HackCtrl

SMART CONTRACT
SECURITY ANALYSIS

Report for:	EDDA
Date:	1 Jan 2021

Document:

Name	Smart Contract Code Review and Security Analysis Report for DeFiBench
Platform	Ethereum / Solidity
Link	
Date of the first audit	01.01.2021
Version of the first audit	fa9001af7d575fb4a24d035b9b39f6b48544dee07bfdd768793f051008c7aa09
Date of the secondary audit	
Version of the secondary audit	

Table of contents

Introduction

HackControl (Consultant) was contracted by (Customer) to conduct a Smart Contract Code Review and Security Analysis. This report presents findings of the security assessment of the Customer's smart contract and its code review conducted 27 December 2020 – 1 January 2021.

The objectives of the audit are the following:

1. Determine correct functioning of the contract in accordance with the project specification
2. Determine possible vulnerabilities, which could be exploited by an attacker
3. Determine the contract bugs, which might lead to its unexpected behavior
4. Analyze whether best practice has been applied during for development of the contract
5. Provide recommendations to improve the code security and readability

This report represents the summary of our findings. As with any code audit, there is a limit to which vulnerabilities can be found, and unexpected execution paths may still be possible. The author of this report does not guarantee a 100% security of the contract, pls see the disclaimer.

Scope

The code has been provided by the developer in the form of a zip archive file with the following SHA256 hash value: fa9001af7d575fb4a24d035b9b39f6b48544dee07bfdd768793f051008c7aa09

We've scanned the smart contract for known and specific vulnerabilities. Below you'll find a list of some of commonly known vulnerabilities that are considered (the full list includes but is not limited to these):

- Reentrancy
- Ownership Takeover
- Timestamp Dependence
- Gas Limit and Loops
- DoS with (Unexpected) Throw
- DoS with Block Gas Limit
- Transaction-Ordering Dependence
- Style guide violation
- Costly Loop
- ERC20 API violation

- Unchecked external call
- Unchecked math
- Unsafe type inference
- Implicit visibility level
- Deployment Consistency
- Repository Consistency
- Data Consistency
- Business Logics Review
- Functionality Checks
- Access Control & Authorization
- Escrow manipulation
- Token Supply manipulation
- Assets integrity
- User Balances manipulation
- Kill-Switch Mechanism
- Operation Trails & Event Generation

Methodology

The audit has been performed in the following steps:

1. Gaining an understanding of the contract's intended purpose by reading the available documentation and codebase

2. Automated scanning of the contract with static code analysis tools for security vulnerabilities and use of best practice guidelines.

    - we scan project's smart contracts with several publicly available auto- mated Solidity analysis tools such as Remix, Mythril and Solhint
    - we manually verify (reject or confirm) all the issues found by tools

3. Manual line by line analysis of the contracts source code for security vulnerabilities and use of best practice guidelines, including but not limited to:

    - Reentrancy analysis
    - Race condition analysis
    - Front-running issues and transaction order dependencies
    - Time dependencies

- Under- / overflow and math precision issues
- Function visibility Issues
- Possible denial of service attacks
- Storage Layout Vulnerabilities

4. Report preparation

Executive Summary

According to the assessment, the Customer's smart contracts are well secure.

Insecure	Poor secured	Secured	Well-secured

You are here 

Our team performed analysis of the code functionality, its manual audit and automated checks with solc, Mythril, Slither and remix IDE. All issues found during the automated analysis were manually reviewed and applicable vulnerabilities are presented in the "Audit Overview" section. General overview is presented in the "AS-IS" section and all found issues can be found in the "Audit Overview " section.

We didn't find any vulnerabilities in the smart contract but outlined two (2) concerns.

