Ether Authority

www.EtherAuthority.io
audit@etherauthority.io

SMART CONTRACT

Security Audit Report

Project:	USD Coin
Website:	circle.com/usdc
Platform:	Binance Network
Language:	Solidity
Date:	April 14th, 2025

Table of Contents

Email: audit@EtherAuthority.io

THIS IS A SECURITY AUDIT REPORT DOCUMENT THAT MAY CONTAIN INFORMATION THAT IS CONFIDENTIAL. WHICH INCLUDES ANY POTENTIAL VULNERABILITIES AND MALICIOUS CODES WHICH CAN BE USED TO EXPLOIT THE SOFTWARE. THIS MUST BE REFERRED INTERNALLY AND ONLY SHOULD BE MADE AVAILABLE TO THE PUBLIC AFTER ISSUES ARE RESOLVED.

Email: audit@EtherAuthority.io

Introduction

As part of EtherAuthority's community smart contract audit initiatives, the smart contract of the USD Coin Token from circle.com/usdc was audited. The audit was performed using manual analysis and automated software tools. This report presents all the findings regarding the audit performed on April 14th, 2025.

The purpose of this audit was to address the following:

- Ensure that all claimed functions exist and function correctly.
- Identify any security vulnerabilities that may be present in the smart contract.

Project Background

This smart contract implements a BEP-20 token using Solidity and is compatible with the Binance Smart Chain (BSC). It includes:

Core Features

- **Standard BEP-20 Interface**: Implements functions such as totalSupply, balanceOf, transfer, approve, transferFrom, and related events.
- **Upgradeable Design**: Uses an initialize function instead of a constructor, following upgradeable proxy patterns via the Initializable contract.
- **Ownership Management**: The contract includes onlyOwner modifier, transferOwnership, and renounceOwnership functions.
- **Mintable Option**: At initialization, a token can be configured as mintable or not.
- **SafeMath Library**: All arithmetic uses SafeMath to prevent overflows.

Audit scope

Name	Code Review and Security Analysis Report for USD Coin Token Smart Contracts
Platform	**Binance Network**
File 1	BEP20TokenImplementation.sol
File 1 Smart Contract Code	0xba5fe23f8a3a24bed3236f05f2fcf35fd0bf0b5c
File 2	BEP20UpgradeableProxy.sol
File 2 Smart Contract Code	0x8ac76a51cc950d9822d68b83fe1ad97b32cd580d
Audit Date	April 14th, 2025

Email: audit@EtherAuthority.io

Claimed Smart Contract Features

Claimed Feature Detail	Our Observation
Tokenomics: • Name: USD Coin • Symbol: USDC	**YES, This is valid.**
Key Features: **BEP-20 Standard Compliance** • Implements all core BEP-20 functions: transfer, approve, transferFrom, balanceOf, totalSupply. • Compatible with wallets, DEXs, and other BEP-20-enabled platforms. **Upgradeable Contract** • Based on OpenZeppelin's Initializable pattern. • Uses initialize() instead of a constructor to support upgradeable proxy deployments. **Ownership & Access Control** • The onlyOwner modifier restricts sensitive actions to the contract owner. • Ownership can be transferred or renounced using transferOwnership and renounceOwnership. **Token Minting (Optional)** • Minting is controlled via a _mintable flag set during initialization.	**YES, This is valid.**

• If enabled, the owner can mint new tokens using mint(address to, uint256 amount). **Token Metadata Configuration** • Token name, symbol, and decimals are set during initialization. • Public view functions allow retrieval of metadata. **Safe Arithmetic** • All math operations use OpenZeppelin's SafeMath to prevent overflows/underflows. **Allowance System** • Supports standard allowance pattern with approve, increaseAllowance, and decreaseAllowance. • Enables delegated spending through transferFrom. **Event Emissions** • Emits Transfer and Approval events for all relevant operations. • Emits OwnershipTransferred when ownership changes. **Efficient State Management** • Stores balances in _balances mapping and allowances in _allowances. • Designed for gas efficiency and BEP-20 compatibility.	

Audit Summary

According to the standard audit assessment, the Customer's solidity-based smart contract is **“Secured.”** Also, these contracts contain owner control, which does not make them fully decentralized.

