DRIVENsecurity

TECHNICAL AUDIT

TCG Staking Contracts

April 19, 2022

Table of Contents

Project Details

Project Name
TCGstaking

Project Type
MasterChef // Locked Staking (Variable APR)

Contract Address
Not Yet Deployed

Blockchain
Not Yet Deployed

This Audit Was Created On
April 19, 2022

Technical Audit Details: Locked with Variable APR

INHERITANCE

Ownable
Openzeppeling Smart Contract for managing the ownership of a smart contract

ReentrancyGuard
Openzeppeling Smart Contract for blocking re-entrancy attacks

VARIABLES

- address rewardToken: the token that is deposited
- address stakingToken : the token used for rewards
- address treasuryAddress : the address that will receive eth or tokens deposited on the smart contract when “transferTokens()” or “withdrawEth()” functions are called
- bool stakingEnabled
- uint256 stakeLockPeriod
- uint256 claimLockPeriod
- uint256 totalReward
- uint256 rewardPerBlock
- uint256 lastRewardBlock
- uint256 totalStaked
- uint256 accTokensPerShare
- uint256 maxPerWallet
- mapping address=>UserInfo UserInfos
- uint256 startBlock
- uint256 blockRewardUpdateCycle
- uint256 blockRewardLastUpdateTime
- uint256 blocksPerDay
- uint256 blockRewardPercentage

TECHNICAL AUDIT DETAILS (CONTINUED)

FUNCTIONS CALLED BY CONTRACT

- updateRewardVariables()

PUBLIC FUNCTIONS

- deposit()
- withdraw()
- emergencyWithdraw()

GETTERS

- getRewardPerBlock()
- getRewardPerBlockUpdateTime()
- pendingRewards()
- getRewardToken1APY()
- getRewardToken1WPY()
- getStakeLockTime()
- getClaimLockTime()

INTERNAL FUNCTIONS

- tokenTransfer()

