VANILLA CAKE (VCAKE) TOKEN SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer:	Vanillacake Team (https://vanillacake.farm)
Prepared on:	10/05/2021
Platform:	Binance Smart Chain
Language:	Solidity
Audit Type:	Standard

audit@etherauthority.io

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 PUBLIC AFTER ISSUES ARE RESOLVED.

Project file

Name	Code Review and Security Analysis Report for VanillaCake (VCAKE) Token Smart Contract
Platform	BSC / Solidity
File	VcakeToken.sol
File MD5 hash	FE9A1E895AAEB082D1091EF68E2E053C
Online Contract Code	https://bscscan.com/address/0xb86F1e870DbE29629134Cb52f568575D819ee0a9#code

Introduction

We were contracted by the VanillaCake team to perform the Security audit of the VanillaCake 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 10/05/2021.

The Audit type was Standard Audit. Which means this audit is concluded based on Standard audit scope, which is one security engineer performing audit procedure for 3 days. This document outlines all the findings as well as an AS-IS overview of the smart contract codes.

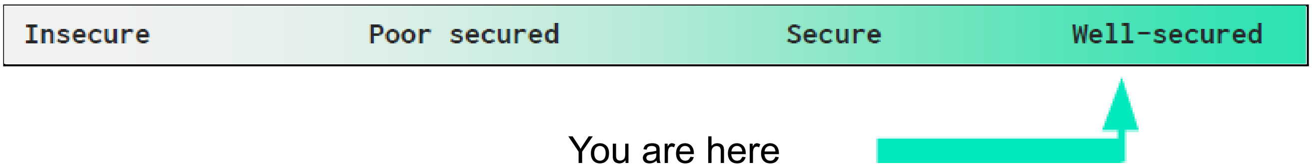
Quick Stats:

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
	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
	Other code specification issues	Passed
Gas Optimization	Assert() misuse	Passed
	High consumption ‘for/while’ loop	Moderated
	High consumption ‘storage’ storage	Passed
	“Out of Gas” Attack	Passed
Business Risk	The maximum limit for mintage not set	Not Set
	“Short Address” Attack	Passed
	“Double Spend” Attack	Passed

Overall Audit Result: PASSED WITH CONSENT

Executive Summary

According to the **standard** audit assessment, Customer`s solidity smart contract is **Well secured**.



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 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 1 medium and 1 low and some very low level issues.

Code Quality

VanillaCake Token smart contract has 1 smart contract. This smart contract also contains Libraries, Smart contract inherits and Interfaces. These are compact and well written contracts.

The libraries in the VanillaCake Token protocol 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 VanillaCake Token protocol.

The VanillaCake team has **not** provided scenario and unit test scripts, which would have helped to determine the integrity of the code in an automated way.

Overall, code parts are well commented (except some parts). Commenting can provide rich documentation for functions, return variables and more.

Documentation

We were given VanillaCake token smart contracts code in the form of a BscScan web link. The hash of that code and that web link are mentioned above in the table.

As mentioned above, most code parts are well commented. so anyone can quickly understand the programming flow as well as complex code logic. Comments are very helpful in understanding the overall architecture of the protocol.

Another source of information was its official website which provided rich information about the project architecture and tokenomics.

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 their core code blocks are written well.

Apart from libraries, VanillaCake token smart contract does not depend on any other smart contracts as external smart contract calls.

AS-IS overview

VanillaCake Token is a BEP20 standard smart contract, with other features like minting, governance, etc. Following are the main components of core smart contract.

VcakeToken.sol

(1) Interfaces

(a) IBEP20: Provides BEP20 standard functions and events.

(2) Inherited contracts

(a) BEP20: Provides BEP20 standard functions.

(b) MinterRole: Provides minting functions.

(c) Ownable: Ownership contract.

(d) Context: Context contract.

(3) Usages

(a) using SafeMath for uint256;

(b) using Address for address;

(4) Events

(a) event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);

(b) event Transfer(address indexed from, address indexed to, uint256 value);

(c) event Approval(address indexed owner, address indexed spender, uint256 value);

(d) event MinterAdded(address indexed account);

(e) event MinterRemoved(address indexed account);

(f) event DelegateChanged(address indexed delegator, address indexed fromDelegate, address indexed toDelegate);

(g) event DelegateVotesChanged(address indexed delegate, uint previousBalance, uint newBalance);

