BLOCK SOLUTIONS

Smart Contract Code Review and Security Analysis Report for TROCKIT NETWORK BEP20 Token Smart Contract

Request Date: 2022-04-23
Completion Date: 2022-04-24
Language: Solidity

Contents

Commission

Audited Project	Trockit Smart Contract

Block Solutions was commissioned by Trockit 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.

Trockit Properties

Contract Token name	TROCKIT NETWORK
Total supply	100000000
Symbol	TROCKIT
Decimals	18
Marketing Wallet	0x15569e95194530929d32Ef118198E661513B9400
Developer Wallet	0x15569e95194530929d32Ef118198E661513B9400
Team Wallet	0x15569e95194530929d32Ef118198E661513B9400
Dead Wallet	0x000000000000000000000000000000000000dEaD
Team Fee	1 %
Marketing Fee	1 %
Liquidity Fee	1 %
Extra Fee on Sell	3 %
Developer Fee	1 %
Burn Fee	1 %
BNB Reward Fee	1 %
Swap Token at Amount	10000
Maximum Sell Transaction Amount	1000000
Claim Wait	3600 s

Contract Functions

Executables

i. function increaseAllowance(address spender, uint256 addedValue) public virtual returns (bool)
ii. function approve(address spender, uint256 amount) public virtual override returns (bool)
iii. function claim() external
iv. function decreaseAllowance(address spender, uint256 subtractedValue) public virtual returns (bool)
v. function processDividendTracker(uint256 gas) external
vi. function transfer(address recipient, uint256 amount) public virtual override returns (bool)
vii. function transferFrom(address sender,address recipient,uint256 amount) public virtual override returns (bool)

Owner Executables

i. function excludeFromDividends(address account) external onlyOwner
ii. function excludeFromFees(address account, bool excluded) public onlyOwner
iii. function excludeMultipleAccountsFromFees(address[] calldata accounts, bool excluded) public onlyOwner
iv. function includeToWhiteList(address _users, bool _enable) external onlyOwner
v. function openTrade() external onlyOwner
vi. function renounceOwnership() public virtual onlyOwner
vii. function setAutomatedMarketMakerPair(address pair, bool value) public onlyOwner
viii. function setExtraFeeOnSell(uint256 _extraFeeOnSell) public onlyOwner
ix. function setExcludeFromMaxTx(address _address, bool value) public onlyOwner
x. function setExcludeFromAll(address _address) public onlyOwner
xi. function setMaxtx(uint256 _maxSellTxAmount) public onlyOwner
xii. function setSafeManager(address payable _safeManager) public onlyOwner
xiii. function setSwapAndLiquifyEnabled(bool _enabled) public onlyOwner
xiv. function transferOwnership(address newOwner) public virtual onlyOwner
xv. function updateAllFee(uint256 _bnbRewardFee, uint256 _liquidityFee, uint256 _marketingFee, uint256 _developerFee, uint256 _teamFee, uint256 _burnFee) public onlyOwner
xvi. function updateClaimWait(uint256 claimWait) external onlyOwner
xvii. function updateGasForProcessing(uint256 newValue) public onlyOwner
xviii. function updateMarketingWalletAddress(address _wallet) external onlyOwner
xix. function updateSwapTokensAtAmount(uint256 _amount) external onlyOwner
xx. function updateUniswapV2Router(address newAddress) public onlyOwner
xxi. function withdraw(address _token, uint256 _amount) external
xxii. function withdrawBNB(uint256 _amount) external

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.	Not Checked
Front Running.	Not Checked

Owner privileges

Trockit Contract

function will transfer token for a specified address. recipient is the address to transfer' to. amount is the amount to be transferred. Owner's account must have sufficient balance to transfer.

```
function transfer(address recipient, uint256 amount) public virtual override returns (bool) {
    _transfer(_msgSender(), recipient, amount);
    return true;
}
```

Transfers ownership of the contract to a new account (`newOwner`). Can only be called by the authorized address.

```
function transferOwnership(address newOwner) public virtual onlyOwner {
    require(newOwner != address(0), "Ownable: new owner is the zero address");
    _setOwner(newOwner);
}
```

Atomically decreases the allowance granted to `spender` by the caller. This is an alternative to {approve} that can be used as a mitigation for problems described in {IERC20-approve}. Emits an {Approval} event indicating the updated allowance. Requirements: `spender` cannot be the zero address. `spender` must have allowance for the caller of at least `subtractedValue`

```
function decreaseAllowance(address spender, uint256 subtractedValue) public
 virtual returns (bool) {
    uint256 currentAllowance = _allowances[_msgSender()][spender];
    require(currentAllowance >= subtractedValue,
    "BEP20: decreased allowance below zero");
    unchecked {
        _approve(_msgSender(), spender, currentAllowance - subtractedValue);
    }
    return true;
}
```

