www.EtherAuthority.io
audit@etherauthority.io

SMART CONTRACT

Security Audit Report

Project: Tether USD (USDT)
Website: binance.com
Platform: Binance Network
Language: Solidity
Date: April 3rd, 2025

Table of Contents

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.

Introduction

As part of EtherAuthority's community smart contract audit initiatives, the smart contract of the USDT Token from binance.com was audited. The audit has been performed using manual analysis as well as using automated software tools. This report presents all the findings regarding the audit performed on April 3rd, 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

The solidity contract for a BEP20 token named "BEP20USDT," which mimics Tether (USDT) on the Binance Smart Chain (BSC). Here's a quick description of the contract:

Description of BEP20USDT Contract:

- Implements the **IBEP20** interface, following the BEP20 token standard.
- **SafeMath** is used for safe arithmetic operations to prevent overflows and underflows.
- Extends the **Ownable** contract, allowing an owner to manage specific administrative functions.
- Defines standard **BEP20 functions**, including transfer, approve, transferFrom, allowance, and balance tracking.
- Supports **minting** (creating new tokens) and **burning** (removing tokens from circulation).
- The contract initializes with a **total supply of 30 million USDT** assigned to the deployer.
- Includes increaseAllowance and decreaseAllowance functions to handle approval changes safely.

Audit scope

Name	**Code Review and Security Analysis Report for Tether USD (USDT) Token Smart Contract**
Platform	**Binance Network**
File	BEP20USDT.sol
Smart Contract Code	0x55d398326f99059ff775485246999027b3197955
Audit Date	April 3rd, 2025

Claimed Smart Contract Features

<table>
<tr><th>Claimed Feature Detail</th><th>Our Observation</th></tr>
<tr><td>Tokenomics:
Name: Tether USD
Symbol: USDT
Decimals: 18</td><td>YES, This is valid.</td></tr>
<tr><td>Key Features:
1. BEP20 Standard Implementation
Follows the BEP20 token standard, ensuring compatibility with Binance Smart Chain (BSC) applications.
Implements functions such as transfer, approve, transferFrom, and allowance.
2. SafeMath for Secure Arithmetic
SafeMath is used to prevent overflows and underflows in token operations.
3. Ownable (Admin Control)
The contract extends Ownable, granting only the owner permission to perform specific administrative tasks.
4. Minting (Token Creation)
Allows the owner to mint new tokens and increase the total supply.
5. Burning (Token Reduction)</td><td></td></tr>
</table>

• Supports **burning tokens**, allowing holders to permanently remove tokens from circulation. **6. Fixed Initial Supply** • **30,000,000 BEP20USDT** tokens are minted at deployment and assigned to the deployer. **7. Approval & Allowance System** • Users can approve third-party **spenders** to use their tokens via approve and transferFrom. • Supports increaseAllowance and decreaseAllowance for flexible spending permissions. **8. Standard Events for Transparency** • Emits Transfer and Approval events for blockchain traceability.	

Audit Summary

According to the standard audit assessment, the Customer's solidity-based smart contract is **"Secured."** This token contract has ownership control; hence, it is not fully 100% 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 3 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	Moderated
	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

Email: audit@EtherAuthority.io

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 USDT 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 USDT 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 USDT 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

BEP20USDT.sol : Functions

Sl.	Functions	Type	Observation	Conclusion
1	constructor	write	Passed	No Issue
2	getOwner	external	Passed	No Issue
3	decimals	external	Passed	No Issue
4	symbol	external	Passed	No Issue
5	name	external	Passed	No Issue
6	totalSupply	external	Passed	No Issue
7	balanceOf	external	Passed	No Issue
8	transfer	external	Lack of Reentrancy Guard	Refer Audit Findings
9	allowance	external	Passed	No Issue
10	approve	external	Passed	No Issue
11	transferFrom	external	Passed	No Issue
12	increaseAllowance	write	Passed	No Issue
13	decreaseAllowance	write	Passed	No Issue
14	mint	write	Lack of Reentrancy Guard, No Events for Function, Potential for Large Gas Fees Due to Token Minting	Refer Audit Findings
15	burn	write	Burn Function Doesn't Restrict to Owner, Lack of Reentrancy Guard, No Events for Function, Potential for Large Gas Fees Due to Token Burning	Refer Audit Findings
16	_transfer	internal	Passed	No Issue
17	_mint	internal	Passed	No Issue
18	_burn	internal	Passed	No Issue
19	_approve	internal	Passed	No Issue
20	_burnFrom	internal	Passed	No Issue
21	owner	read	Passed	No Issue
22	onlyOwner	modifier	Passed	No Issue
23	renounceOwnership	write	access only Owner	No Issue
24	transferOwnership	write	access only Owner	No Issue
25	_transferOwnership	internal	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 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 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 Severity

No Critical severity vulnerabilities were found.