Severity Definitions

Risk Level	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to the token losses
High	High-level vulnerabilities are difficult to exploit; however, they also have significant impact on smart contract execution, e.g. public access to crucial functions
Medium	Medium-level vulnerabilities are important to fix; however, they can't lead to the token loss
Low	Low-level vulnerabilities are mostly related to outdated, unused etc. code snippets, that can't have significant impact on execution
Lowest / Code Style / Info	Lowest-level vulnerabilities, code style violations and informational statements can't affect execution of the smart-contract and can be ignored

AS-IS Overview

Intended Functionality

The submitted code implements two reward pools with staking functionality. They only have initialization in the constructor and inherit all of the logic of a single contract, which implements staking and expanded with two additional contracts

UniEDDARewardsPool

This contract only has a constructor with parameters for RewardsPool contract initialization that inherits this contract

EDDARewardsPool

The same as for UniEDDARewardsPool

RewardsPool

The core contract of the reward pools is inherited from LPTokenWrapper and IRewardDistributionRecipient contracts. The RewardPool contract contains logic for staking. Users can make a few actions:

- Stake
- Withdraw
- Exit
- Get Reward

Staking has predefined parameters in the contract:

- Duration for reward calculation – seven (7) days
- Staking duration – there (3) days
- Minimum staking duration – two (2) minutes

$$\frac{Balance * (Reward\ per\ Token\ -\ Paid\ Reward\ per\ Token)}{1e18} + Rewards$$

For each account, the earning is calculating by the formula.

Balance is the number of tokens on the account in the staking contract.

Rewards represents the account earned amount. Rewards updates lives in the updateReward modifier. Due to the smart contract nature, this means that only an externally owned account can initially call function and no way to update its state automatically. Which is why updateReward assigned to all non-view functions and is activated every time it's called. Also, this modifier updates Reward Per Token Stored and Paid Reward per Token.

$$Reward\ per\ Token\ Stored\ +\ \frac{(Last\ Time\ Reward\ Applicable\ -\ Last\ Update\ Time) * Reward\ Rate * 1e18}{Total\ Supply}$$

Reward per Token is calculated by the formula:

In addition, the Reward Pool contract has a function to notify its reward amount. When the caller is a distributor of the award.

All formulas were taken from the code itslef, any additional specification was not provided. We cannot be responsible for correctness of the project business logic and its staking limitation. Also, we cannot predict interaction with other protocols and unknown smart contracts.

LPTokenWrapper

This contract provides ERC20-like function for staking its state storing. It allows to stake and withdraw assets. Each of these functions update the total supply and the account balance.

IRewardDistributionRecipient

This contract represents Ownable contract and provides only a reward distribution address update as the owner.

Smart Contracts Audited

The following files have been covered in the audit process:

contracts
├── EDDARewardsPool.sol
├── UniEDDARewardsPool.sol
├── RewardsPool.sol
├── LPTokenWrapper.sol

interfaces
├── IRewardDistributionRecipient.sol

openzeppelin contracts
├── Context.sol
├── IERC20.sol
├── Math.sol
├── SafeMath.sol
├── Ownable.sol
├── Address.sol
├── SafeERC20.sol

Smart Contracts Functions Audited

Custom contracts

+ Contract EDDARewardsPool
    - constructor() - Rewards Pool initialization with parameters: address lp_, IERC20 outToken_, address rewardDistribution_, uint256 stakeDurationSeconds_

+ Contract UniEDDARewardsPool
    - constructor() - Rewards Pool initialization with parameters: address lp_, IERC20 outToken_, address rewardDistribution_, uint256 stakeDurationSeconds_

+ Contract RewardsPool

- lastTimeRewardApplicable() - view function, shows minimum between current block timestamp and period finish

- rewardPerToken() - view function, shows result of formula for reward per token calculation

- earned(address) - view function, shows result of formula for earning calculation by address
- stake(address) - allows to stake a certain amount of tokens
- withdraw(address) - allows to withdraw a certain amount of tokens, not earlier than minimum staking duration
- exit() - allows to withdraw stake and getting reward
- getReward() - allows to get reward, which calculated by earned formula
- notifyRewardAmount() - allows only reward distribution address to update reward rate, period finish and last update time
- lockedDetails() - view function, shows locked details with period finish
- voteLock(address) - view function, shows in what time address can withdraw token

