BLOCK SOLUTIONS

Smart Contract Code Review and Security Analysis Report for PROVISION TREE BEP20 Token Smart Contract

Request Date: 2022-04-08
Completion Date: 2022-04-12
Language: Solidity

Contents

Commission

Audited Project	PROVISION TREE BEP20 Token Smart Contract

Block Solutions was commissioned by PROVISION TREE BEP20 TOKEN 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.

PROVISION TREE
BEP20 TOKEN Properties

Contract Token name	Provision Tree
Total Supply	500000000000
Decimals	18
Symbol	PTREE
Marketing Fee	3 %
Tax Fee	3 %
Social Funding	5 %
Max Room Rent	50000000000
Max Tx Amount	1% of total supply
Max Wallet Amount	1% of total supply
PancakeSwapV2Router	0x9Ac64Cc6e4415144C455BD8E4837Fea55603e5c3
PancakeSwapV2Pair	0x5501Dcaa6Fb9E1456b9425C381Eb5A8182047866

Contract Functions

Executables

i. function approve(address spender, uint256 amount) public returns (bool)
ii. function transfer(address recipient, uint256 amount) public returns (bool)
iii. function transferFrom(address sender, address recipient, uint256 amount) public returns (bool)

Owner Executable:

i. function activateAllFees() external onlyOwner()
ii. function activateHotelCalifornia(bool activated) external onlyOwner
iii. function blacklistWallet(address wallet) external onlyOwner()
iv. function deactivateAllFees() external onlyOwner()
v. function excludeFromFee(address account) public onlyOwner
vi. function excludeFromLimits(address account) public onlyOwner
vii. function includeInFee(address account) public onlyOwner
viii. function includeInLimits(address account) public onlyOwner
ix. function manualBurn(uint256 _amount) external onlyOwner()
x. function recoverAllTokens() external onlyOwner
xi. function recoverAllBNB() external onlyOwner
xii. function removeFromBlacklistWallet(address wallet) external onlyOwner()
xiii. function renounceOwnership() public virtual onlyOwner
xiv. function sendFeesOwnable() public onlyOwner
xv. function setMarketingAddress(address _marketingAddress) external onlyOwner()
xvi. function setMarketingFeePercent(uint256 _marketingFee) external onlyOwner()
xvii. function setMaxTransaction(uint256 _maxTransaction) external onlyOwner()
xviii. function setMaxTransaction(uint256 _maxTransaction) external onlyOwner()
xix. function setMaxWallet(uint256 _maxWallet) external onlyOwner()
xx. function setRoomRent(uint256 maxTxRoomRent) external onlyOwner
xxi. function setSocialFundingAddress(address _socialFundingAddress) external onlyOwner()
xxii. function setSocialFundingPercent(uint256 _socialFee) external onlyOwner()
xxiii. function setSwapAndLiquifyEnabled(bool _enabled) public onlyOwner
xxiv. function setTaxAddress(address _taxAddress) external onlyOwner()
xxv. function setTaxFeePercent(uint256 _taxFee) external onlyOwner()
xxvi. function excludeFromDividends(address account) external onlyOwner
xxvii. function changeMaxTxLimit(uint256 maxTx) external onlyOwner
xxviii. function changeMaxWalletLimit(uint256 maxWallet) external onlyOwner
xxix. function changeSwapTokensAtAmount(uint256 amount) external onlyOwner

xxx. function excludeFromFees(address account, bool excluded) public onlyOwner
xxxi. function excludeMultipleAccountsFromFees(address[] calldata accounts, bool excluded) public onlyOwner
xxxii. function renounceOwnership() public virtual onlyOwner
xxxiii. function setAutomatedMarketMakerPair(address pair, bool value) public onlyOwner
xxxiv. function setBUYFee(uint256 newRewardFee, uint256 newLiquidityFee, uint256 newBurnFee) external onlyOwner
xxxv. function setSELLFee(uint256 newRewardFee, uint256 newLiquidityFee, uint256 newBurnFee) external onlyOwner
xxxvi. function transferOwnership(address newOwner) public virtual onlyOwner
xxxvii. function updateClaimWait(uint256 claimWait) external onlyOwner
xxxviii. function updateDividendTracker(address newAddress) public onlyOwner
xxxix. function updateGasForProcessing(uint256 newValue) public onlyOwner
xl. function updateUniswapV2Router(address newAddress) public onlyOwner

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

PROVISION TREE BEP20 TOKEN Contract

function will transfer token for a specified address recipient is the address to transfer. “amount” is the amount to be transferred.