We used various tools like Slither, Solhint, and Remix IDE. At the same time, this finding is based on a critical analysis of the manual audit.

All issues found during automated analysis were manually reviewed, and applicable vulnerabilities are presented in the Audit overview section. The general overview is presented in the AS-IS section, and all identified issues can be found in the Audit overview section.

We found 0 critical, 0 high, 0 medium, 3 low, and 2 very low-level issues.

Investors' Advice: A Technical audit of the smart contract does not guarantee the ethical nature of the project. Any owner-controlled functions should be executed by the owner with responsibility. All investors/users are advised to do their due diligence before investing in the project.

Technical Quick Stats

Main Category	Subcategory	Result
Contract Programming	Solidity version not specified	Passed
	Solidity version too old	Moderated
	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	Moderated
	Human/contract checks bypass	Passed
	Random number generation/use vulnerability	N/A
	Fallback function misuse	Passed
	Race condition	Passed
	Logical vulnerability	Passed
	Features claimed	Passed
	Other programming issues	Moderated
Code Specification	Function visibility not explicitly declared	Passed
	Var. storage location not explicitly declared	Passed
	Use keywords/functions to be deprecated	Passed
	Unused code	Moderated
Gas Optimization	“Out of Gas” Issue	Passed
	High consumption ‘for/while’ loop	Passed
	High consumption ‘storage’ storage	Passed
	Assert() misuse	Passed
Business Risk	The maximum limit for mintage is not set	Passed
	“Short Address” Attack	Passed
	“Double Spend” Attack	Passed

Overall Audit Result: **PASSED**

Code Quality

This audit scope has 1 smart contract. Smart contracts contain Libraries, Smart contracts, inheritance, and Interfaces. This is a compact and well-written smart contract.

The libraries in USDC Token are part of its logical algorithm. A library is a different type of smart contract that contains 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 the USDC Token.

The EtherAuthority team has no scenario and unit test scripts, which would have helped to determine the integrity of the code in an automated way.

Code parts are well commented on in the smart contract. Ethereum's NatSpec commenting style is recommended.

Documentation

We were given a USDC Token smart contract code in the form of a BSCscan web link.

As mentioned above, code parts are well commented on. And the logic is straightforward. So it is easy to quickly understand the programming flow as well as complex code logic. Comments are very helpful in understanding the overall architecture of the protocol.

Use of Dependencies

As per our observation, the libraries used in this smart contract infrastructure that is based on well-known industry standard open-source projects.

Apart from libraries, its functions are not used in external smart contract calls.

AS-IS overview

BEP20TokenImplementation.sol : Functions

Sl.	Functions	Type	Observation	Conclusion
1	constructor	write	Unprotected Upgradeable Constructor	Refer Audit Findings
2	onlyOwner	modifier	Passed	No Issue
3	initialize	write	initializer	No Issue
4	renounceOwnership	write	Lack of Events	Refer Audit Findings
5	transferOwnership	write	access only Owner	No Issue
6	mintable	external	Passed	No Issue
7	getOwner	external	Passed	No Issue
8	decimals	external	Passed	No Issue
9	symbol	external	Passed	No Issue
10	name	external	Passed	No Issue
11	totalSupply	external	Passed	No Issue
12	balanceOf	external	Passed	No Issue
13	transfer	external	Passed	No Issue
14	allowance	external	Passed	No Issue
15	approve	external	Passed	No Issue
16	transferFrom	external	Passed	No Issue
17	increaseAllowance	write	Passed	No Issue
18	decreaseAllowance	write	Passed	No Issue
19	mint	write	Lack of Events, Missing Mint Function Access Control	Refer Audit Findings
20	burn	write	Lack of Access Control on `burn()` Function, Lack of Events	Refer Audit Findings
21	_transfer	internal	Passed	No Issue
22	_mint	internal	Passed	No Issue
23	_burn	internal	Passed	No Issue
24	_approve	internal	Passed	No Issue
25	_burnFrom	internal	Passed	No Issue
26	initializer	modifier	Passed	No Issue
27	_isConstructor	read	Passed	No Issue

Severity Definitions

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 a significant impact on smart contract execution, e.g., public access to crucial
Medium	Medium-level vulnerabilities are important to fix; however, they can't lead to tokens being lost
Low	Low-level vulnerabilities are mostly related to outdated, unused, etc. code snippets, that can't have a 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.