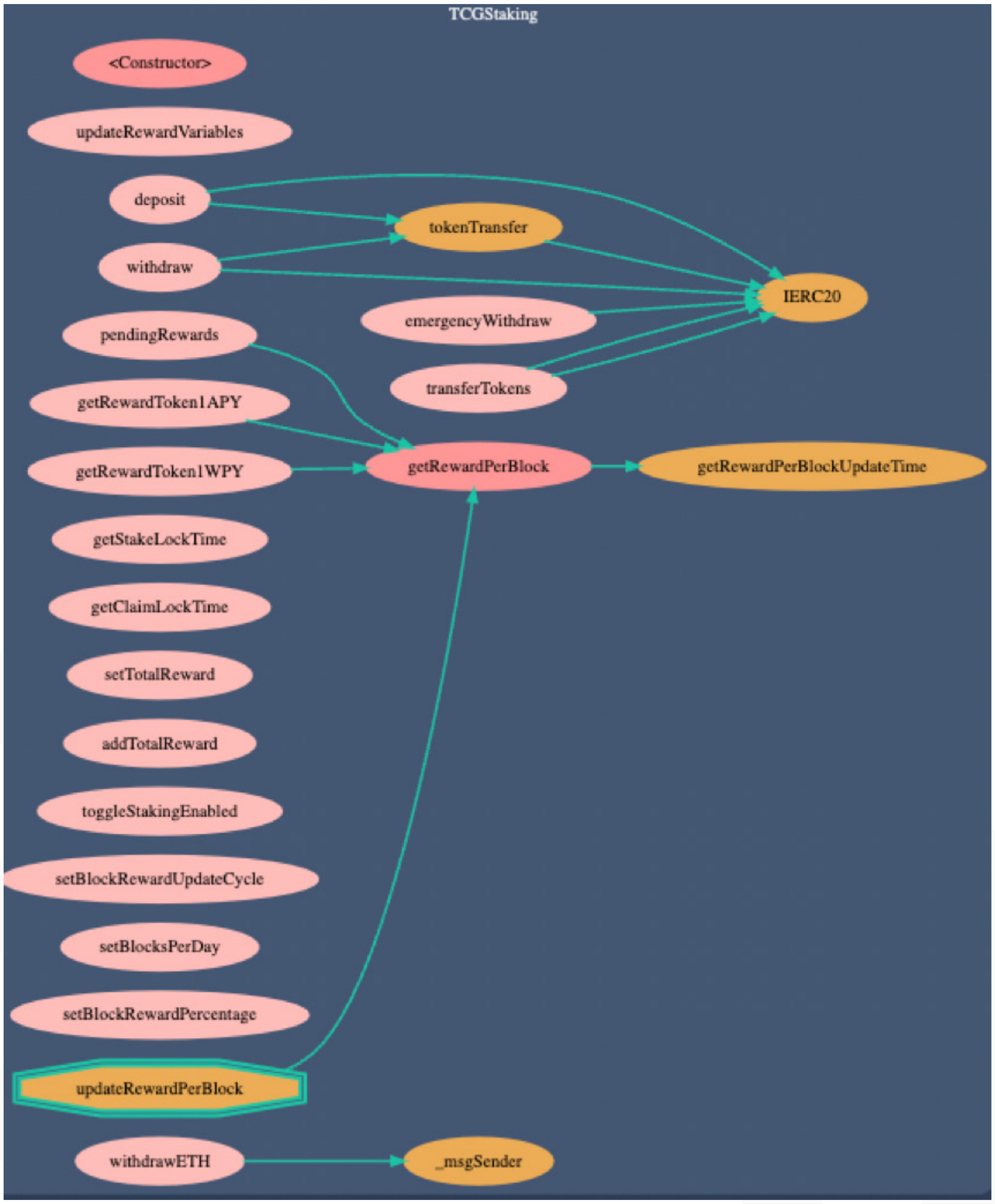
EXTERNAL ONLYOWNER FUNCTIONS

- setTotalReward()
- addTotalReward()
- toggleStakingEnabled()
- setBlockRewardUpdateCycle()
- setBlocksPerDay()
- setBlockRewardPercentage()

EXTERNAL FUNCTIONS

- transferTokens() - transfer ERC20 tokens from the smart contract
- withdrawETH() - transfer ETH from the smart contract

Smart Contract Architecture: Locked with Variable APR



CONCLUSION

Architecture: A best practice would be to make the “withdrawETH()” and “transferTokens()” functions callable only by the owner.

We recommend as well to:

- Use SafeERC20 library for ERC20 transfers (in/out);
- Use a simple transfer for “withdrawETH()” function:
  uint256 amount = address(this).balance;
  payable(treasuryAddress).transfer(amount)

Technical Audit Overview: Locked with Variable APR

GENERAL ISSUES

- Security Issues: **PASSED**
- Gas & Fees Issues: **PASSED**
- ERC Errors: **PASSED**
- Compilation Errors: **PASSED**
- Design Logic: **PASSED**
- Timestamp Dependence: **PASSED**

SECURITY AGAINST CYBER ATTACKS

- Private user's data: **PASSED**
- Reentrancy: **PASSED**
- Cross-Function Reentrancy: **PASSED**
- Front Running: **PASSED**
- Taxonomy Attacks: **PASSED**
- Integer Overflow and Underflow: **PASSED**
- DoS (Denial of Service) with Unexpected Revert: **PASSED**
- DoS (Denial of Service) with Block Gas Limit: **PASSED**
- Insufficient Gas Griefing: **PASSED**
- Forcibly Sending ETH to a Contract: **PASSED**

Penetration Testing: Locked with Variable APR

RE-ENTRANCY

What is Re-Entrancy?
A re-entrancy attack can arise when you write a function that calls another untrusted contract before resolving any consequences. If the attacker has authority over the untrusted contract, he can initial a recursive call back to the original function, repeating interactions that would otherwise not have occurred after the effects were resolved.

Attackers can take over the smart contract's control flow and make modifications to the data that the calling function was not anticipating.

To avoid this, make sure that you do not call an external function until the contract has completed all of the internal work.

TEST: PASSED

CROSS-FUNCTION RE-ENTRANCY

What is Cross-Function Re-Entrancy?
When a vulnerable function shares the state with another function that has a beneficial effect on the attacker, this cross-function re-entrancy attack is achievable. This re-entrancy issue that is the employment of intermediate functions to trigger the fallback function and a re-entrancy attack is not unusual.

Attackers can gain control of a smart contract by calling public functions that use the same state/variables as "private" or "onlyOwner" functions.

To avoid this, make sure there are no public functions that use private variables, and avoid calling routines that call external functions or use mutex (mutual exclusion).

TEST: PASSED

FRONT RUNNING

What is Front Running?
Front running indicates that someone can obtain prior information of transactions from other beneficial owners by technology or market advantage, allowing them to influence the price ahead of time and result in economic benefit, which usually results in loss or expense to others

Since all transactions are visible in the block explorer for a short period of time before they are executed, network observers can see and react to an action before it is included in a block.

