Definitex Staking

Code Security Assessment

PREPARED BY:

THE AUDIT INSTITUTE ANALYST TEAM

PREPARED FOR:

DEFINITEX STAKING CONTRACT

PREPARED ON:

FEBRUARY 27TH 2021

THE AUDIT INSTITUTE

Report Version 1.0

Table of Contents

Disclaimer

The Audit Institute Reports are not, nor should be considered, an “endorsement” or “disapproval” of any particular project or team. These reports are not, nor should be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts The Audit Institute to perform a security review.

The Audit Institute Reports do not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technology’s proprietors, business, business model or legal compliance.

The Audit Institute Reports should not be used in any way to make decisions around investment or involvement with any particular project. These reports in no way provide investment advice, nor should be leveraged as investment advice of any sort.

The Audit Institute Reports represent an extensive auditing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

𝓑lockchain technology and cryptographic assets present a high level of ongoing risk. The Audit Institute's position is that each company and individual are responsible for their own due diligence and continuous security. The Audit Institute's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze. View our full legal terms and conditions at https://audit.institute/

What is included in a report by *The Audit Institute*?

- A document describing the detailed analysis of a particular piece(s) of source code provided to The Audit Institute by a Client.
- An organized collection of testing results, analysis and inferences made about the structure, implementation and overall best practices of a particular piece of source code.
- Representation that a Client of The Audit Institute has indeed completed a round of auditing with the intention to increase the quality of the company/product's IT infrastructure and or source code.

Overview

Project Summary

Project Name & Website	**Definitex** - **Https://definitex.org**
Project Description	The Audit Institute team reviewed a piece of code for the staking contract of the Definitex platform. The goal of the staking aspect is to reward users with a hardcoded percentage of interest for staking the token. This is designed to encourage users to gain passive value while holding the token.
Platform	Ethereum, Solidity
Compiler Version	^0.6.6
Mainnet Address	Staking Contract - Not Yet Deployed
GitHub Commit Hash	05eaad359d7540100c330e73b1521dc4dd5fd5dc
Delivery Date	February 27th 2021
Method of Audit	Static Analysis, Fuzzing, Manual Review
Consultants Engaged	2

Summary of Findings

Critical	2
Medium	0
Informational	9
Total Issues	**11**

Executive Summary

This audit report exclusively covers the analysis that was conducted on Definitex's staking contract written in Solidity. The Audit Institute analysts did not review the Definitex token contract or any other platform code as it was not provided.

The platform intends to offer a hardcoded interest rate of 0.1% per day. The users can utilize the withdraw functions to claim their rewards, or to exit their staking position. The platform intends to use a hardcoded fee of 0.5% for each withdraw.

After reviewing the code and conducting tests on the StakingDefinitex.sol contract, the analyst team discovered some notable discrepancies between their claimed intentions versus the actual values provided within the code.

Disclosed in the report below is a full analytical review of the platform after undergoing various test scenarios and code review. The findings varied in criticality as some were related to Solidity code standards and optimization, while others put users at risk of losing their funds. In its current state, the Definitex staking contract provides no guarantee for users to receive their original investment or interest accrued on said investment.

Contracts in Scope

CONTRACT NAME	CONTRACT DESCRIPTION
StakingDefinitex.sol	The Definitex Staking Contract

External Vulnerability Findings

Vulnerability Category	Notes	Results
Arbitrary Storage Write	N/A	PASS
Arbitrary Jump	N/A	PASS
Delegate Call to Untrusted Contract	N/A	PASS
Dependence on Predictable Variables	N/A	PASS
Deprecated Opcodes	N/A	PASS
Ether / Token Loss	See Critical Findings	FAIL*
Exceptions	N/A	PASS
External Calls	N/A	PASS
External Service Providers	N/A	PASS
Flash Loans	N/A	PASS
Inconsistent Emission of Events	N/A	PASS
Integer Over/Underflow	N/A	PASS
Multiple Sends	N/A	PASS
Oracles	N/A	PASS
Reentrancy Issues	N/A	PASS
Unchecked Retval	N/A	PASS
Suicide	N/A	PASS
State Change External Calls	N/A	PASS
Unchecked Retval	N/A	PASS

