



## BLOCK SOLUTIONS

# Smart Contract Code Review and Security Analysis Report for MTHAIBAHT BEP20 Token Smart Contract



Request Date: 2025-08-14

Completion Date: 2025-08-14

Language: Solidity



## Contents

Commission .....	3
MTHAIBAHT BEP20 TOKEN Properties .....	4
Contract Functions .....	5
Executables .....	5
Checklist.....	6
Executable Functions .....	8
MTHAIBAHT BEP20 TOKEN Contract.....	8
Testing Summary .....	12
Quick Stats: .....	13
Executive Summary .....	14
Code Quality .....	14
Documentation .....	14
Use of Dependencies.....	14
Critical .....	15
High .....	15
Medium.....	15
Low .....	15
Conclusion .....	16
Our Methodology.....	16



## Smart Contract Code Review and Security Analysis Report for MTHAIBAHT BEP20 Token Smart Contract

---

### Commission

<b>Audited Project</b>	MTHAIBAHT BEP20 Token Smart Contract
<b>Smart Contract Creator</b>	0x7F53f476E19023F9052480A7453bDdd8a61628F6
<b>Smart Contract Address</b>	0xD7B86e2C3eB88f7A1Fa5924ecc3b69a274c74c4
<b>Smart Contract Owner</b>	0x7F53f476E19023F9052480A7453bDdd8a61628F6
<b>Blockchain Platform</b>	Binance Smart Chain Main net

Block Solutions was commissioned by MTHAIBAHT 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.



## MTHAIBAHT BEP20 TOKEN Properties

<b>Contract Token name</b>	MTHAIBAHT
<b>Symbol</b>	MTHB
<b>Decimals</b>	8
<b>Total Supply</b>	1,000,000,000,000 MTHB
<b>Holder</b>	4,274 Addresses
<b>TOTAL TRANSFERS</b>	5,226
<b>Owner Holding Supply</b>	99.99 %
<b>Smart Contract Creator</b>	0x7F53f476E19023F9052480A7453bDdd8a61628F6
<b>Smart Contract Address</b>	0xaD7B86e2C3eB88f7A1Fa5924ecc3b69a274c74c4
<b>Smart Contract Owner</b>	0x7F53f476E19023F9052480A7453bDdd8a61628F6
<b>Blockchain Platform</b>	Binance Smart Chain Main net



## Contract Functions

### Executables

- i. function approve(address spender, uint256 amount) public virtual override returns (bool)
- ii. function mint(uint256 amount) public onlyOwner
- iii. function decreaseAllowance(address spender, uint256 subtractedValue) public virtual returns (bool)
- iv. function increaseAllowance(address spender, uint256 addedValue) public virtual returns (bool)
- v. function renounceOwnership() public virtual onlyOwner
- vi. function transferOwnership(address newOwner) public virtual onlyOwner
- vii. function transfer(address recipient, uint256 amount) public virtual override returns (bool)
- viii. function transferFrom(address sender, address recipient, uint256 amount) public virtual override returns (bool)
- ix. function rescueBalance(address toAddr) public onlyOwner
- x. function rescueToken(address tokenAddress, address receiver) public onlyOwner



## Smart Contract Code Review and Security Analysis Report for MTHAIBAHT BEP20 Token Smart Contract

---

### 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



## Smart Contract Code Review and Security Analysis Report for MTHAIBAHT BEP20 Token Smart Contract

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



## Executable Functions

### MTHAIBAHT 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 function allows the contract owner to mint (amount) new tokens and assign them directly to their own address. The `onlyOwner` modifier restricts access to the owner account. The `_mint` function increases the total supply and updates the balance of the recipient (`owner()`).

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

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;  
}
```

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[to] == 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;  
}
```

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));  
}
```

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"  
    );  
    _setOwner(newOwner);  
}
```

This function allows the contract owner to withdraw all ETH held by the contract and send it to a specified address (toAddr). The onlyOwner modifier restricts execution to the owner. The call method transfers the full contract balance and returns a boolean indicating success, which is then enforced by the require statement.

```
function rescueBalance(address toAddr) public onlyOwner {  
    (bool success, ) = toAddr.call{value:  
        address(this).balance}("");  
    require(success);  
}
```

This function allows the contract owner to withdraw the entire balance of any ERC-20 token held by the contract and send it to a specified receiver address. The onlyOwner modifier ensures that only the contract owner can execute this function. The function queries the contract's token balance and transfers all tokens to the receiver.



```
function rescueToken(address tokenAddress ,  
    address receiver) public onlyOwner {  
    uint balance = IERC20(tokenAddress)  
        .balanceOf(address(this));  
    IERC20(tokenAddress).transfer(receiver, balance);  
}
```



## Testing Summary

# PASS

### BLOCK SOLUTIONS BELIEVES

*This smart contract passes the security qualifications to be listed on digital assets exchanges.*

**14<sup>th</sup> AUGUST, 2025**





## Smart Contract Code Review and Security Analysis Report for MTHAIBAHT BEP20 Token Smart Contract

### Quick Stats:

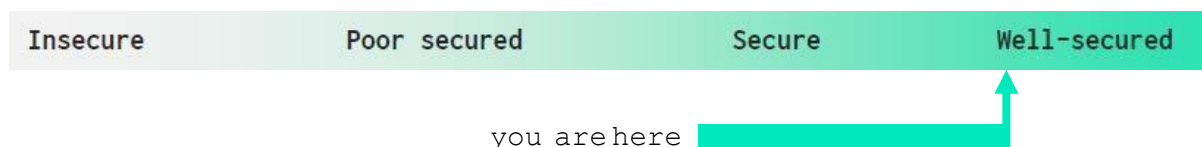
Main Category	Subcategory	Result
Contract Programming	Solidity version not specified	Passed
	Solidity version too old	Passed
	Integer overflow/underflow	Passed
	Function input parameters lack of check	Passed
	Function input parameters check bypass	Passed
	Function access control lacks management	Passed
	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 MTHAIBAHT BEP20 TOKEN Smart Contract protocol consists of Context, IERC20, IERC20Metadata, IERC20Errors. 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 MTHAIBAHT 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.



## Smart Contract Code Review and Security Analysis Report for MTHAIBAHT BEP20 Token Smart Contract

---

Risk Level	Description
<b>Critical</b>	Critical vulnerabilities are usually straightforward to exploit and can lead to token loss etc.
<b>High</b>	High-level vulnerabilities are difficult to exploit; however, they also have significant impact on smart contract execution, e.g. public access to crucial functions
<b>Medium</b>	Medium-level vulnerabilities are important to fix; however, they can't lead to tokens lose
<b>Low</b>	Low-level vulnerabilities are mostly related to outdated, unused etc. code snippets, that can't have significant impact on execution
<b>Lowest / Code Style / Best Practice</b>	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





## Smart Contract Code Review and Security Analysis Report for MTHAIBAHT BEP20 Token Smart Contract

---

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.