(5) Functions

Sl.	Functions	Type	Observation	Conclusion
1	name	read	Passed	No Issue
2	decimals	read	Passed	No Issue
3	symbol	read	Passed	No Issue
4	totalSupply	read	Passed	No Issue
5	balanceOf	read	Passed	No Issue
6	transfer	write	Passed	No Issue
7	allowance	read	Passed	No Issue
8	approve	write	Passed	No Issue
9	transferFrom	write	Passed	No Issue
10	increaseAllowance	write	Passed	No Issue
11	decreaseAllowance	write	Passed	No Issue
12	_transfer	internal	Passed	No Issue
13	_mint	internal	No Max minting set	Minter should mint responsibly
14	_burn	internal	Not used anywhere	No Issue
15	_approve	internal	Passed	No Issue
16	_burnFrom	internal	Not used anywhere	No Issue
17	_burn	internal	Passed	No Issue
18	isMinter	read	Passed	No Issue
19	addMinter	write	Passed	No Issue
20	removeMinter	write	Passed	No Issue
21	renounceMinter	write	Passed	No Issue
22	_addMinter	internal	Passed	No Issue
23	_removeMinter	internal	Passed	No Issue
24	mint	write	No Max minting set	Minter should mint it responsibly
25	delegates	read	Passed	No Issue
26	delegate	write	Passed	No Issue

27	delegateBySig	write	Sig was used	Keep sig secure
28	getCurrentVotes	read	Passed	No Issue
29	getPriorVotes	read	Infinite loop possibility	Keep array length limited
30	_delegate	internal	Passed	No Issue
31	_moveDelegates	internal	Passed	No Issue
32	_writeCheckpoint	internal	Passed	No Issue
33	safe32	read	Passed	No Issue
34	getChainId	read	Passed	No Issue

Severity Definitions

Risk Level	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to tokens 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 Max minting of the tokens set.

```
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);
}
```

Setting max minting for the tokens is good for the tokenomics. Since this is an owner(miner) function, the owner must take care of minting with limitation. or even better, just add a max minting limit.

Fix: We got confirmation from the VanillaCake token team that tokens will be minted by masterChef contract as well as token sale contract. And minting will be done as needed by those contracts. So this issue is acknowledged.

Low

(1) Infinite loop possibility

```
        uint32 lower = 0;
        uint32 upper = nCheckpoints - 1;
        while (upper > lower) {
            uint32 center = upper - (upper - lower) / 2; // ceil, avoiding overflow
            Checkpoint memory cp = checkpoints[account][center];
            if (cp.fromBlock == blockNumber) {
                return cp.votes;
            } else if (cp.fromBlock < blockNumber) {
                lower = center;
            } else {
                upper = center - 1;
            }
        }
        return checkpoints[account][lower].votes;
    }
```

If there are so many nCheckpoints, then this logic will fail. Because it might hit the block's gas limit. If there are very limited nCheckpoints, then this will work.

Fix: This issue is acknowledged.

Very Low / Discussion / Best practices:

(1) Solidity Version can be fixed

```
pragma solidity >=0.4.0;
```

Use the latest solidity version while contract deployment to prevent any compiler version level bugs.

Solution: This issue is acknowledged.

(2) Any minter can add/remove any other minters. So, the private key of those minters should be kept secret. Because if that would be compromised, then it will put the smart contract's fate into hands of the attacker.

(3) Approve of ERC20 standard.
This can be used to front run. From the client side, only use this function to change the allowed amount to 0 or from 0 (wait till transaction is mined and approved).

(4) Ownership functions not used

There are no owner only functions.
Solution: It's better to remove ownership contract to make the code clean. Or just renounce the ownership, so users can feel safe thinking owner also can not do any manipulations.

(5) All functions which are not called internally, must be declared as external. It is more efficient as sometimes it saves some gas.
https://ethereum.stackexchange.com/questions/19380/external-vs-public-best-practices

Centralization

This smart contract has some functions which can be executed by Admin (Minter) only. If the admin wallet private key would be compromised, then it puts this smart contract in the hands of an attacker. Following are Admin functions:

- addMinter
- removeMinter
- mint

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 contract and those are fixed/ackolenged in the smart contract. **So it is good to go for the production.**

Since possible test cases can be unlimited for such extensive smart contract 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 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 "**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.

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).