Email: audit@EtherAuthority.io

Audit Findings

Critical Severity

No Critical severity vulnerabilities were found.

High Severity

No High severity vulnerabilities were found.

Medium

No Medium severity vulnerabilities were found.

Low

(1) Lack of Access Control on `burn()` Function:

The `burn()` function is publicly accessible, meaning any token holder can burn their own tokens. While this may be intended, it must be clarified because it affects the total supply.

Resolution: Ensure the burn functionality aligns with the intended tokenomics. If burning should be restricted, add an `onlyOwner` modifier or a whitelist mechanism. If not, document this behavior.

(2) Lack of Events for Mint, Burn, and Ownership Renounce:

There are no custom events emitted for `mint`, `burn`, or `renounceOwnership`.

Resolution: Emit events for these functions to improve transparency and off-chain monitoring.

(3) Missing Mint Function Access Control:

The `_mint` function is used in `initialize` to mint tokens to the owner, but if the contract is made mintable (`_mintable = true`), there is no public `mint` function with access control, or the existing `_mint` function isn't restricted in any way.

Resolution: If you intend to have minting available, add a `mint` function and restrict it to `onlyOwner` or a proper access role.

```
function mint(address to, uint256 amount) external onlyOwner {
    require(_mintable, "Token: minting is disabled");
    _mint(to, amount);
}
```

Very Low / Informational / Best practices:

(1) Unprotected Upgradeable Constructor:

The constructor is empty but public, which is misleading. In upgradeable contracts, constructors should be removed or made internal to prevent confusion or incorrect use.

Resolution: Change the constructor to:

```
constructor() internal {}
```

Or better yet, remove it if unused.

(2) No Pausing Mechanism:

No mechanism to pause transfers or minting in case of emergency.

Resolution: Implement a pausable pattern with modifiers like `whenNotPaused` to improve security in case of an incident.

Centralization

This smart contract has some functions that can be executed by the Admin (Owner) only. If the admin wallet's private key is compromised, then it creates trouble. The following are Admin functions:

BEP20TokenImplementation.sol

- renounceOwnership: Deleting ownership will leave the contract without an owner, removing any owner-only functionality.
- transferOwnership: The Current owner can transfer ownership of the contract to a new account.
- mint: The owner can create `amount` tokens and assign them to `msg.sender`, increasing the total supply.

Conclusion

We were given a contract code in the form of bscscan web links. We have used all possible tests based on the given objects as files. We observed 3 low and 2 informational issues in the smart contract, and those issues are not critical. So, **it's good to go for production**.

Since possible test cases can be unlimited for such smart contract protocols, we provide no such guarantee of future outcomes. We have used all the latest static tools and manual observations to cover the 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 the As-is overview section of the report.

The audit report contains all found security vulnerabilities and other issues in the reviewed code.

The security state of the reviewed smart contract, based on standard audit procedure scope, is **"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 website 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 not to 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.

Disclaimers

EtherAuthority.io Disclaimer

EtherAuthority team has analyzed this smart contract in accordance with the best industry practices at the date of this report, in relation to: cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report, (Source Code); the Source Code compilation, deployment and functionality (performing the intended functions).

