BLOCK SOLUTIONS

Smart Contract Code Review and Security Analysis Report for Gentop Presale & Staking Smart Contracts

Request Date: 2024-11-28
Completion Date: 2024-11-28
Language: Solidity

Contents

Commission

Audited Project	Staking Smart Contract, Presale Smart Contract
Staking Smart Contract Address	0xc94cDB70F1ec91437C5d22340d5206B4B8928482
Presale Smart Contract Address	0xAb1e13E8A7a7d95EE8aDDC1f74aAc3CF6CccA597
Contract Owner	0xc3D4a3226ebD5801b4D23d4fa771d0A5B1f3dC06

Block Solutions was commissioned by STAKING Smart Contract owners to perform an audit of their main smart contract. The purpose of the audit was to achieve the following:

- Ensure that the smart contract functions as intended.
- Identify potential security issues with the smart contract.

The information in this report should be used to understand the risk exposure of the smart contract, and as a guide to improve the security posture of the smart contract by remediating the issues that were identified.

STAKING, PRESALE
Properties

Audited Project	Staking Smart Contract, Presale Smart Contract
Smart Contract Address	0xc94cDB70F1ec91437C5d22340d5206B4B8928482
Contract Owner	0xc3D4a3226ebD5801b4D23d4fa771d0A5B1f3dC06
Contract Deployer	0xc3D4a3226ebD5801b4D23d4fa771d0A5B1f3dC06
Genius People Token	0x4DF17Ed886b3237fDbc29EdB6e4dc986433f2377
BSC-USD Address	0x55d398326f99059fF775485246999027B3197955
Presale Address	0xAb1e13E8A7a7d95EE8aDDC1f74aAc3CF6CccA597
Presale Contract Details	
Presale Address	0xAb1e13E8A7a7d95EE8aDDC1f74aAc3CF6CccA597
Contract Owner	0xc3D4a3226ebD5801b4D23d4fa771d0A5B1f3dC06
Contract Deployer	0xc3D4a3226ebD5801b4D23d4fa771d0A5B1f3dC06
Sales Closing Time	GMT: Friday, December 27, 2024 9:45:27 AM
Genius People Token	0x4DF17Ed886b3237fDbc29EdB6e4dc986433f2377
BSC-USD Address	0x55d398326f99059fF775485246999027B3197955
Staking Smart Contract Address	0xc94cDB70F1ec91437C5d22340d5206B4B8928482

Contract Functions

Executables

i. function setPreSale(address __preSale) public
ii. function WithdrawReward(address userAddress, uint _num) external
iii. function withdrawAdmin()external
iv. function Buy(uint _usdtAmount, address _buyer, uint _runner) public
v. function withdrawAdmin()external
vi. function setPercentage(uint256 _percentage, uint _whichTypeOfPercentage) public

Checklist

Compiler errors.	Passed
Possible delays in data delivery.	Passed
Timestamp dependence.	Passed
Integer Overflow and Underflow.	Passed
Race Conditions and Reentrancy.	Passed
DoS with Revert.	Passed
DoS with block gas limit.	Passed
Methods execution permissions.	Passed
Economy model of the contract.	Passed
Private user data leaks.	Passed
Malicious Events Log.	Passed
Scoping and Declarations.	Passed
Uninitialized storage pointers.	Passed
Arithmetic accuracy.	Passed
Design Logic.	Passed
Impact of the exchange rate.	Passed
Oracle Calls.	Passed
Cross-function race conditions.	Passed

Fallback function security.	Passed
Safe Open Zeppelin contracts and implementation usage.	Passed
Whitepaper-Website-Contract correlation.	Passed
Front Running.	Not Checked

Executable Functions

STAKING Smart Contract

Owner of staking smart contract can execute this function to set the presale address into staking contract.

```
function setPreSale(address __preSale) public {
    require(msg.sender == _owner, "Only owner can run this function");
    _preSale = IpreSale(__preSale);
}
```

