Ether Authority

www.EtherAuthority.io
audit@etherauthority.io

SMART CONTRACT

Security Audit Report

Project: C77 Token
Platform: Binance Smart Chain
Language: Solidity
Date: December 20th, 2021

Table of contents

Email: audit@EtherAuthority.io

THIS IS SECURITY AUDIT REPORT DOCUMENT AND WHICH MAY CONTAIN INFORMATION WHICH 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

EtherAuthority was contracted by the C77 team to perform the Security audit of the C77 Token smart contract code. 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 December 20th, 2021.

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

C77 is a standard BEP20 token smart contract. This audit only considers C77 token smart contracts, and does not cover any other smart contracts in the platform.

Audit scope

Name	**Code Review and Security Analysis Report for C77 Token Smart Contract**
Platform	**BSC / Solidity**
File	C77Token.sol
File MD5 Hash	7E62543CE11C0175DF170DF290B36540
Online code	0x235C6122Ae5Bb4E42913372D1B899A8c75387AD4
Audit Date	December 20th, 2021

Email: audit@EtherAuthority.io

Claimed Smart Contract Features

Claimed Feature Detail	Our Observation
Tokenomics: • Name: C77 Token • Symbol: C77 • Decimals: 18 • Total Supply: 3 Billion	**YES, This is valid.**

Audit Summary

According to the standard audit assessment, Customer`s solidity smart contracts are **"Secured"**. This token contract does contain owner control, which does not make it fully decentralized.

We used various tools like Slither, Solhint 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 Audit overview section. General overview is presented in AS-IS section and all identified issues can be found in the Audit overview section.

We found 0 critical, 0 high, 1 medium and 0 low and some very low level issues. These issues are not critical ones.

Investors Advice: 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	Passed
	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	Passed
Code Specification	Function visibility not explicitly declared	Passed
	Var. storage location not explicitly declared	Passed
	Use keywords/functions to be deprecated	Passed
	Unused code	Passed
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 not set	Moderated
	“Short Address” Attack	Passed
	“Double Spend” Attack	Passed

Overall Audit Result: **PASSED**

Code Quality

This audit scope has 1 smart contract file. Smart contract contains Libraries, Smart contracts, inherits and Interfaces. This is a compact and well written smart contract.

The libraries in C77 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 C77 Token.

The C77 Token team has **not** provided 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 smart contracts.

Documentation

We were given a C77 Token smart contracts code in the form of a BSCscan web link.The hash of that code is mentioned above in the table.

As mentioned above, code parts are **well** commented. 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 are used in this smart contract infrastructure that are based on well known industry standard open source projects.

Apart from libraries, its functions are used in external smart contract calls.

AS-IS overview

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	Passed	No Issue
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	external	Passed	No Issue
15	burn	external	Ambiguous error message	Refer Audit Findings
16	_transfer	internal	Passed	No Issue
17	_mint	internal	Unlimited token minting	Refer Audit Findings
18	_burn	internal	Passed	No Issue
19	_approve	internal	Passed	No Issue
20	_burnFrom	internal	Unused function	Refer Audit Findings
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

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.

Email: audit@EtherAuthority.io

Audit Findings

Critical Severity

No Critical severity vulnerabilities were found.

High Severity

No High severity vulnerabilities were found.

Medium

(1) Unlimited token minting:

```
function _mint(address account, uint256 amount) internal {
  require(account != address(0), "BEP20: mint to the zero address");

  _totalSupply = _totalSupply.add(amount);
  _balances[account] = _balances[account].add(amount);
  emit Transfer(address(0), account, amount);
}
```

Token minting without any maximum limit is considered inappropriate for tokenomics.

Resolution: We recommend placing some limit on token minting.

Status: Acknowledged

Low

No low severity vulnerabilities were found.

Very Low / Informational / Best practices:

(1) Declare variables constant:

```
uint256 private _initialSupply;
uint256 private _totalSupply;
uint8 public _decimals;
string public _symbol;
string public _name;
```

These variables' values will remain unchanged.

Resolution: We suggest making them constant. It is best practice and it also saves some gas. Just add a constant keyword.

(2) Use the latest solidity version:

```
pragma solidity >=0.6.0 <0.8.0;
```

Using the latest solidity will prevent any compiler-level bugs.
Resolution: Please use 0.8.10 which is the latest version.

(3) Ambiguous error message:

```
function burn(uint256 amount) external override returns (bool) {
  require(msg.sender == owner(), "owner: wut?");
  _burn(_msgSender(), amount);
  return true;
}
```

Ambiguous error message used in burn function.
Resolution: Set proper relevant error message to identify the failure of the transaction.

(4) Unused function:

```
function _burnFrom(address account, uint256 amount) internal {
  _burn(account, amount);
  _approve(account, _msgSender(), _allowances[account][_msgSender()].sub(amount, "BEP20: burn amount exceeds allowance"));
}
```

There is _burnFrom() internal function defined but not used anywhere.
Resolution: We suggest removing unused functions from the contract code.