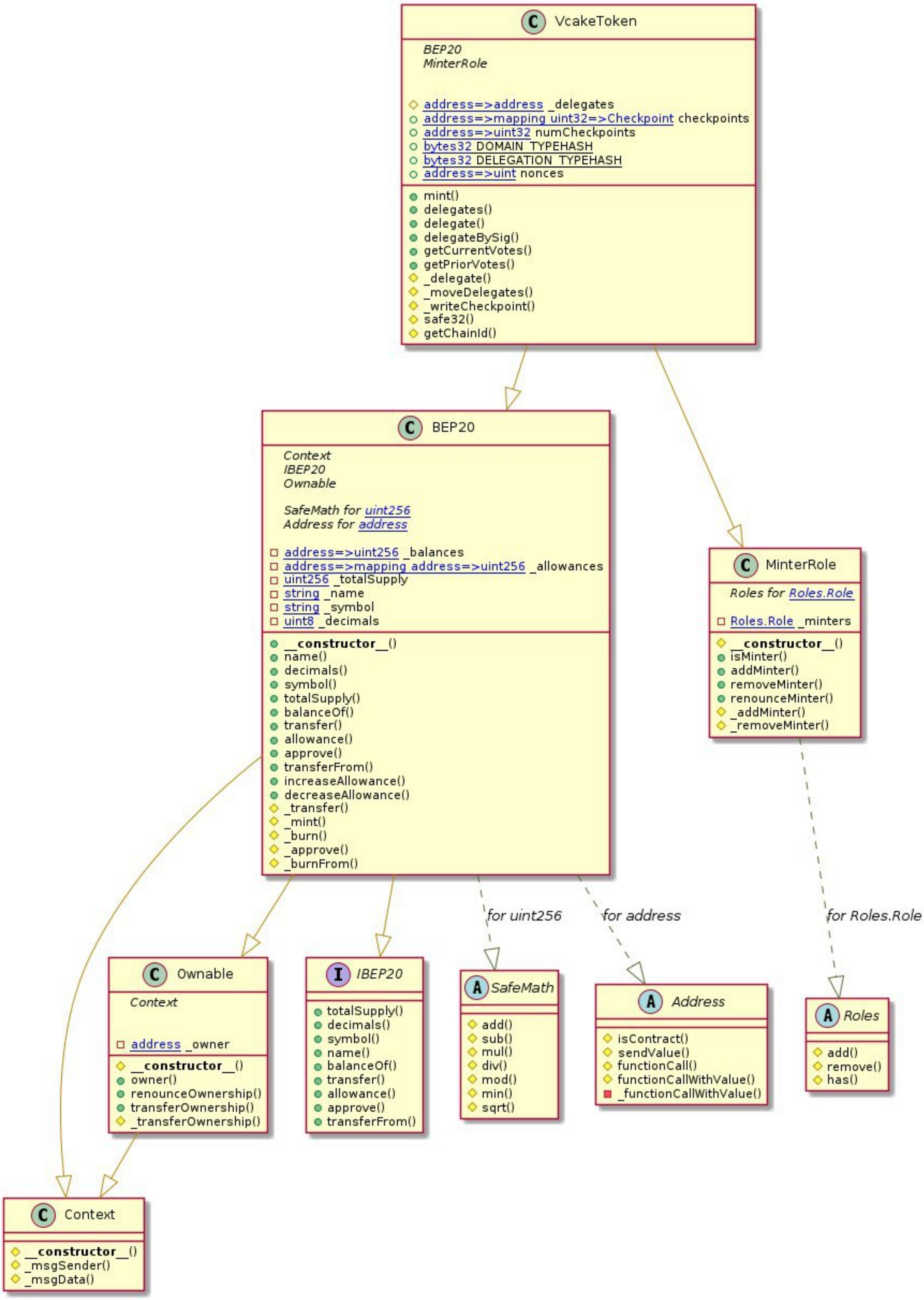
Due to the fact that the total number of test cases are unlimited, so the audit makes no statements or warranties on security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bugfree status or any other statements of the contract. While 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 to conduct a bug bounty program to confirm the high level of security of this smart contract.

Technical Disclaimer

Smart contracts are deployed and executed on 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 - Vcake Token



