BUSDPIRATES.COM

SMART-CONTRACT AUDIT REPORT

BY SOLIDSECT

CONTRACT ADDRESS:
https://bscscan.com/address/0x97f8F6138D105c5B674E58777d66063c23f474B4#code

Website: https://busdpirates.com/

Telegram: https://t.me/bscpirates

Contents

Disclaimer

This audit makes no statements or warranties about the utility or safety of the code, the suitability or regulatory regime of the business model, or any other statements about the fitness of the contracts regarding their purpose or bug-free status. This audit documentation is for informative purposes only.

AUDIT SUMMARY

Concluding, the inspected code is well-written and performs as expected. There is no backdoor to steal funds.

Description of the Smart Contract Functionality:

The contract provides Hour-Glass like distribution mechanics for users who invest BUSD into the contract. Re-payment of the funds are made in real time as long as the contract has enough funds.

The system is similar to classic ROI Games with some sustainability features added and the daily return is soft-pegged around 3 percent.

These games are considered as High-Risk games because Dividends are paid from deposits of users. Therefore, users should always DYOR before participating.

Critical Vulnerabilities Found

=> No critical vulnerabilities found

Medium-Severity Vulnerabilities Found

=> No medium-severity vulnerabilities found

Low-Severity Vulnerabilities Found

=> No low-severity vulnerabilities found

IMPORTANT NOTES

CONTRACT MECHANICS

Pirate Price changes according to:

BUSD amount in contract

Mining Rate (marketEggs variable) of contract

Daily Payment Rate adjusts according to:

Mining Rate (marketEggs variable) of the contract

Each Deposit, Compound and Withdraw impacts:

Mining Rate (marketEggs variable) of the contract

FEES AND REWARDS

10% Referral Reward in BUSD (1-Tier) - Verified.

5% Developer's Fee - Verified.

Audit Overview

The project has 1 file. It contains approx. 171 lines of Solidity code. All functions and state variables are well-commented using natspec documentation, which that does not create any vulnerability.

Executed Attacks to the Contract

In order to check for the security of the contract, we tested several attacks in order to make sure that the contract is secure and follows best practices.

Over and Under Flows

An overflow happens when the limit of the type variable uint256, 2 ** 256, is exceeded. What happens is that the value resets to zero instead of incrementing more. Yet, an underflow happens when you try to subtract 0 minus a number larger than 0. For example, if you subtract 0 - 1 the result will be = 2 ** 256 instead of -1. This is quite dangerous. This contract does check for overflows and underflows by using Open Zeppelin's Safe Math to mitigate this attack, and all the functions have strong validations, which prevented this attack.

Short Address Attack

If the token contract has enough amount of tokens and the buy function does not check the length of the address of the sender, the BSC's virtual machine will add zeros to the transaction until the address is complete. Although this contract is not vulnerable to this attack, there are some points where users can mess themselves up due to this (please see below). It is highly recommended to call functions after checking validity of the address.

Visibility and 'Delegate Call' Misuse

This is also known as Parity Hack, which occurs during misuse of a Delegate Call. No such issues were found in this smart contract, and the visibility was also properly addressed. There are some places where there is no visibility defined. The smart contract will assume 'public' visibility if there is no visibility defined. It is good practice to explicitly define the visibility; but again, the contract is not prone to any vulnerability due to that in this case.

Reentrancy / TheDAO Hack

Reentrancy occurs in this case: Any interaction from a contract (A) with another contract (B) and any transfer of Tron hands over control to that contract (B). This makes it possible for B to call back into A before this interaction is completed. Use of 'require' function in this smart contract mitigates this vulnerability.

Forcing BNB to a Contract

While implementing 'selfdestruct' in a smart contract, it sends all the BNB to the target address. Now, if the target address is a contract address, then the fallback function of the target contract does not get called. And thus, a hacker can bypass the 'required' conditions. Here, the smart contract's balance has never been used as a guard, which mitigates this vulnerability.

END OF REPORT – 8 PAGES