Centralization

This smart contract has some functions which can be executed by the Admin (Owner) only. If the admin wallet private key would be compromised, then it would create trouble. Following are Admin functions:

- mint: The Owner can create `amount` tokens and assign them to `msg.sender`, increasing the total supply.
- burn: The Owner can burn `amount` tokens and decrease the total supply.

Conclusion

We were given a contract code. And we have used all possible tests based on given objects as files. We observed some issues in the smart contracts, but they are not critical ones. So, **it's good to go to 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 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.

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

Security state of the reviewed 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 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.

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).

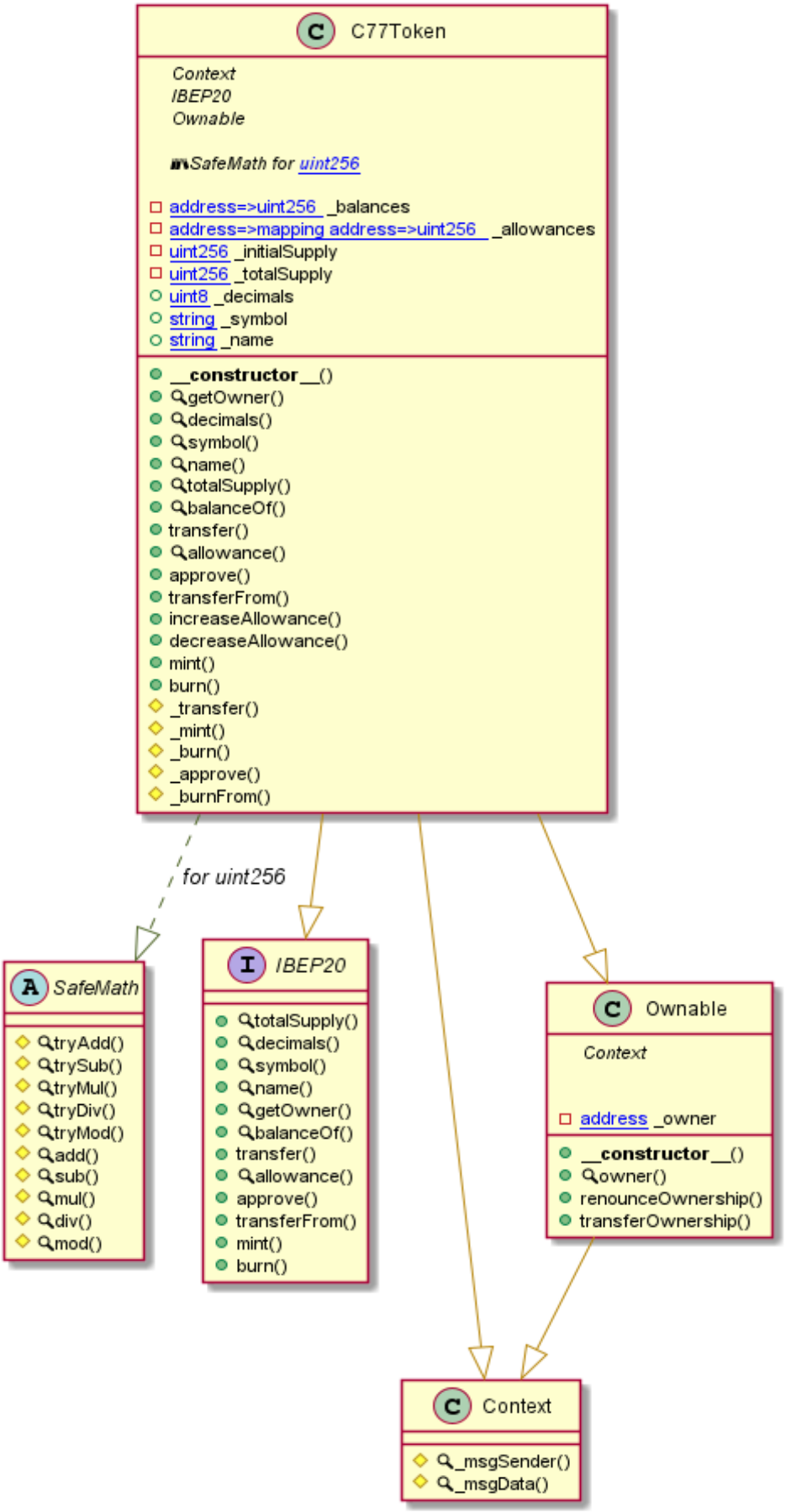
Due to the fact that the total number of test cases are unlimited, the audit makes no statements or warranties on security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bugfree status or any other statements of the contract. While we 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 - C77 Token



