BLOCK SOLUTIONS

Smart Contract Code Review and Security Analysis Report for CLASSIC USDC BEP20 Token Smart Contract

Request Date: 2024-05-04
Completion Date: 2024-05-05
Language: Solidity

Contents

Commission

Audited Project	CLASSIC USDC BEP20 Token Smart Contract
Contract Creator	0xf8Bf882435f74825C170C1Ce8E312DC688E95cC1
Contract Owner	0x548bF14CAe58628932966B2180057cAAd83DCf97
Contract Address	0x7DB54758503c25200B331131C219acD64CFa74c8
Blockchain Platform	Binance Smart Chain Main net

Block Solutions was commissioned by CLASSIC USDC 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.

CLASSIC USDC BEP20 TOKEN Properties

Contract Token name	Classic USDC
Symbol	USDC
Decimals	18
Total Supply	50,000,000,000 USDC
Holder	12,501 Addresses
Transfer	15,973
PancakeSwapV2 Router	0x10ED43C718714eb63d5aA57B78B54704E256024E
PancakeSwapV2 Pair	0x9d0d5E0C05Ec7686468b4591e5573C3193A6B21a
Contract Creator	0xf8Bf882435f74825C170C1Ce8E312DC688E95cC1
Contract Owner	0x548bF14CAe58628932966B2180057cAAd83DCf97
Contract Address	0x7DB54758503c25200B331131C219acD64CFa74c8
Blockchain Platform	Binance Smart Chain Main net

Contract Functions

Executables

i. function approve(address spender, uint256 amount) public virtual override returns (bool)
ii. function decreaseAllowance(address spender, uint256 subtractedValue) public virtual returns (bool)
iii. function increaseAllowance(address spender, uint256 addedValue) public virtual returns (bool)
iv. function mint(address account, uint256 amount) public onlyOwner
v. function renounceOwnership() public virtual onlyOwner
vi. function toggleTransfersPaused(bool _paused) public onlyOwner
vii. function toggleBuyingEnabled(bool _enabled) public onlyOwner
viii. function toggleSellingEnabled(bool _enabled) public onlyOwner
ix. function transfer(address recipient, uint256 amount) public virtual override returns (bool)
x. function transferFrom(address sender, address recipient, uint256 amount) public virtual override returns (bool)
xi. function transferOwnership(address newOwner) public virtual onlyOwner
xii. function whitelistForTransfers(address _account, bool _whitelisted) public onlyOwner
xiii. function whitelistForBuying(address _account, bool _whitelisted) public onlyOwner
xiv. function whitelistForSelling(address _account, bool _whitelisted) 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

Executable Functions

CLASSIC USDC BEP20 TOKEN Contract

This 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 virtual override
returns (bool) {
    _transfer(_msgSender(), recipient, amount);
    return true;
}
```

Transfer tokens from the “sender” account to the “recipient” account. The calling account must already have sufficient tokens approved for spending from the “sender” account and “sender” account must have sufficient balance to transfer. “recipient” 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, "ERC20: transfer amount exceeds allowance");
    unchecked {
        _approve(sender, _msgSender(), currentAllowance - 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. “amount” the number of tokens to be spent.

```
function approve(address spender, uint256 amount) public virtual override
 returns (bool) {
    _approve(_msgSender(), spender, 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 allowed[spender] == 0. To increment allowed 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;
}
```

This will decrease approval number of tokens to spender address. “spender” is the address whose allowance will decrease and “subtractedValue” are number of tokens which are going to be subtracted from current allowance.

```
function decreaseAllowance(address spender, uint256 subtractedValue) public
 virtual returns (bool) {
    uint256 currentAllowance = _allowances[_msgSender()][spender];
    require(currentAllowance >= subtractedValue,
     "ERC20: decreased allowance below zero");
    unchecked {
        _approve(_msgSender(), spender, currentAllowance - subtractedValue);
    }

    return true;
}
```

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

```
function renounceOwnership() public virtual onlyOwner {
    emit OwnershipTransferred(_owner, address(0));
    _owner = address(0);
}
```

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;
}
```

Owner of this contract can mint new tokens on any address.

```
function mint(address account, uint256 amount) public onlyOwner {
    _mint(account, amount);
}
```

Owner of this contract can toggle the transfer pause status to true/false.

```
function toggleTransfersPaused(bool _paused) public onlyOwner {
    transfersPaused = _paused;
    emit TransfersPausedUpdated(_paused);
}
```

Owner of this contract can toggle the buying pause status to true/false.

```
function toggleBuyingEnabled(bool _enabled) public onlyOwner {
    buyingEnabled = _enabled;
    emit BuyingEnabledUpdated(_enabled);
}
```

Owner of this contract can toggle the selling pause status to true/false.

```
function toggleSellingEnabled(bool _enabled) public onlyOwner {
    sellingEnabled = _enabled;
    emit SellingEnabledUpdated(_enabled);
}
```

Owner of this contract can add address to transfer whitelist list or remove it from transfer whitelist addresses list.

```
function whitelistForTransfers(address _account, bool _whitelisted) public onlyOwner {
    transferWhitelist[_account] = _whitelisted;
    emit WhitelistedForTransfer(_account, _whitelisted);
}
```

Owner of this contract can add address to buy whitelist list or remove it from buy whitelist addresses list.

```
function whitelistForBuying(address _account, bool _whitelisted) public onlyOwner {
    buyWhitelist[_account] = _whitelisted;
    emit WhitelistedForBuying(_account, _whitelisted);
}
```

Owner of this contract can add address to sell whitelist list or remove it from sell whitelist addresses list.

```
function whitelistForSelling(address _account, bool _whitelisted) public onlyOwner {
    sellWhitelist[_account] = _whitelisted;
    emit WhitelistedForSelling(_account, _whitelisted);
}
```

Testing Summary

PASS

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

5th May, 2024

YOUR SCORE
100%

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 CLASSIC USDC BEP20 TOKEN Smart Contract protocol consists of ERC20, Ownable. 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 CLASSIC USDC 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. There were no 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.