Attackers can front run transactions because every transaction is visible to the blockchain, even if it is in the "processing" or "indexing" state. This is a very low security vulnerability because it is based on the blockchain rather than the contract.

The only possible attack is seeing transactions made by bots. Using transaction fees, you can avoid bots.

TEST: PASSED

PENETRATION TESTING (CONTINUED)

TAXONOMY ATTACKS

These taxation attacks can be made in 3 ways:

1. Displacement
Performed by increasing the gasPrice higher than network average, often by a multiplier of 10.

2. Insertion
Outbidding transaction in the gas price auction

3. Suppression (Block Stuffing)
The attacker sent multiple transactions with a high gasPrice and gasLimit to custom smart contracts that assert to consume all the gas and fill up the block's gasLimit.

This type of attack occurs mainly for exchanges, so this smart contract is secured

TEST: PASSED

INTEGER OVERFLOW AND UNDERFLOW

Overflow
An overflow occurs when a number gets incremented above its maximum value.

In the audited contract: uint8 private _decimals = 9;

(_decimals can't reach a value bigger than it's limit)

TEST: PASSED

Underflow

An overflow occurs when a number gets decremented below its maximum value.

(There are no decrementation functions for parameters and users can't call functions that are using uint values);

TEST: PASSED

PENETRATION TESTING (CONTINUED)

DOS (DENIAL OF SERVICE) WITH UNEXPECTED REVERT

DoS (Denial of Service) attacks can occur in functions when you attempt to transmit funds to a user and the functionality is dependent on the successful transfer of funds.

This can be troublesome if the funds are given to a bad actor's smart contract (when they call functions like "Redeem" or "Claim"), since they can simply write a fallback function that reverts all payments.

TEST: PASSED

DOS (DENIAL OF SERVICE) WITH BLOCK GAS LIMIT

Each block has an upper bound on the amount of gas that can be spent, and thus the amount of computation that can be done. This is the Block Gas Limit. If the gas spent exceeds this limit, the transaction will fail. This leads to a couple possible Denial of Service vectors.

TEST: PASSED

INSUFFICIENT GAS GRIEFING

This attack can be carried out against contracts that accept data and use it in a sub-call on another contract.

This approach is frequently employed in multisignature wallets and transaction relayers. If the sub-call fails, either the entire transaction is rolled back or execution is resumed.

TEST: PASSED.

FORCIBLY SENDING ETH TO THE SMART CONTRACT

TEST: PASSED

Project Details

Project Name
TCGstaking

Project Type
MasterChef // Locked Staking (Fixed APR)

Contract Address
Not Yet Deployed

Blockchain
Not Yet Deployed

This Audit Was Created On
April 19, 2022

Technical Audit Details: Locked with Fixed APR

INHERITANCE

Ownable

Openzeppeling Smart Contract for managing the ownership of a smart contract

ReentrancyGuard

Openzeppeling Smart Contract for blocking re-entrancy attacks

VARIABLES

- address rewardToken
- address stakingToken
- address treasuryAddress
- bool stakingEnabled
- uint256 stakeLockPeriod
- uint256 claimLockPeriod
- uint256 apr
- uint256 maxStake
- uint256 maxStakePlusOne
- uint256 startBlock
- uint256 totalReward
- mapping address=>UserInfo UserInfos
- uint256 rewardPerMinutePerToken
- uint256 totalStaked

TECHNICAL AUDIT DETAILS (CONTINUED)

PUBLIC FUNCTIONS

- deposit()
- withdraw()
- emergencyWithdraw()

GETTERS

- getPending()
- getStakeLockTime()
- getClaimLockTime()
- pendingRewards()

INTERNAL FUNCTIONS

- tokenTransfer()