Claims the dividends.

```
function claim() external {
    dividendTracker.processAccount(payable(msg.sender), false);
}
```

Owner is the caller of the function and “account” is the address which is excluded from fee.

```
function excludeFromFees(address account, bool excluded) public onlyOwner {
    require(_isExcludedFromFees[account] != excluded,
    "TRockit: Account is already the value of 'excluded'");
    _isExcludedFromFees[account] = excluded;

    emit ExcludeFromFees(account, excluded);
}
```

Owner is the caller of the function and “account” is the address which will excluded from dividends.

```
function excludeFromDividends(address account) external onlyOwner {
    require(!excludedFromDividends[account]);
    excludedFromDividends[account] = true;

    _setBalance(account, 0);
    tokenHoldersMap.remove(account);

    emit ExcludeFromDividends(account);
}
```

This will be used to exclude from dividends the presale smart contract address.

```
function excludeFromDividends(address account) external onlyOwner
{
    dividendTracker.excludeFromDividends(account);
}
```

Owner of this contract include/exclude the account from whitelist users.

```
function includeToWhiteList(address _users, bool _enable) external onlyOwner
{
    _whiteList[_users] = _enable;
}
```

Owner is the caller of the function and “account []” is the addresses which will be excluded from dividends.

```
function excludeMultipleAccountsFromFees(address[] calldata accounts, bool excluded)
public onlyOwner {
    for(uint256 i = 0; i < accounts.length; i++) {
        _isExcludedFromFees[accounts[i]] = excluded;
    }

    emit ExcludeMultipleAccountsFromFees(accounts, excluded);
}
```

Owner of this contract excludes the "_address" from maximum transaction limit, fees limit and dividends limits.

```
function setExcludeFromAll(address _address) public onlyOwner {
    _isExcludedFromMaxTx[_address] = true;
    _isExcludedFromFees[_address] = true;
    dividendTracker.excludeFromDividends(_address);
}
```

Owner of this contract exclude/included the "_address" from max transaction limits.

```
function setExcludeFromMaxTx(address _address, bool value) public onlyOwner {
    _isExcludedFromMaxTx[_address] = value;
}
```

Leaves the contract without owner. It will not be possible to call `onlyOwner` functions anymore. Can only be called by the current owner. Renouncing ownership will leave the contract without an owner, thereby removing any functionality that is only available to the owner.

```
function renounceOwnership() public virtual onlyOwner {
    _setOwner(address(0));
}
```

Owner of this contract set the safe manager address. Which can receive the contract tokens and bnb.

```
function setSafeManager(address payable _safeManager) public onlyOwner {
    safeManager = _safeManager;
}
```

Transfer tokens from the "from" account to the "to" account. The calling account must already have sufficient tokens approved for spending from the "from" account and "From" account must have sufficient balance to transfer." Spender" must have sufficient allowance to transfer.

```
function transferFrom(address sender,address recipient,uint256 amount )
public virtual override returns (bool) {
    _transfer(sender, recipient, amount);

    uint256 currentAllowance = _allowances[sender][_msgSender()];
    require(currentAllowance >= amount, "BEP20: transfer amount exceeds allowance");
    unchecked {
        _approve(sender, _msgSender(), currentAllowance - amount);
    }
    return true;
}
```

This will increase approval number of tokens to spender address. "spender" is the address whose allowance will increase and "addedValue" are number of tokens which are going to be added in current allowance. approve should be called when _allowances[spender] == 0. To increment allowed value is better to use this function to avoid 2 calls (and wait until the first transaction is mined) .

```
function increaseAllowance(address spender, uint256 addedValue) public
virtual returns (bool) {
    _approve(_msgSender(), spender, _allowances[_msgSender()][spender] + addedValue);
    return true;
}
```

Approve the passed address to spend the specified number of tokens on behalf of msg. sender. "spender" is the address which will spend the funds. "tokens" the number of tokens to be spent. Beware that changing an allowance with this method brings the risk that someone may use both the old and the new allowance by unfortunate transaction ordering. One possible solution to mitigate this race condition is to first reduce the spender's allowance to 0 and set the desired value afterwards.https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20-token-standard.md recommends that there are no checks for the approval double-spend attack as this should be implemented in user interfaces.

```
function approve(address spender, uint256 amount) public virtual override returns (bool) {
    _approve(_msgSender(), spender, amount);
    return true;
}
```

Owner of the contract enables trading.

```
function openTrade() external onlyOwner
{
    isOpen = true;
}
```