User of the staking contract can withdraw the rewards.
Requirements:

- User has not claimed reward already.
- User can withdraw only once staking period ends.

```
function WithdrawReward(address userAddress, uint _num) external  {
    require(userPurcahasesS[userAddress][_num] < 2, "you have already claimed reward");
    (,uint joinTime,,uint percentage,,) = _preSale.getUserData(userAddress, _num);
    // Check cumulative reward
    uint256 rewardAmount = checkCumulativeReward(userAddress, _num);
    // Determine staking period
    uint256 stakingPeriod;
    if (percentage == 24) {
        stakingPeriod = STAKING_120_DAYS;
    } else if (percentage == 36) {
        stakingPeriod = STAKING_90_DAYS;
    } else if (percentage == 45) {
        stakingPeriod = STAKING_45_DAYS;
    }
    // Ensure staking period is over before transferring the reward
    require(block.timestamp >= (joinTime + stakingPeriod),"Too Early to withdraw");
    // if (block.timestamp >= (joinTime + stakingPeriod)) {
        _GentTop.transfer(userAddress, rewardAmount);
        userPurcahasesS[userAddress][_num] = 2;
        // Update total rewards given
        totalRewardsGiven += rewardAmount;
        // Successful withdrawal after staking period
}
```

Owner of staking smart contract can withdraw all Gentop tokens which are stored in staking contract to own account.

```
function withdrawAdmin()external{
    require(msg.sender == _owner, "only owner can Run this function ");
    uint bal = _GentTop.balanceOf(address(this));
    _GentTop.transfer(_owner, bal);
}
```

Owner can set the percentage for BRONZE, SILVER, GOLD.

```
function setPercentage(uint256 _percentage, uint _whichTypeOfPercentage) public {
        require(msg.sender == owner, "only owner can run this function");
        if(_whichTypeOfPercentage == 1){
            _BronzePercentage = _percentage;
        }else if(_whichTypeOfPercentage == 2){
            _SilverPercentage = _percentage;
        }else if(_whichTypeOfPercentage == 3){
            _GoldPercentage = _percentage;
        }
    }
```

User can spend BSC-USD to buy from Presale.

```
function Buy(uint _usdtAmount, address _buyer, uint _runner) public  {
    require(_runner >= 1 || _runner <= 3, "Wrong number of runner");
    require(block.timestamp <= salesClosingTime ,"Sales Times up!");
    uint256 num = userPurcahases[_buyer];
    uint256 _price;
    uint256 percentage;
    uint256 sold;
    uint endtimee;
    if( _runner == 1){
        percentage = _BronzePercentage;
        _price = _BronzePrice;
        sold = _bronzeSold;
        endtimee = 120 * 86400;

    }else if(_runner == 2){
        percentage = _SilverPercentage;
        _price = _SilverPrice;
```

Owner of this address can withdraw Gen Top tokens and BSC-USD token from presale contract to own address.

```
function withdrawAdmin()external{
    require(msg.sender == owner, "only owner can Run this function");
    uint bal = _GentTop.balanceOf(address(this));
    uint balUsd = _USDT.balanceOf(address(this));
    _GentTop.transfer(owner, bal);
    _USDT.transfer(owner, balUsd);
}
```

Testing Summary

Quick Stats:

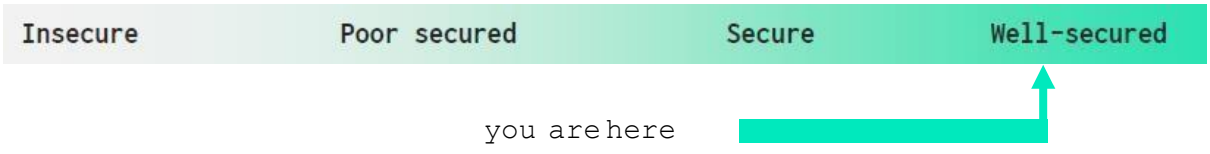
Main Category	Subcategory	Result
Contract Programming	Solidity version not specified	Passed
	Solidity version too old	Passed
	Integer overflow/underflow	Passed
	Function input parameters lack of check	Passed
	Function input parameters check bypass	Passed
	Function access control lacks management	Passed
	Critical operation lacks event log	Passed
	Human/contract checks bypass	Passed