+ From LPTokenWrapper
    - totalSupply() - view function, shows total amount of staked tokens
    - balanceOf(address) - view function, shows balance of account
    - stake(uint256) - allows to stake, update total supply, balances and charge tokens by transfer from
    - withdraw(uint256) - allows to withdraw any available stake amount: updates total supply, balances and transfer tokens

+ From IRewardDistributionRecipient
    - setRewardDistribution(address) - allows to set address as owner, which called reward distribution

OpenZeppelin contracts and libraries

+ Contract Address (Most derived contract)
    - isContract(address) (internal)
    - toPayable(address) (internal)
    - sendValue(address payable, uint256) (internal)

+ Contract SafeMath (Most derived contract)
    - add(uint256, uint256) (internal)
    - div(uint256, uint256) (internal)
    - mul(uint256, uint256) (internal)
    - sub(uint256, uint256) (internal)

+ Contract Ownable (Most derived contract)
    - owner() public
    - isOwner() public
    - renounceOwnership() public

- transferOwnership(address) public
- _transferOwnership(address) internal

+ Contract Context (Most derived contract)
- _msgSender() (internal)
- _msgData() (internal)

+ Library SafeERC20
- safeTransfer(IERC20, address, uint256) internal
- safeTransferFrom(IERC20, address, address, uint256) internal
- safeApprove(IERC20, address, uint256) internal
- safeIncreaseAllowance(IERC20, address, uint256) internal
- safeDecreaseAllowance(IERC20, address, uint256) internal
- callOptionalReturn(IERC20, bytes memory) private

Audit Overview

Critical

No critical vulnerabilities were found.

High

No high vulnerabilities were found.

Medium

No medium vulnerabilities were found.

Low

No low vulnerabilities were found.

Concerns

1. Prevent reentrancy by adding Reentrancy Guard to all public and external functions. This contract module helps prevent reentrant calls to a function. Inheriting from ReentrancyGuardl makes the nonReentrant modifier available, which can be applied to functions and make sure there are no nested (reentrant) calls to these.

2. If we try to build an atomic transaction which will already contains two calls:

stake -> withdraw,

then its withdraw is secured by canWithdrawTime check in guard.

But what if we try to build an atomic transaction with

stake -> getReward,

then is there anything preventing getting a reward immediatley? Please, check it carefully depending on the formula, because in connection with flash loans it could bring a lot of issues.

Conclusion

The smart contracts within the scope were manually reviewed and analyzed with static analysis tools. For the contract, high level description of functionality was presented in the “As-Is” overview section of the report.

The audit report contains all found security vulnerabilities and other issues in the reviewed code.

Project has tests for the business logic and blockchain interaction. Tests covered each contract and described different use cases. Contract deployment implemented with test contract for flexibility.

Most of all critical functions covered by tests.

The overall quality of the code submitted for the audit is impressive.

Best practice recommendations have mainly been followed. Existing audited code has been used whenever possible in the form of the OpenZeppelin libraries (https://openzeppelin.com/)

A safe math library has been used for arithmetic operations to avoid overflow and underflow issues. Code layout mostly follows the official Solidity style guide

https://docs.soliditylang.org/en/v0.6.6/style-guide.html

Disclaimer

Hackcontrol Disclaimer

The smart contracts given for the audit have been analyzed according to the best industry practices on the date of this report issuance in relation to: cybersecurity vulnerabilities and issues in the smart contract source code, details of which are disclosed in this report.

The audit makes no statements or warranties on security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bug-free status or any other statements of the contract. While we have done our best conducting the analysis and producing this report, it is important to note that you should not rely on this report only - we recommend proceeding with several independent audits and a public bug bounty program to ensure security of the smart contracts.

Technical Disclaimer

Smart contracts are deployed and executed on a block chain platform. The platform, its programming language, and other software related to the smart contract can have their own vulnerabilities leading to hacks. Thus, the audit can’t guarantee an explicit 100% security of the smart contracts.