Email: audit@EtherAuthority.io

Slither Results Log

Slither log >> C77Token.sol

```
INFO:Detectors:
C77Token.allowance(address,address).owner (C77Token.sol#491) shadows:
        - Ownable.owner() (C77Token.sol#373-375) (function)
C77Token._approve(address,address,uint256).owner (C77Token.sol#660) shadows:
        - Ownable.owner() (C77Token.sol#373-375) (function)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#local-variable-shadowing
INFO:Detectors:
C77Token._burnFrom(address,uint256) (C77Token.sol#674-677) is never used and should be removed
Context._msgData() (C77Token.sol#348-351) is never used and should be removed
SafeMath.div(uint256,uint256) (C77Token.sol#141-144) is never used and should be removed
SafeMath.div(uint256,uint256,string) (C77Token.sol#200-207) is never used and should be removed
SafeMath.mod(uint256,uint256) (C77Token.sol#158-161) is never used and should be removed
SafeMath.mod(uint256,uint256,string) (C77Token.sol#224-231) is never used and should be removed
SafeMath.mul(uint256,uint256) (C77Token.sol#122-127) is never used and should be removed
SafeMath.tryAdd(uint256,uint256) (C77Token.sol#10-18) is never used and should be removed
SafeMath.tryDiv(uint256,uint256) (C77Token.sol#58-65) is never used and should be removed
SafeMath.tryMod(uint256,uint256) (C77Token.sol#72-79) is never used and should be removed
SafeMath.tryMul(uint256,uint256) (C77Token.sol#39-51) is never used and should be removed
SafeMath.trySub(uint256,uint256) (C77Token.sol#25-32) is never used and should be removed
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dead-code
INFO:Detectors:
Variable C77Token._decimals (C77Token.sol#420) is not in mixedCase
Variable C77Token._symbol (C77Token.sol#421) is not in mixedCase
Variable C77Token._name (C77Token.sol#422) is not in mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions
INFO:Detectors:
Redundant expression "this (C77Token.sol#349)" inContext (C77Token.sol#343-352)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#redundant-statements
INFO:Detectors:
C77Token.constructor() (C77Token.sol#424-431) uses literals with too many digits:
        - _initialSupply = 3000000000 * 10 ** 18 (C77Token.sol#425)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#too-many-digits
INFO:Detectors:
renounceOwnership() should be declared external:
        - Ownable.renounceOwnership() (C77Token.sol#392-395)
transferOwnership(address) should be declared external:
        - Ownable.transferOwnership(address) (C77Token.sol#401-408)
increaseAllowance(address,uint256) should be declared external:
```

```
increaseAllowance(address,uint256) should be declared external:
        - C77Token.increaseAllowance(address,uint256) (C77Token.sol#537-540)
decreaseAllowance(address,uint256) should be declared external:
        - C77Token.decreaseAllowance(address,uint256) (C77Token.sol#556-559)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#public-function-that-could-be-declared-external
INFO:Slither:C77Token.sol analyzed (5 contracts with 75 detectors), 23 result(s) found
INFO:Slither:Use https://crytic.io/ to get access to additional detectors and Github integration
```

Solidity Static Analysis

C77Token.sol

Gas & Economy

Gas costs:

Gas requirement of function C77Token.transferOwnership 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: 401:4:

Gas costs:

Gas requirement of function C77Token._symbol 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: 421:2:

Gas costs:

Gas requirement of function C77Token._name 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: 422:2:

ERC

ERC20:

ERC20 contract's "decimals" function should have "uint8" as return type
more
Pos: 243:2:

ERC20:

ERC20 contract's "decimals" function should have "uint8" as return type
more
Pos: 443:2:

Miscellaneous

Similar variable names:

C77Token._mint(address,uint256) : Variables have very similar names "account" and "amount". Note: Modifiers are currently not considered by this static analysis.
Pos: 621:12:

Similar variable names:

C77Token._mint(address,uint256) : Variables have very similar names "account" and "amount". Note: Modifiers are currently not considered by this static analysis.
Pos: 623:36:

Similar variable names:

C77Token._mint(address,uint256) : Variables have very similar names "account" and "amount". Note: Modifiers are currently not considered by this static analysis.
Pos: 624:14:

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.
more
Pos: 570:4:

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.
more
Pos: 583:4:

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.
more
Pos: 604:4:

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.
more
Pos: 621:4:

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.
more
Pos: 640:4:

Data truncated:

Division of integer values yields an integer value again. That means e.g. 10 / 100 = 0 instead of 0.1 since the result is an integer again. This does not hold for division of (only) literal values since those yield rational constants.
Pos: 206:15:

Solhint Linter

C77Token.sol

```
C77Token.sol:2:1: Error: Compiler version >=0.6.0 <0.8.0 does not
satisfy the r semver requirement
```

Software analysis result:

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

Email: audit@EtherAuthority.io