	Random number generation/use vulnerability	Passed
	Fallback function misuse	Passed
	Race condition	Passed
	Logical vulnerability	Passed
	Other programming issues	Passed
Code Specification	Visibility not explicitly declared	Passed
	Var. storage location not explicitly declared	Passed
	Use keywords/functions to be deprecated	Passed
	Other code specification issues	Passed
Gas Optimization	Assert () misuse	Passed
	High consumption 'for/while' loop	Passed
	High consumption 'storage' storage	Passed
	"Out of Gas" Attack	Passed
Business Risk	The maximum limit for mintage not set	Passed
	"Short Address" Attack	Passed
	"Double Spend" Attack	Passed

Overall Audit Result: Passed

Executive Summary

According to the standard audit assessment, Customer`s solidity smart contract is **Well-Secured.** Again, it is recommended to perform an Extensive audit assessment to bring a more assured conclusion.



We used various tools like Mythril, Slither and Remix IDE. At the same time this finding is based on critical analysis of the manual audit.
All issues found during automated analysis were manually reviewed and applicable vulnerabilities are presented in the Quick Stat section.
We found critical, 0 high, 0 medium and 0 low level issues.

Code Quality

The STAKING Smart Contract protocol consists of 2 smart contracts. Once deployed on the blockchain (only once), it is assigned a specific address and its properties / methods can be reused many times by other contracts in protocol. The BLOCKSOLUTIONS team has **not** provided scenario and unit test scripts, which would help to determine the integrity of the code in an automated way.
Overall, the code is not commented. Commenting can provide rich documentation for functions, return variables and more.

Documentation

As mentioned above, it's recommended to write comments in the smart contract code, so anyone can quickly understand the programming flow as well as complex code logic. We were given a STAKING Smart Contract smart contract code in the form of File.

Use of Dependencies

As per our observation, the libraries are used in this smart contract infrastructure that are based on well-known industry standard open-source projects. And even core code blocks are written well and systematically. This smart contract does not interact with other external smart contracts.

Risk Level	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to token loss etc.
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 tokens lose
Low	Low-level vulnerabilities are mostly related to outdated, unused etc. code snippets, that can't have significant impact on execution
Lowest / Code Style / Best	Lowest-level vulnerabilities, code style violations and info statements can't affect smart contract

Practice	execution and can be ignored.

Audit Findings

Critical

No Critical severity vulnerabilities were found.

High

No high severity vulnerabilities were found.

Medium

No Medium severity vulnerabilities were found.

Low

No Low severity vulnerabilities were found.

Conclusion

The Smart Contracts code passed the audit successfully. There were no warnings raised. We were given 2 contracts code. And we have used all possible tests based on given objects as files. So, it is good to go for production. Since possible test cases can be unlimited for such extensive smart contracts protocol, hence we provide no such guarantee of future outcomes. We have used all the latest static tools and manual observations to cover maximum possible test cases to scan everything. Smart contracts within the scope were manually reviewed and analyzed with static analysis tools. Smart Contract's high-level description of functionality was presented in Quick Stat section of the report. Audit report contains all found security vulnerabilities and other issues in the reviewed code.

Security state of the reviewed contract is "Well Secured".

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review:

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis:

Our audit techniques included manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's web site to get a high-level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation.

Documenting Results:

We follow a conservative, transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is conservative because we document our

suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally, follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this we analyze the feasibility of an attack in a live system.

Suggested Solutions:

We search for immediate mitigations that live deployments can take, and finally we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the details are made public.