Slither Results Log

INFO:Detectors:
VcakeToken._writeCheckpoint(address,uint32,uint256,uint256)
(VcakeToken.sol#1161-1179) uses
a dangerous strict equality:
- nCheckpoints > 0 && checkpoints[delegatee][nCheckpoints - 1].fromBlock ==
blockNumber (VcakeToken.sol#1171)
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#dangerous-strictequalities
INFO:Detectors:
BEP20.constructor(string,string).name (VcakeToken.sol#604) shadows:
- BEP20.name() (VcakeToken.sol#620-622) (function)
- IBEP20.name() (VcakeToken.sol#129) (function)
BEP20.constructor(string,string).symbol (VcakeToken.sol#604) shadows:
- BEP20.symbol() (VcakeToken.sol#634-636) (function)
- IBEP20.symbol() (VcakeToken.sol#124) (function)
BEP20.allowance(address,address).owner (VcakeToken.sol#668) shadows:
- Ownable.owner() (VcakeToken.sol#65-67) (function)
BEP20._approve(address,address,uint256).owner (VcakeToken.sol#840) shadows:
- Ownable.owner() (VcakeToken.sol#65-67) (function)
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#local-variable-shadowing
INFO:Detectors:
VcakeToken.delegateBySig(address,uint256,uint256,uint8,bytes32,bytes32)
(VcakeToken.sol#1027-1068) uses timestamp for comparisons
Dangerous comparisons:
- require(bool,string)(now <= expiry,TOKEN::delegateBySig: signature expired)
(VcakeToken.sol#1066)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#block-timestamp
INFO:Detectors:
Address.isContract(address) (VcakeToken.sol#417-428) uses assembly
- INLINE ASM (VcakeToken.sol#424-426)
Address._functionCallWithValue(address,bytes,uint256,string) (VcakeToken.sol#525-551)
uses
assembly
- INLINE ASM (VcakeToken.sol#543-546)
VcakeToken.getChainId() (VcakeToken.sol#1186-1190) uses assembly
- INLINE ASM (VcakeToken.sol#1188)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#assembly-usage
INFO:Detectors:
Different versions of Solidity is used in :
- Version used: ['0.6.12', '>=0.4.0']
- >=0.4.0 (VcakeToken.sol#7)
- 0.6.12 (VcakeToken.sol#867)
- 0.6.12 (VcakeToken.sol#905)
- 0.6.12 (VcakeToken.sol#952)
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#different-pragmadirectives-are-used
INFO:Detectors:
Pragma version>=0.4.0 (VcakeToken.sol#7) allows old versions
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-ofsolidity
INFO:Detectors:
Low level call in Address.sendValue(address,uint256) (VcakeToken.sol#446-452):
- (success) = recipient.call{value: amount}() (VcakeToken.sol#450)
Low level call in Address._functionCallWithValue(address,bytes,uint256,string)
(VcakeToken.sol#525-551):

- (success,returndata) = target.call{value: weiValue}(data) (VcakeToken.sol#534)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#low-level-calls
INFO:Detectors:
Parameter VcakeToken.mint(address,uint256)._to (VcakeToken.sol#957) is not in mixedCase
Parameter VcakeToken.mint(address,uint256)._amount (VcakeToken.sol#957) is not in mixedCase
Variable VcakeToken._delegates (VcakeToken.sol#969) is not in mixedCase
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-soliditynamin g-
conventions
INFO:Detectors:
Redundant expression "this (VcakeToken.sol#29)" inContext (VcakeToken.sol#19-32)
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#redundant-statements
INFO:Detectors:
owner() should be declared external:
- Ownable.owner() (VcakeToken.sol#65-67)
renounceOwnership() should be declared external:
- Ownable.renounceOwnership() (VcakeToken.sol#84-87)
transferOwnership(address) should be declared external:
- Ownable.transferOwnership(address) (VcakeToken.sol#93-95)
decimals() should be declared external:
- BEP20.decimals() (VcakeToken.sol#627-629)
symbol() should be declared external:
- BEP20.symbol() (VcakeToken.sol#634-636)
totalSupply() should be declared external:
- BEP20.totalSupply() (VcakeToken.sol#641-643)
transfer(address,uint256) should be declared external:
- BEP20.transfer(address,uint256) (VcakeToken.sol#660-663)
allowance(address,address) should be declared external:
- BEP20.allowance(address,address) (VcakeToken.sol#668-670)
approve(address,uint256) should be declared external:
- BEP20.approve(address,uint256) (VcakeToken.sol#679-682)
transferFrom(address,address,uint256) should be declared external:
- BEP20.transferFrom(address,address,uint256) (VcakeToken.sol#696-708)
increaseAllowance(address,uint256) should be declared external:
- BEP20.increaseAllowance(address,uint256) (VcakeToken.sol#722-725)
decreaseAllowance(address,uint256) should be declared external:
- BEP20.decreaseAllowance(address,uint256) (VcakeToken.sol#741-748)
addMinter(address) should be declared external:
- MinterRole.addMinter(address) (VcakeToken.sol#928-930)
removeMinter(address) should be declared external:
- MinterRole.removeMinter(address) (VcakeToken.sol#932-934)
renounceMinter() should be declared external:
- MinterRole.renounceMinter() (VcakeToken.sol#936-938)
mint(address,uint256) should be declared external:
- VcakeToken.mint(address,uint256) (VcakeToken.sol#957-960)
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#public-function-thatcould-be-declared-external
INFO:Slither:VcakeToken.sol analyzed (9 contracts with 72 detectors), 33 result(s) found
INFO:Slither:Use https://crytic.io/ to get access to additional detectors and Github integration

Email: audit@EtherAuthority.io