High Severity

No High severity vulnerabilities were found.

Medium

No Medium-severity vulnerabilities were found.

Low

(1) Burn Function Doesn't Restrict to Owner:

The burn() function can be called by any user, allowing anyone to burn tokens from their own balance. While this may be intended, if the token supply is controlled centrally (like in many tokens), this can create vulnerabilities by allowing users to burn tokens arbitrarily.

Resolution: Restrict the burn() function to only the owner or specific roles, unless it's an intentional feature to allow anyone to burn tokens.

(2) Lack of Reentrancy Guard:

The contract does not have a reentrancy guard in place, which is essential to prevent reentrancy attacks, especially in token transfer and mint/burn functions.

Resolution: Implement the ReentrancyGuard modifier from OpenZeppelin or custom reentrancy protection for functions such as transfer(), mint(), and burn().

(3) No Events for mint() and burn() Functions:

The mint() and burn() functions don't emit events, making it harder to track the changes to the total supply and individual token balances. This can make it difficult for users or other systems to interact with the contract.

Resolution: Emit events for minting and burning activities (Mint() and Burn()) to ensure transparency and easier tracking on-chain.

Very Low / Informational / Best practices:

(1) Potential for Large Gas Fees Due to Token Minting and Burning:

Both mint() and burn() functions involve increasing or decreasing the total supply, which could result in high gas fees if not carefully optimized.

Resolution: Reassess the need for continuous minting or burning. Introduce batch operations to allow more gas-efficient transactions when minting or burning tokens.

(2) Contract Versioning:

The contract uses Solidity version 0.5.16, which is quite outdated. Newer versions of Solidity have numerous improvements in terms of security, gas efficiency, and features.

Resolution: Upgrade the Solidity version to the latest stable release (currently Solidity 0.8.x) and ensure compatibility with existing features.

(3) Unused _allowances Mapping:

The _allowances mapping and related functions (approve, allowance, transferFrom) are included in the contract but are not used as extensively as they could be, which might introduce unnecessary complexity and higher gas costs.

Resolution: Review the need for allowance functionality in this contract. If it's not needed, remove these functions and mappings to simplify the contract and reduce gas consumption.

Centralization

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

BEP20USDT.sol

- mint: Allows the owner to create `amount` tokens and assigns 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 3 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 contracts protocol, 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 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 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.

