```

Recommendation: Refactor the calculation as mentioned.

Resolution: Marked as resolved after the discussion with the Client.
Comment: Calculation is confirmed by the Client.

6. **YieldFarm.sol**, **85** Unneeded variable.

Evidence: YieldFarm.sol *removeStakableToken()* has a variable which can be reduced. Variable **index** is used for storage of **i** value. The first cycle can have a **break** statement instead of **index** assignment, so the second cycle can use **i** instead of index.

```
YieldFarmLP.removeStakableToken(address).index (StakingMilestones\YieldFarmLP.sol#90) is a local variable never in
zed
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-local-variables
INFO:Detectors:
```

Recommendation: Refactor the calculation as mentioned.

Resolution: Fixed by the Client.

Lowest

Informational statements

1. **Vault.sol**. Missing check for 0 address.

   Evidence: Both *constructor()* and *setAllowance()* methods do not have checks for 0 address. Since both methods can be called only from the contracts owner there is no risk, though missing check may lead to the error from the deployer/owner side.

   Recommendation: Add checks for 0 address for both methods.

   Resolution: Marked as resolved after the discussion with the Client.
   Comment: After the discussion with the Client, set up that the value is checked in the deployment script.

2. **YieldFarm.sol, 45**. Missing check for 0 address.

   Evidence: initialize() method do not have checks for 0 address for **vault** address.

   Recommendation: Add checks for 0 address for initializer.

   Resolution: Marked as resolved after the discussion with the Client.
   Comment: After the discussion with the Client, set up that the value is checked in the deployment script.

3. **YieldFarmLP.sol, 51**. Missing check for 0 address.

   Evidence: initialize() method do not have checks for 0 address for **vault** address.

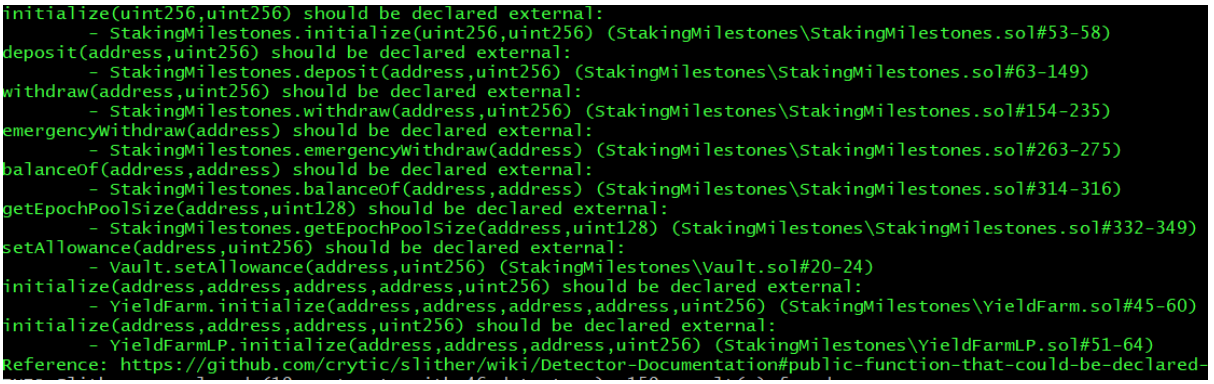
   Recommendation: Add checks for 0 address for initializer.

   Resolution: Marked as resolved after the discussion with the Client.
   Comment: After the discussion with the Client, set up that the value

is checked in the deployment script.

4. Methods should be marked as external.

   Evidence: from the automatic tool output



   Recommendation: mark methods as external

   Resolution: Fixed by the Client.

5. **StakingMilestones.sol**, **264** “Magical” number.

   Evidence: emergencyWithdraw() method contains require() statement which refers to undocumented number “10”.

   Recommendation: Add a constant and proper documentation for the unreferenced number.

6. **YieldFarmLP.sol**, **85** Unnecessary statement.

   Evidence: removeStakableToken() method contains an extra **delete** statement which can be reduced to simple 0 (or 0 address) assignment with some gas savings.

```
delete weightOfStakableToken[_tokenAddress]
delete stakableToken[j]
```

   Recommendation: Replace **delete** with 0 (or 0 address) assignment.

   Resolution: Fixed by the Client.

Unit Test Coverage

All present tests can be successfully run. Nevertheless, the non-standard approach for test writing does not allow to check the test coverage in an automatic way, so manual review for tests coverage was applied. The Auditor's team has considered that the project has sufficient test coverage.

Conclusion

According to the audit the contract was manually reviewed and analyzed with static analysis tools. The Audit team has found some **high**, **medium** and **low** issues during the analysis. Though, all issues were fixed by the Customer's team following Auditor's recommendations. Most of the issues were connected with the checks for constructor and initializers, though they were implemented in the deployment script. Due to the foundings and fixes, the contracts system can be marked as secure.

The overall security of the smart-contracts system can be evaluated as **97** out of **100**.

Audit report contains all necessary information related to it as well as recommendations for their elimination.