CodiSecure

Manual Smart Contract Audit

Low-Risk	Medium-Risk	High-Risk
low-risk code	medium-risk code	high-risk code

Contract Address

0x3b556DEe236E220DEdb8549a732580CdAE879df6 BSC

Disclaimer

CodiSecure is not responsible if a project turns out to be a scam, rug-pull or honeypot. We only provide a detailed analysis for your own research.

CodiSecure is not responsible for any financial losses. Nothing in this contract audit is financial advice, please do your own research.

The information provided in this audit is for informational purposes only and should not be considered investment advice. Coinsult does not endorse, recommend, support or suggest to invest in any project.

CodiSecure can not be held responsible for when a project turns out to be a rug-pull, honeypot or scam.

Tokenomics

- BSC

Source Code

- CodiSecure was comissioned by KIaToken to perform an audit based on the following smart contract:

- https://www.bscscan.com/token/0x3b556DEe236E220DEdb8549a732580CdAE879df6

- BSC NETWORK

About the Project and Company

KiaToken

Website: https://kiahelps.com/

Manual Code Review

In this audit report we will highlight all these issues:

- Low-Risk

  Deal with low-risk code issues

- Medium-Risk

  Deal with medium-risk code issues

- High-Risk

  Deal with high-risk code issues f

The detailed report continues on the next page…

Auditing Strategy and Techniques Applied

Throughout the review process, care was taken to evaluate the repository for security-related issues, code quality, and adherence to specification and best practices. To do so, reviewed line-by-line by our team of expert pentesters and smart contract developers, documenting any issues as there were discovered.

Methodology

The auditing process follows a routine series of steps:

- Code review that includes the following:
  - Review of the specifications, sources, and instructions provided to CodiSecure to make sure we understand the size, scope, and functionality of the smart contract.
  - Manual review of code, which is the process of reading source code line-by-line in an attempt to identify potential vulnerabilities.
  - Comparison to specification, which is the process of checking whether the code does what the specifications, sources, and instructions provided to CodiSecure describe.
- Testing and automated analysis that includes the following:
  - Test coverage analysis, which is the process of determining whether the test cases are actually covering the code and how much code is exercised when we run those test cases.
  - Symbolic execution, which is analysing a program to determine what inputs causes each part of a program to execute.
- Best practices review, which is a review of the smart contracts to improve efficiency, effectiveness, clarify, maintainability, security, and control based on the established industry and academic practices, recommendations, and research.
- Specific, itemized, actionable recommendations to help you take steps to secure your smart contracts.

Contract Manual Code

```
contract KIATOKEN  is PausableToken {

    string public name;
    string public symbol;
    uint public decimals;
event Mint(address indexed from, address indexed to, uint256 value);
event Burn(address indexed burner, uint256 value);

constructor(string memory _name, string memory _symbol, uint256 _decimals,
uint256 _supply, address tokenOwner)

public {        name = _name;
      symbol = _symbol;
      decimals = _decimals;
      totalSupply = _supply * 10**_decimals;
      balances[tokenOwner] = totalSupply;
      owner = tokenOwner;
      emit Transfer(address(0), tokenOwner, totalSupply);    }

function burn(uint256 _value) public (_burn(msg.sender, _value);        }

function _burn(address _who, uint256 _value) internal {
                require(_value <= balances[_who]);
                balances[_who] = balances[_who].sub(_value);
                totalSupply = totalSupply.sub(_value);
                emit Burn(_who, _value);
emit Transfer(_who, address(0), _value);         }

 function mint(address account, uint256 amount) onlyOwner public {

                totalSupply = totalSupply.add(amount);
                balances[account] = balances[account].add(amount);
                emit Mint(address(0), account, amount);
                emit Transfer(address(0), account, amount);
  }    }
```