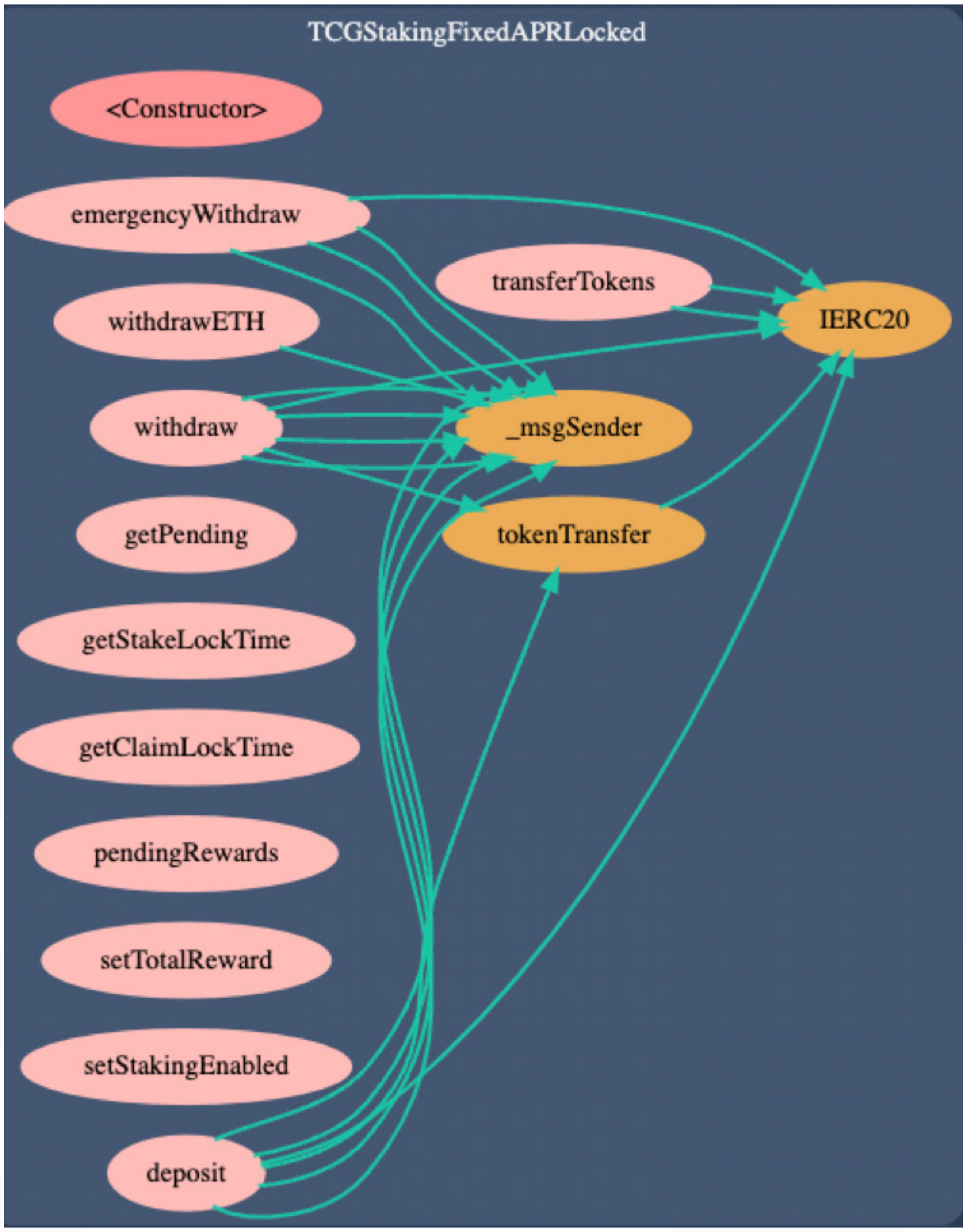
EXTERNAL ONLYOWNER FUNCTIONS

- setTotalReward()
- setStakingEnabled()

EXTERNAL FUNCTIONS

- transferTokens() - transfer ERC20 tokens from the smart contract
- withdrawETH() - transfer ETH from the smart contract

Smart Contract Architecture: Locked with Fixed APR



CONCLUSION

Architecture: A best practice would be to make the “withdrawETH()” and “transferTokens()” functions callable only by the owner..

We recommend as well to:

- Use a function that allows the owner of the smart contract to change the FIXED APR
- Use SafeERC20 library for ERC20 transfers (in/out);
- Use SafeMath for uint256 mathematical operations;
- Use a simple transfer for “withdrawETH()” function:
  uint256 amount = address(this).balance;
  payable(treasuryAddress).transfer(amount)

Technical Audit Overview: Locked with Fixed APR

GENERAL ISSUES

- Security Issues: **PASSED**
- Gas & Fees Issues: **PASSED**
- ERC Errors: **PASSED**
- Compilation Errors: **PASSED**
- Design Logic: **PASSED**
- Timestamp Dependence: **PASSED**