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

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

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

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 returns (bool) {
    _transfer(sender, recipient, amount);
    require(_allowances[sender][_msgSender()]-amount <= amount,
    "transfer amount exceeds allowance");
    _approve(sender, _msgSender(), _allowances[sender][_msgSender()]-amount);
    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 returns (bool) {
    _approve(_msgSender(), spender, amount);
    return true;
}
```

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 {
    transferOwnership(address(0));
}
```

Owner of the smart contract flip the Hotel California Mode.

```
function activateHotelCalifornia(bool activated) external onlyOwner {
    hotelCaliforniaMode = activated;
}
```

Owner of the contract activates all the previous fees.

```
function activateAllFees() external onlyOwner() {
    taxFee = previousTaxFee;
    marketingFee = previousMarketingFee;
    socialFunding = previousSocialFunding;
}
```

Owner of this contract blacklist wallet addresses.

```
function blacklistWallet(address wallet) external onlyOwner() {
    _blacklistedAccount[wallet] = true;
}
```

Owner of the contract excludes the address from fee.

```
function excludeFromFee(address account) public onlyOwner {
    _isExcludedFromFee[account] = true;
}
```

Owner of this contract includes the address into fee.

```
function includeInFee(address account) public onlyOwner {
    _isExcludedFromFee[account] = false;
}
```

Owner of this contract excludes the account from limits.

```
function excludeFromLimits(address account) public onlyOwner {
    _isExcludedFromLimits[account] = true;
}
```

Owner of the contract includes the address in to limit.

```
function includeInLimits(address account) public onlyOwner {
    _isExcludedFromLimits[account] = false;
}
```

Owner of this contract removes the address from blacklist wallet lists.

```
function removeFromBlacklistWallet(address wallet) external onlyOwner() {
    _blacklistedAccount[wallet] = false;
}
```

Owner of this contract set the tax fee receiver address.

```
function setTaxAddress(address _taxAddress) external onlyOwner() {
    taxAddress = payable(_taxAddress);
}
```

Owner of this contract updates the marketing fee receiver address.

```
function setMarketingAddress(address _marketingAddress) external onlyOwner() {
    marketingAddress = payable(_marketingAddress);
}
```

Owner of this contract updates the social funding fee receiver address.

```
function setSocialFundingAddress(address _socialFundingAddress) external onlyOwner() {
    socialFundingAddress = payable(_socialFundingAddress);
}
```

Owner of this contract updates the tax fee.

```
function setTaxFeePercent(uint256 _taxFee) external onlyOwner() {
    taxFee = _taxFee;
}
```

Owner of this contract set the social funding fee.

```
function setSocialFundingPercent(uint256 _socialFee) external onlyOwner() {
    socialFunding = _socialFee;
}
```

Owner of this contract updates the marketing fee.

```
function setMarketingFeePercent(uint256 _marketingFee) external onlyOwner() {
    marketingFee = _marketingFee;
}
```

Owner of this contract updates the maximum transaction amount.

```
function setMaxTransaction(uint256 _maxTransaction) external onlyOwner(){
    _maxTxAmount = _maxTransaction;
}
```

Owner of this contract updates the set maximum wallet amount.

```
function setMaxWallet(uint256 _maxWallet) external onlyOwner(){
    _maxWalletAmount = _maxWallet;
}
```

Owner of this contract deactivate all feeses.

```
function deactivateAllFees() external onlyOwner() {

    previousTaxFee = taxFee;
    previousMarketingFee = marketingFee;
    previousSocialFunding = socialFunding;

    taxFee = 0;
    marketingFee = 0;
    socialFunding = 0;
}
```

Owner of this contract burns the token from own address.

```
function manualBurn(uint256 _amount) external onlyOwner() {
    balances[msg.sender] -=_amount;
    _totalSupply -=_amount;
}
```

Owner of this contract set the maximum room rent.

```
function setRoomRent(uint256 maxTxRoomRent) external onlyOwner {
    maxRoomRent = maxTxRoomRent;
}
```

Owner of this contract send all contract's bnb to own address.

```
function recoverAllBNB() external onlyOwner {
   sendViaCall(payable(owner()), address(this).balance);
}
```

Owner of this contract sends all tokens to own address.

```
function recoverAllTokens() external onlyOwner {
    balances[owner()]+=(balanceOf(contractAddress));
    balances[contractAddress]-=(balanceOf(contractAddress));
}
```

Owner of the contract flips the swap and liquify condition.

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

Testing Summary

PASS

Block Solutions Believes
this smart contract security qualifications to passes listed be on digital asset exchanges.

12 APR, 2022

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 PROVISION TREE BEP20 TOKEN Smart Contract protocol consists of one smart contract. It has other inherited contracts like Context, Ownable. These are compact and well written contracts. Libraries used in PROVISION TREE BEP20 TOKEN 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 PROVISION TREE BEP20 TOKEN 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 warnings raised. The last change is advisable in order to provide more security to new holders. Nonetheless this is not necessary if the holders and/or investors feel confident with the contract owners. 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.