Findings and Recommendations

Finding Name	Criticality	Analyst Notes
The calculateReward function is off by a factor of 1000.	Critical	The calculation that was provided does not offer an interest rate of 0.1%, but rather gives the users a 100% return each day. We recommend the team sets _interestPerDay to 1, or by setting the divisor in the calculation to 86400000000. *See Issue#1 in Detailed Findings*
Token loss via reward in the withdraw function.	Critical	The withdraw function provides rewards from the contract balance. If the owner fails to provide the contract balance with enough tokens to cover the rewards owed to the user, the user will be unable to retrieve their rewards, or their reward will be funded by other user's staked tokens. The reward payments for the first users to deposit will be pulled from the subsequent users in the staking contract. (This can cause users to be unable to retrieve all of their staked tokens). *See Issue#2 in Detailed Findings*
The withdraw function uses the Hex version of the Max uint256.	Informational	The code currently works with this Hex version of the MAX uint256, However, it is hard to be sure with this method you actually provided all 64 Fs, and is fairly common to see people using a string with too few. For safety, it is considered a better practice to instead use the ~unint256(0).
Unnecessary check to see if the withdraw amount is greater than the token balance. (Gas Optimization)	Informational	The withdraw function checks to see if the amount > token_balance However, this will never be true since the amount will always be less than or equal to the amount being staked. This also applies to the withdrawStake function.
Unnecessary variable declaration and maintenance in the withdraw function. (Gas Optimization)	Informational	There is no need for the code in Line #138: uint256 token_balance = _stakingToken.balanceOf(address(this)); There is no need for the code in Line #152. token_balance = token_balance.sub(withdraw_amount) These are local changes and are not used to update the actual token balance. The token balance is updated on the next line, (#153) by calling the .transfer() function.
Duplicate statements to set same values (Gas Optimization)	Informational	The stake function sets the: stakers[msg.sender].stakeTime = now more than once when the 'if' statement is satisfied.

Multiple calls to transfer() within the same function. (Gas Optimization)	Informational	There are two calls to transfer in the withdraw function. One call is for the staking amount, and another call for the reward amount. These two calls can be combined into one call to the transfer function, resulting in less gas being used.
A require statement can replace the if statement in the withdrawStake function. (Gas Optimization)	Informational	The if statement in the withdrawStake function is not efficient on gas. *Recommendation:* In order to save on gas, the team can change the if statement to a require statement and move it to the top of the function. (Staking.withdrawStake()#184)
Unnecessary check to determine if reward>0 (Gas Optimization)	Informational	The statement: If (reward>0) (Staking.withdraw()#160) (Staking.withdrawRewards()#204) This is not needed in the code because the only time this statement will ever be true is if the user decides to stake and call rewards in the same block.
Functions should be `external` (Gas Optimization)	Informational	The following functions that should be set to external: Staking.stake(uint256) (Line#107-128) Staking.withdraw() (Line#130-131) Staking.withdrawStake() (Line#168-169) Staking.withdrawReward() (Line#195-210) **Recommendation: set these variables as constant to slightly reduce gas cost.*
Variables should be `constant` (Gas Optimization)	Informational	The following variables should be set to constant: Staking._fee (Line#80) Staking._interestPerDay (Line#79) **Recommendation: set these variables as constant to slightly reduce gas cost.*

Detailed Findings

Issue #1

Description of Issue:
The calculateReward function is off by a factor of 1000. The calculation that was provided does not offer an interest rate of 0.1%, but rather gives the users a 100% return each day.

Risk/Impact if Exploited:
This issue can cause the entire platform to be unsustainable. There will most likely not be enough tokens in circulation to provide users the ability to claim their rewards or withdraw their stake.

Recommendations:
Our recommendation is to modify the function by changing the _interestPerDay from 1000 to 1.
OR
Change the divisor in the calculation from 86400000 to 86400000000.

Supporting documentation:
Given:
_interestPerDay = 1000
now = 10000000
Staker
{
  stake: 100
  stakeTime: 9913600
  reward: 0
}

And the reward calculation:

```
reward = now.sub(stakers[staker].stakeTime)
                .mul(stakers[staker].stake)
                .mul(_interestPerDay)
                .div(86400000);
```

When we subsitute the values in for the variables:
reward = (10000000).sub(9913600).mul(100).mul(1000).div(86400000)

And simplify in order of execution:
= (86400).mul(100).mul(1000).div(86400000)
= (8640000).mul(1000).div(86400000)
= (8640000000).div(86400000)
reward = 100

The reward is equal to the stake amount, so the reward must be 100% of the stake amount. Therefore, the interest per day is 100%.

Issue #2

Description of Issue:
The user is susceptible to token loss because the withdraw function provides rewards from the contract balance. The problem is that the contract states the interest rate arbitrarily, rather than being segmented based upon the use of the token and the pool of users.

Risk/Impact if Exploited:
If the owner fails to provide the contract balance with enough tokens to cover the rewards owed to the user, the user will be unable to retrieve their rewards, or their reward will be funded by other user's staked tokens. The reward payments for the first users to deposit will be pulled from the subsequent users in the staking contract. (This can cause users to be unable to retrieve all of their staked tokens).

Recommendations:
When rates are set arbitrarily, it is advisable to have separate variables and a function that calculates the reward percentage (based on the number of tokens that are truly available for rewards) and guarantee the ability to pay out those rewards.