Tested Contract Files

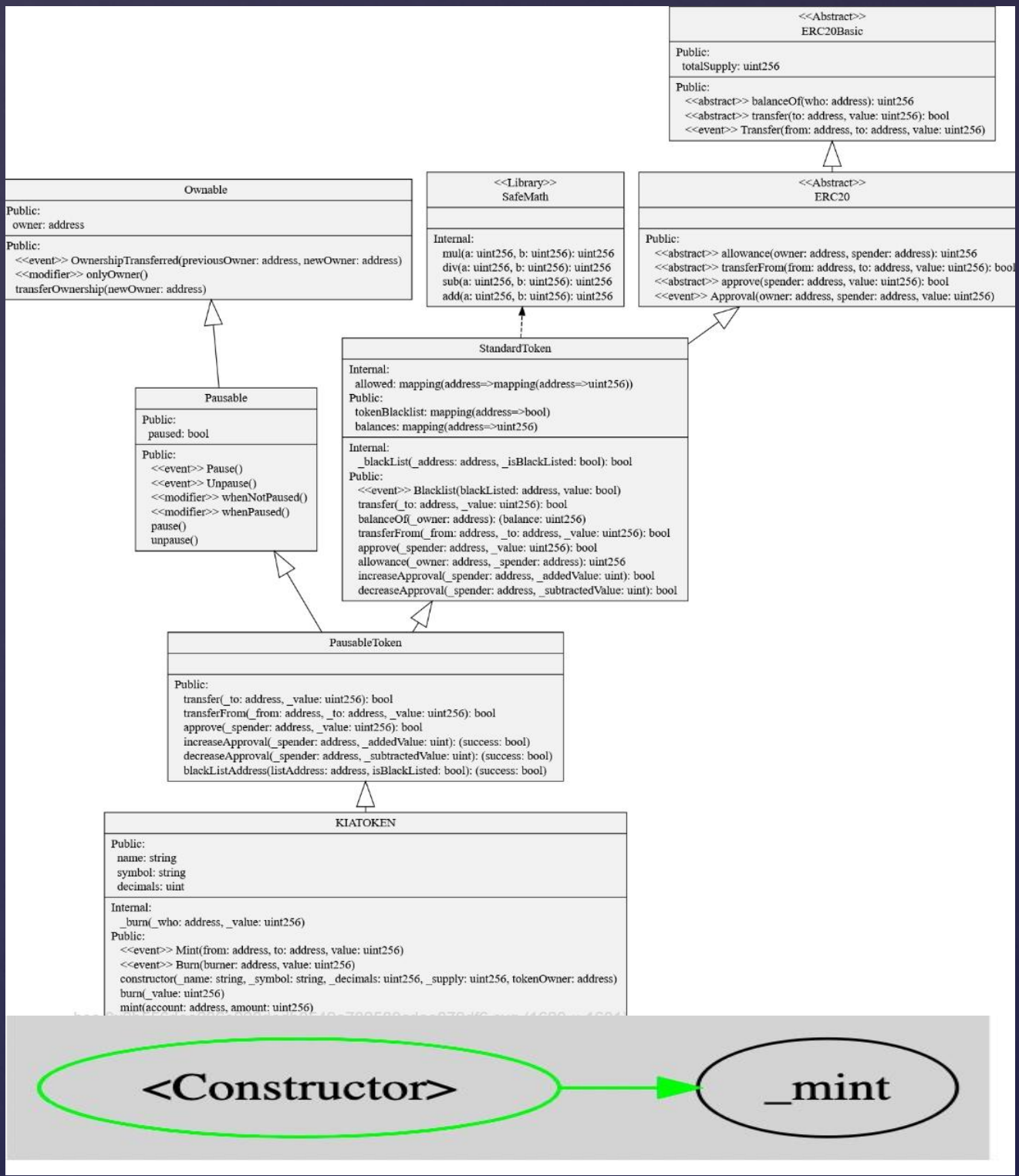
The following are the MD5 hashes of the reviewed files. A file with a different MD5 hash has been modified, intentionally or otherwise,
after the security review. You are cautioned that a different MD5 hash could be (but is not necessarily) an indication of a changed
condition or potential vulnerability that was not within the scope of the review

File	Fingerprint (MD5
Contract/kiatoken.sol	723bcb577169ba64652075de5924383c

Used Code from other Frameworks/Smart Contracts (direct imports)

Dependency / Import Path	Source
@openzeppelin/contracts/token/ERC20/ERC20.sol	https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v4.5.0/contracts/token/ERC20/ERC20.so