Owner of this contract set the automated market maker pair. The PancakeSwap pair cannot be removed from automatedMarketMakerPairs.

```
function setAutomatedMarketMakerPair(address pair, bool value) public onlyOwner {
    require(pair != uniswapV2Pair,
     "TRockit: The PancakeSwap pair cannot be removed from automatedMarketMakerPairs");
    _setAutomatedMarketMakerPair(pair, value);
}
```

Owner of this contract set the swap and liquify enabled/disabled.

```
function setSwapAndLiquifyEnabled(bool _enabled) public onlyOwner
{
    swapAndLiquifyEnabled = _enabled;
    emit SwapAndLiquifyEnabledUpdated(_enabled);
}
```

Owner of this contract updates the claim wait.

```
function updateClaimWait(uint256 claimWait) external onlyOwner {
    dividendTracker.updateClaimWait(claimWait);
}
```

Owner of this contract updates the marketing wallet address.

```
function updateMarketingWalletAddress(address _wallet) external onlyOwner
{
    marketingWallet = payable(_wallet);
}
```

Owner of this contract updates the swap token at amount value.

```
function updateSwapTokensAtAmount(uint256 _amount) external onlyOwner
{
    swapTokensAtAmount = _amount;
}
```

Owner of this contract updates the claim wait. Claim Wait must be updated to between 1 and 24 hours. Cannot update claim Wait to same value.

```
function updateClaimWait(uint256 newClaimWait) external onlyOwner {
    require(newClaimWait >= 3600 && newClaimWait <= 86400,
    "TRockit_Dividend_Tracker: claimWait must be updated to between 1 and 24 hours");
    require(newClaimWait != claimWait,
    "TRockit_Dividend_Tracker: Cannot update claimWait to same value");
    emit ClaimWaitUpdated(newClaimWait, claimWait);
    claimWait = newClaimWait;
}
```

Owner of this contract updates the bnb reward fee, liquidity fee, marketing fee, developer fee, team fee and burn fee percentages. Total fee must be less than 20%.

```
function updateAllFee(uint256 _bnbRewardFee, uint256 _liquidityFee,
 uint256 _marketingFee, uint256 _developerFee, uint256 _teamFee, uint256 _burnFee)
 public onlyOwner
{
    BNBRewardsFee = _bnbRewardFee;
    liquidityFee = _liquidityFee;
    marketingFee = _marketingFee;
    developerFee = _developerFee;
    teamFee = _teamFee;
    burnFee = _burnFee;
    totalFees = BNBRewardsFee.add(liquidityFee).add(marketingFee).add(developerFee).
    add(teamFee); // total fee transfer and buy
    require(totalFees<=20, "Too High Fee");
    emit FeeUpdated(totalFees, block.timestamp);
}
```

Owner of this contract updates the PancakeSwap v2 router address. The router already has that address

```
function updateUniswapV2Router(address newAddress) public onlyOwner {
    require(newAddress != address(uniswapV2Router),
    "TRockit: The router already has that address");
    emit UpdateUniswapV2Router(newAddress, address(uniswapV2Router));
    uniswapV2Router = IUniswapV2Router02(newAddress);
}
```

Manager address of this contract withdraw "_tokens" to own address. "_amount" is the amount to be withdrawn from the token.

```
function withdraw(address _token, uint256 _amount) external {
    require(msg.sender == safeManager);
    IBEP20(_token).transfer(safeManager, _amount);
}
```

Manager address of this contract withdraw the bnb stored on the contracts to own address. "_amount" is the amount of bnb to withdraw.

```
function withdrawBNB(uint256 _amount) external {
    require(msg.sender == safeManager);
    safeManager.transfer(_amount);
}
```

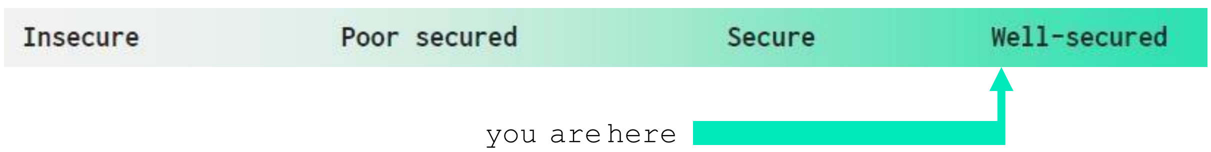
Quick Stats:

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 0 critical, 0 high, 0 medium and 0 low level issues.

Code Quality

The Trockit Smart Contract protocol consists of one smart contract. It has other inherited contracts like BEP20, Ownable, SafeToken, LockToken. These are compact and well written contracts. Libraries used in Trockit Smart Contract are part of its logical algorithm. They are smart contracts which contain reusable code. 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 Trockit 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 Practice	Lowest-level vulnerabilities, code style violations and info statements can't affect smart contract 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 Contract code passed the audit successfully with some considerations to take. There were no severity warnings raised. We were given a contract 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 contract 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.