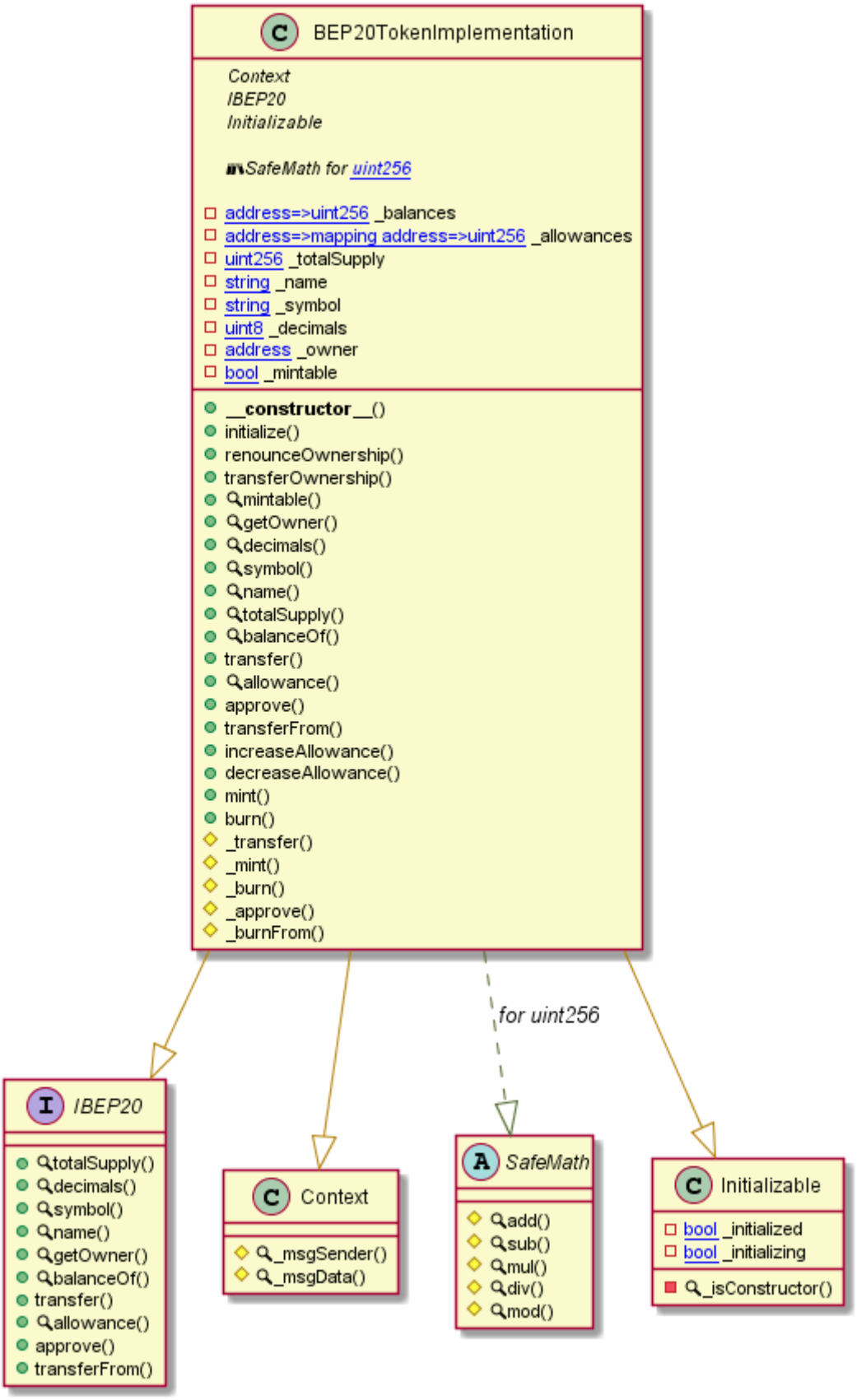
Due to the fact that the total number of test cases is unlimited, the audit makes no statements or warranties on the security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only. We also suggest conducting a bug bounty program to confirm the high level of security of this smart contract.

Technical Disclaimer

Smart contracts are deployed and executed on the blockchain platform. The platform, its programming language, and other software related to the smart contract can have their own vulnerabilities that can lead to hacks. Thus, the audit can't guarantee explicit security of the audited smart contracts.

Appendix

Code Flow Diagram - USD Coin Token



Email: audit@EtherAuthority.io

Slither Results Log

Slither is a Solidity static analysis framework that uses vulnerability detectors, displays contract details, and provides an API for writing custom analyses. It helps developers identify vulnerabilities, improve code comprehension, and prototype custom analyses quickly. The analysis includes a report with warnings and errors, allowing developers to quickly prototype and fix issues.

We did the analysis of the project altogether. Below are the results.

