DRIVENsecurity

TECHNICAL AUDIT

JoinCoin Staking Contract

March 11, 2022

Table of Contents

Accepting a project audit can be viewed as a sign of confidence and is typically the first indicator of trust for a project, but it does not guarantee that a team will not remove all liquidity, sell tokens, or engage in any other type of fraud. There is also no method to restrict private sale holders from selling their tokens. It is ultimately your obligation to read through all documentation, social media posts, and contract code for each particular project in order to draw your own conclusions and define your own risk tolerance.

DRIVENsecurity accepts no responsibility for any losses or encourages speculative investments. This audit's material is given solely for information reasons and should not be construed as investment advice.

Project Details

Project Name
JoinCoin

Project Type
Staking Pool

Contract Address
Smart Contract is not deployed

Blockchain
Smart contract is not deployed

Project Website
joincoin.io

This Audit Was Created On
March 11, 2022

Technical Audit Overview

GENERAL ISSUES

- Security issues: **PASSED**
- Gas & Fees issues: **PASSED**
- ERC errors: **PASSED**
- Compilation errors: **PASSED**
- Design logic: **PASSED**
- Timestamp dependence: **PASSED**
- Buy & sell: ***The owner can't enable/disable swapping for certain users***

SECURITY AGAINST CYBER ATTACKS

- Reentrancy: **SECURED**
- 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**

Technical Audit Details

FUNCTIONS FOR USERS

Deposit (Name of the function: deposit)

- Deposit tokens on the staking smart contract by user
- Protection: Re-entrancy Guard (using nonReentrant modifier);

Emergency Withdraw (Name of the function: emergencyWithdraw):

- Allow users to unstake the tokens before the expiration period (4 weeks); - The user will pay an extra fee for doing this action: penaltyFee;
- Protection: Re-entrancy Guard (using nonReentrant modifier);

Withdraw (Name of the function: withdraw):

- Allow users to unstake the tokens if the expiration period has passed; - The user will pay an extra fee for doing this action: penaltyFee;
- Protection: Re-entrancy Guard (using nonReentrant modifier);

FUNCTIONS FOR USERS

Emergency withdraw of rewards (Name of the function: emergencyRewardWithdraw)

- Allow the owner of the smart contract to withdraw the rewards pool;

Note: Instead of msg.sender argument from the last line of code on this function “rewardToken. safeTransfer(address(msg.sender), _amount);” use a hardcoded address. In this way, if somebody will break the contract, the only address where the reward pool will be sent is a team address.

Mass Update Pool (Name of the function: massUpdatePools):

- Update the rewards on the existing pool;

Renounce Ownership (Name of the function: renounceOwnership):

- Change the owner of the smart contract to the dead address;

Start The Rewards(Name of the function: startReward):

- Start the reward distribution;

Stop The Rewards(Name of the function: stopReward):

- Stop the reward distribution;

Transfer Ownership (Name of the function: transferOwnership):

- Change the owner of the smart contract to another address;

Update APY (Name of the function: updateApy):

- Update the APY for the staking pool. Can’t be higher than 1000;

Update The Penalty Fee (Name of the function: updateExitPenalty):

- Update the penalty fee for users that unstable their tokens earlier; - Can’t be higher than 20;

TECHNICAL AUDIT DETAILS (CONTINUED)

INTERNAL FUNCTIONS

Update The Pool (Name of the function: updatePool)

- Is used only buy the “massUpdatePools” function in order to update the rewards of the staking pool;

CONCLUSION AND RECOMMENDATIONS:

The logic of the smart contract is well-designed and is using Re-entrancy Guard by OpenZeppeling in order to stop the possible Re-Entrancy Attacks. There are no errors or hidden issues.

Penetration Testing

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

PENETRATION TESTING (CONTINUED)

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

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

PENETRATION TESTING (CONTINUED)

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

Conclusion

This contract uses the updated version of SafeMath for Solidity 0.8 and above that will prevent Integer Overflows and Underflows.

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.

There are no functions that deliver money to users (ACCEL Defi team is using 3rd party/external contracts in order to do that), so attackers are unable to communicate using a contract with fallBack functions.

TEST: PASSED

PENETRATION TESTING (CONTINUED)

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—USERS CAN'T EXECUTE SUB-CALLS.

FORCIBLY SENDING ETH TO THE SMART CONTRACT

TEST: PASSED

This audit was created by

Accepting a project audit can be viewed as a sign of confidence and is typically the first indicator of trust for a project, but it does not guarantee that a team will not remove all liquidity, sell tokens, or engage in any other type of fraud. There is also no method to restrict private sale holders from selling their tokens. It is ultimately your obligation to read through all documentation, social media posts, and contract code for each particular project in order to draw your own conclusions and define your own risk tolerance.

DRIVENsecurity accepts no responsibility for any losses or encourages speculative investments. This audit's material is given solely for information reasons and should not be construed as investment advice.