Functions Overview

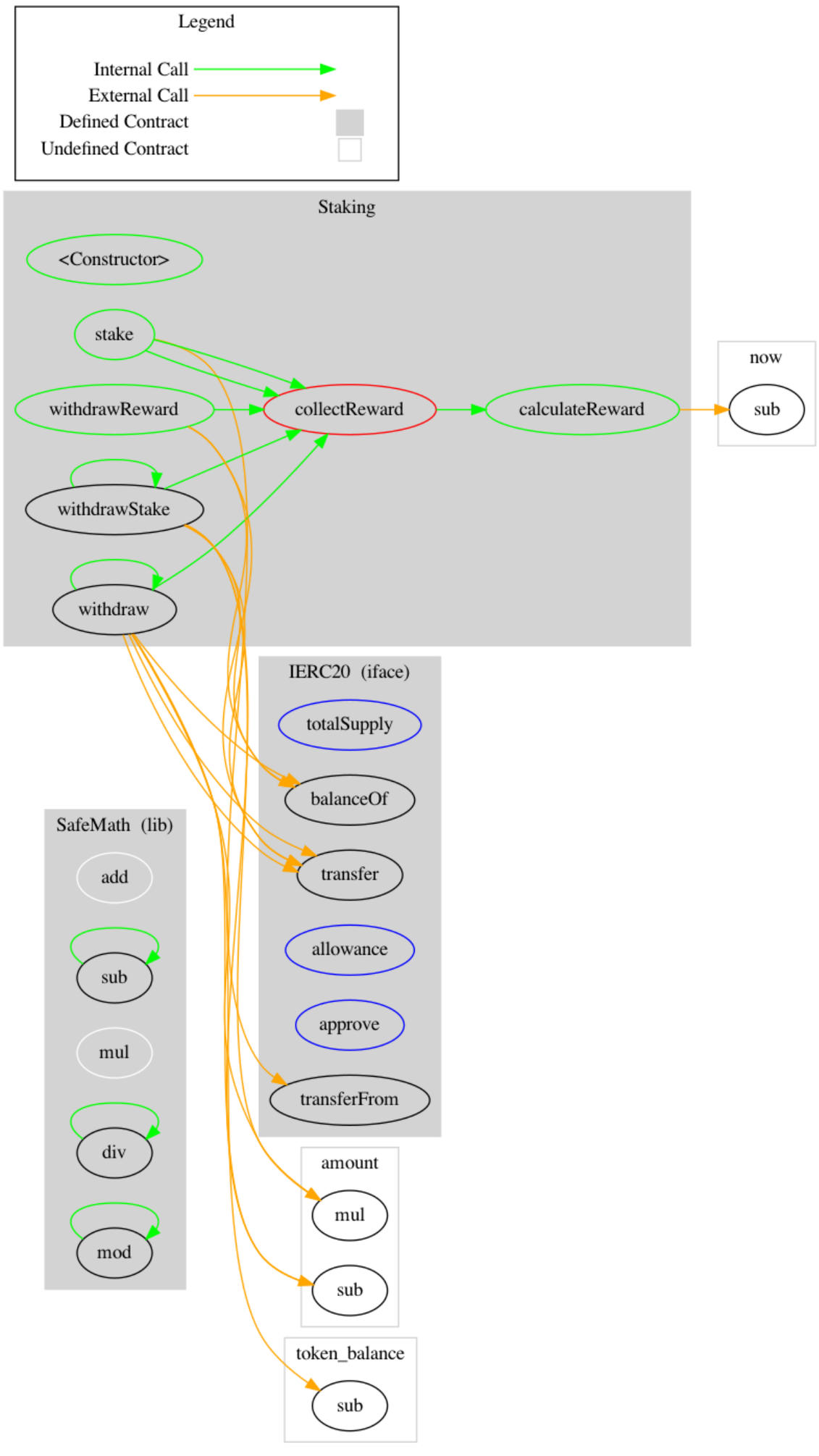
($) = payable function
 # = non-constant function
Int = Internal
Ext = External
Pub = Public

+ [Int] IERC20
- [Ext] totalSupply
- [Ext] balanceOf
- [Ext] transfer #
- [Ext] allowance
- [Ext] approve #
- [Ext] transferFrom #

+ [Lib] SafeMath
- [Int] add
- [Int] sub
- [Int] sub
- [Int] mul
- [Int] div
- [Int] div
- [Int] mod
- [Int] mod

+ Staking
- [Pub] <Constructor> #
- [Pub] calculateReward
- [Prv] collectReward #
- [Pub] stake #
- [Pub] withdraw #
- [Pub] withdraw #
- [Pub] withdrawStake #
- [Pub] withdrawStake #
- [Pub] withdrawReward #

Control Flow



END OF REPORT