Slither Log >> BEP20TokenImplementation.sol

```
INFO:Detectors:
BEP20TokenImplementation.initialize(string,string,uint8,uint256,bool,address).name
(BEP20TokenImplementation.sol#389) shadows:
	- BEP20TokenImplementation.name() (BEP20TokenImplementation.sol#451-453) (function)
	- IBEP20.name() (BEP20TokenImplementation.sol#29) (function)
BEP20TokenImplementation.initialize(string,string,uint8,uint256,bool,address).mintable
(BEP20TokenImplementation.sol#389) shadows:
	- BEP20TokenImplementation.mintable() (BEP20TokenImplementation.sol#423-425)
(function)
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#local-variable-shadowing
INFO:Detectors:
BEP20TokenImplementation.initialize(string,string,uint8,uint256,bool,address).owner
(BEP20TokenImplementation.sol#389) lacks a zero-check on :
		- _owner = owner (BEP20TokenImplementation.sol#390)
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#missing-zero-address-validation
INFO:Detectors:
Initializable._isConstructor() (BEP20TokenImplementation.sol#338-349) uses assembly
	- INLINE ASM (BEP20TokenImplementation.sol#347)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#assembly-usage
INFO:Detectors:
BEP20TokenImplementation._burnFrom(address,uint256)
(BEP20TokenImplementation.sol#664-667) is never used and should be removed
Context._msgData() (BEP20TokenImplementation.sol#121-124) is never used and should be
removed
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dead-code
INFO:Detectors:
Version constraint ^0.6.0 contains known severe issues
(https://solidity.readthedocs.io/en/latest/bugs.html)
	- AbiReencodingHeadOverflowWithStaticArrayCleanup
	- DirtyBytesArrayToStorage
```


Email: audit@EtherAuthority.io

```
        - NestedCalldataArrayAbiReencodingSizeValidation
        - ABIDecodeTwoDimensionalArrayMemory
        - KeccakCaching
        - EmptyByteArrayCopy
        - DynamicArrayCleanup
        - MissingEscapingInFormatting
        - ArraySliceDynamicallyEncodedBaseType
        - ImplicitConstructorCallvalueCheck
        - TupleAssignmentMultiStackSlotComponents
        - MemoryArrayCreationOverflow
        - YulOptimizerRedundantAssignmentBreakContinue.
solc-0.6.12 is an outdated solc version. Use a more recent version (at least 0.8.0), if possible.
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
INFO:Detectors:
Redundant expression "this (BEP20TokenImplementation.sol#122)" inContext
(BEP20TokenImplementation.sol#116-125)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#redundant-statements
INFO:Slither:BEP20TokenImplementation.sol analyzed (5 contracts with 93 detectors), 13
result(s) found
```

Solidity Static Analysis

Static code analysis is used to identify many common coding problems before a program is released. It involves examining the code manually or using tools to automate the process. Static code analysis tools can automatically scan the code without executing it.

BEP20TokenImplementation.sol

```
Inline assembly:
The Contract uses inline assembly, this is only advised in rare cases. Additionally static analysis modules do not parse inline Assembly, this can lead to wrong analysis results.
Pos: 349:8:

Gas costs:
Gas requirement of function BEP20TokenImplementation.mint is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)
Pos: 566:4:

ERC20:
ERC20 contract's "decimals" function should have "uint8" as return type
Pos: 439:4:

Similar variable names:
BEP20TokenImplementation._burn(address,uint256) : Variables have very similar names "account" and "amount". Note: Modifiers are currently not considered by this static analysis.
Pos: 632:16:

Guard conditions:
Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.
Pos: 653:8:
```

Solhint Linter

Solhint Linters are the utility tools that analyze the given source code and report programming errors, bugs, and stylistic errors. For the Solidity language, there are some linter tools available that a developer can use to improve the quality of their Solidity contracts.

BEP20TokenImplementation.sol

```
Compiler version ^0.6.0 does not satisfy the ^0.5.8 semver
requirement
Pos: 1:6
Error message for require is too long
Pos: 9:213
Compiler version >=0.4.24 <0.7.0 does not satisfy the ^0.5.8 semver
requirement
Pos: 1:292
Error message for require is too long
Pos: 9:323
Compiler version ^0.6.0 does not satisfy the ^0.5.8 semver
requirement
Pos: 1:355
Code contains empty blocks
Pos: 26:376
Error message for require is too long
Pos: 9:652
Error message for require is too long
Pos: 9:653
```

Software analysis result:

This software reported many false positive results and some are informational issues. So, those issues can be safely ignored.

Email: audit@EtherAuthority.io