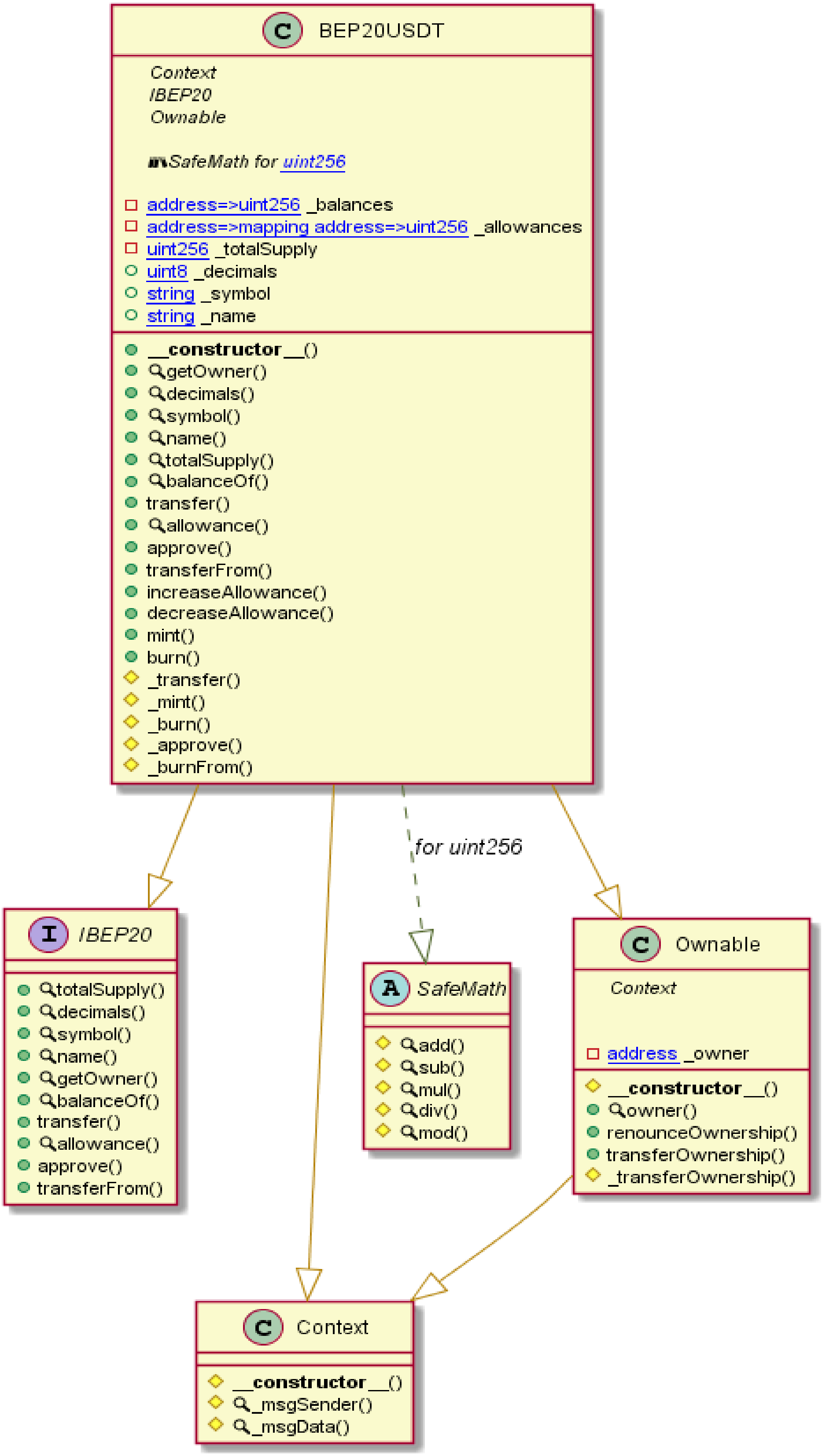
Technical Disclaimer

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 - Tether USD (USDT) Token

TetherToken Diagram



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 >> BEP20USDT.sol

```
INFO:Detectors:
BEP20USDT.allowance(address,address).owner (BEP20USDT.sol#423) shadows:
	- Ownable.owner() (BEP20USDT.sol#301-303) (function)
BEP20USDT._approve(address,address,uint256).owner (BEP20USDT.sol#586) shadows:
	- Ownable.owner() (BEP20USDT.sol#301-303) (function)
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#local-variable-shadowing
INFO:Detectors:
BEP20USDT._burnFrom(address,uint256) (BEP20USDT.sol#600-603) is never used and should
be removed
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dead-code
INFO:Detectors:
Version constraint 0.5.16 contains known severe issues
(https://solidity.readthedocs.io/en/latest/bugs.html)
	- AbiReencodingHeadOverflowWithStaticArrayCleanup
	- DirtyBytesArrayToStorage
	- NestedCalldataArrayAbiReencodingSizeValidation
	- ABIDecodeTwoDimensionalArrayMemory
	- KeccakCaching
	- EmptyByteArrayCopy
	- DynamicArrayCleanup
	- MissingEscapingInFormatting
	- ImplicitConstructorCallvalueCheck
	- TupleAssignmentMultiStackSlotComponents
	- MemoryArrayCreationOverflow
	- privateCanBeOverridden.
It is used by:
	- 0.5.16 (BEP20USDT.sol#5)
solc-0.5.16 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:
Variable BEP20USDT._decimals (BEP20USDT.sol#351) is not in mixedCase
Variable BEP20USDT._symbol (BEP20USDT.sol#352) is not in mixedCase
Variable BEP20USDT._name (BEP20USDT.sol#353) is not in mixedCase
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions
INFO:Detectors:
Redundant expression "this (BEP20USDT.sol#118)" inContext (BEP20USDT.sol#108-121)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#redundant-statements
INFO:Slither:BEP20USDT.sol analyzed (5 contracts with 93 detectors), 10 result(s) found
```


Email: audit@EtherAuthority.io

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.

BEP20USDT.sol

Gas costs:
Gas requirement of function BEP20USDT.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: 501:2:

Gas costs:
Gas requirement of function BEP20USDT.burn 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: 509:2:

ERC20:
ERC20 contract's "decimals" function should have "uint8" as return type
Pos: 375:2:

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: 176:4:

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.

BEP20USDT.sol

```
Code contains empty blocks
Pos: 27:110
Error message for require is too long
Pos: 5:199
Error message for require is too long
Pos: 5:336
Error message for require is too long
Pos: 5:528
Error message for require is too long
Pos: 5:529
Error message for require is too long
Pos: 5:565
Error message for require is too long
Pos: 5:586
Error message for require is too long
Pos: 5:587
```

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