SECURITY AGAINST CYBER ATTACKS

- Private user's data: **PASSED**
- Reentrancy: **PASSED**
- Cross-Function Reentrancy: **PASSED**
- Front Running: **PASSED**
- Taxonomy Attacks: **PASSED**
- Integer Overflow and Underflow: **PASSED**
- DoS (Denial of Service) with Unexpected Revert: **PASSED**
- DoS (Denial of Service) with Block Gas Limit: **PASSED**
- Insufficient Gas Griefing: **PASSED**
- Forcibly Sending ETH to a Contract: **PASSED**

Penetration Testing: Locked with Fixed APR

RE-ENTRANCY

What is Re-Entrancy?
A re-entrancy attack can arise when you write a function that calls another untrusted contract before resolving any consequences. If the attacker has authority over the untrusted contract, he can initial a recursive call back to the original function, repeating interactions that would otherwise not have occurred after the effects were resolved.

Attackers can take over the smart contract's control flow and make modifications to the data that the calling function was not anticipating.

To avoid this, make sure that you do not call an external function until the contract has completed all of the internal work.

TEST: PASSED

CROSS-FUNCTION RE-ENTRANCY

What is Cross-Function Re-Entrancy?
When a vulnerable function shares the state with another function that has a beneficial effect on the attacker, this cross-function re-entrancy attack is achievable. This re-entrancy issue that is the employment of intermediate functions to trigger the fallback function and a re-entrancy attack is not unusual.

Attackers can gain control of a smart contract by calling public functions that use the same state/variables as "private" or "onlyOwner" functions.

To avoid this, make sure there are no public functions that use private variables, and avoid calling routines that call external functions or use mutex (mutual exclusion).

TEST: PASSED

FRONT RUNNING

What is Front Running?
Front running indicates that someone can obtain prior information of transactions from other beneficial owners by technology or market advantage, allowing them to influence the price ahead of time and result in economic benefit, which usually results in loss or expense to others

Since all transactions are visible in the block explorer for a short period of time before they are executed, network observers can see and react to an action before it is included in a block.

Attackers can front run transactions because every transaction is visible to the blockchain, even if it is in the "processing" or "indexing" state. This is a very low security vulnerability because it is based on the blockchain rather than the contract.

The only possible attack is seeing transactions made by bots. Using transaction fees, you can avoid bots.

TEST: PASSED

PENETRATION TESTING (CONTINUED)

TAXONOMY ATTACKS

These taxation attacks can be made in 3 ways:

1. Displacement
Performed by increasing the gasPrice higher than network average, often by a multiplier of 10.

2. Insertion
Outbidding transaction in the gas price auction

3. Suppression (Block Stuffing)
The attacker sent multiple transactions with a high gasPrice and gasLimit to custom smart contracts that assert to consume all the gas and fill up the block's gasLimit.

This type of attack occurs mainly for exchanges, so this smart contract is secured

TEST: PASSED

INTEGER OVERFLOW AND UNDERFLOW

Overflow
An overflow occurs when a number gets incremented above its maximum value.

In the audited contract: uint8 private _decimals = 9;

(_decimals can't reach a value bigger than it's limit)

TEST: PASSED

Underflow

An overflow occurs when a number gets decremented below its maximum value.

(There are no decrementation functions for parameters and users can't call functions that are using uint values);

TEST: PASSED

PENETRATION TESTING (CONTINUED)

DOS (DENIAL OF SERVICE) WITH UNEXPECTED REVERT

DoS (Denial of Service) attacks can occur in functions when you attempt to transmit funds to a user and the functionality is dependent on the successful transfer of funds.

This can be troublesome if the funds are given to a bad actor's smart contract (when they call functions like "Redeem" or "Claim"), since they can simply write a fallback function that reverts all payments.

TEST: PASSED

DOS (DENIAL OF SERVICE) WITH BLOCK GAS LIMIT

Each block has an upper bound on the amount of gas that can be spent, and thus the amount of computation that can be done. This is the Block Gas Limit. If the gas spent exceeds this limit, the transaction will fail. This leads to a couple possible Denial of Service vectors.

TEST: PASSED

INSUFFICIENT GAS GRIEFING

This attack can be carried out against contracts that accept data and use it in a sub-call on another contract.

This approach is frequently employed in multisignature wallets and transaction relayers. If the sub-call fails, either the entire transaction is rolled back or execution is resumed.

TEST: PASSED.

FORCIBLY SENDING ETH TO THE SMART CONTRACT

TEST: PASSED