Metrics


<<Abstract>>
ERC20Basic
Public:
totalSupply: uint256
Public:
<<abstract>> balanceOf(who: address): uint256
<<abstract>> transfer(to: address, value: uint256): bool
<<event>> Transfer(from: address, to: address, value: uint256)
Ownable
Public:
owner: address
Public:
<<event>> OwnershipTransferred(previousOwner: address, newOwner: address)
<<modifier>> onlyOwner()
transferOwnership(newOwner: address)
<<Library>>
SafeMath
Internal:
mul(a: uint256, b: uint256): uint256
div(a: uint256, b: uint256): uint256
sub(a: uint256, b: uint256): uint256
add(a: uint256, b: uint256): uint256
<<Abstract>>
ERC20
Public:
<<abstract>> allowance(owner: address, spender: address): uint256
<<abstract>> transferFrom(from: address, to: address, value: uint256): bool
<<abstract>> approve(spender: address, value: uint256): bool
<<event>> Approval(owner: address, spender: address, value: uint256)
StandardToken
Internal:
allowed: mapping(address=>mapping(address=>uint256))
Public:
tokenBlacklist: mapping(address=>bool)
balances: mapping(address=>uint256)
Internal:
_blackList(_address: address, _isBlackListed: bool): bool
Public:
<<event>> Blacklist(blackListed: address, value: bool)
transfer(_to: address, _value: uint256): bool
balanceOf(_owner: address): (balance: uint256)
transferFrom(_from: address, _to: address, _value: uint256): bool
approve(_spender: address, _value: uint256): bool
allowance(_owner: address, _spender: address): uint256
increaseApproval(_spender: address, _addedValue: uint): bool
decreaseApproval(_spender: address, _subtractedValue: uint): bool
Pausable
Public:
paused: bool
Public:
<<event>> Pause()
<<event>> Unpause()
<<modifier>> whenNotPaused()
<<modifier>> whenPaused()
pause()
unpause()
PausableToken
Public:
transfer(_to: address, _value: uint256): bool
transferFrom(_from: address, _to: address, _value: uint256): bool
approve(_spender: address, _value: uint256): bool
increaseApproval(_spender: address, _addedValue: uint): (success: bool)
decreaseApproval(_spender: address, _subtractedValue: uint): (success: bool)
blackListAddress(listAddress: address, isBlackListed: bool): (success: bool)
KIATOKEN
Public:
name: string
symbol: string
decimals: uint
Internal:
_burn(_who: address, _value: uint256)
Public:
<<event>> Mint(from: address, to: address, value: uint256)
<<event>> Burn(burner: address, value: uint256)
constructor(_name: string, _symbol: string, _decimals: uint256, _supply: uint256, tokenOwner: address)
burn(_value: uint256)
mint(account: address, amount: uint256)
<Constructor>
_mint

⚜ Public	⚜ Payable		
0	0		
External	Internal	Pure	View
0	1	0	0

Metrics / Capabilities

Total	⚜ Public
0	0

Exposed Functions	Experimental Features	StateVariables		Has Destroyable Contracts

Solidity Versions observed	Can Receive Funds	Uses Assembly

Transfers ETH	New/Create/Create2

Low-Level Calls	DelegateCall	Uses Hash Functions	ECRecover

Exposed Functions

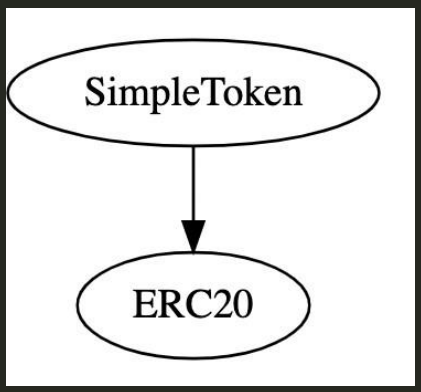
This section lists functions that are explicitly declared public or payable. Please note that getter methods for public stateVars are not included.

Contract Snapshot

Following contracts with the direct imports has been tested: kiatoken.sol
The audit team put forward the following assumption regarding the security and usage of the contract:

The contract is using the ERC-20 token standard
Owner/Deployer cannot mint new tokens
Owner/Deployer cannot burn or lock user funds
Owner/Deployer cannot pause the contract
The smart contract is coded according to the newest standards and in a secure way

The main goal of this audit was to verify these claims. The auditors can provide additional feedback on the code upon the client's request.



Manual and Automated Vulnerability Test

CRITICAL ISSUES

During the audit, Chainsulting's experts found **no Critical issues** in the code of the smart contract.

HIGH ISSUES

During the audit, Chainsulting's experts found **no High issues** in the code of the smart contract.

MEDIUM ISSUES

During the audit, Chainsulting's experts found **no Medium issues** in the code of the smart contract.

LOW ISSUES

During the audit, Chainsulting's experts found **no Low issues** in the code of the smart contract.

INFORMATIONAL ISSUES

During the audit, Chainsulting's experts found **no Informational issues** in the code of the smart contract.

SWC Attacks

ID	Title		Test Result
SWC-131	Presence of unused variables	CWE-1164: Irrelevant Code	
SWC-130	Right-To-Left-Override control character (U+202E)	CWE-451: User Interface (UI) Misrepresentation of Critical Information	
SWC-129	Typographical Error	CWE-480: Use of Incorrect Operator	
SWC-128	DoS With Block Gas Limit	CWE-400: Uncontrolled Resource Consumption	
SWC-127	Arbitrary Jump with Function TypeVariable	CWE-695: Use of Low-Level Functionality	
SWC-125	Incorrect Inheritance Order	CWE-696: Incorrect Behavior Order	
SWC-124	Write to Arbitrary Storage Location	CWE-123: Write-what-where Condition	
SWC-123	Requirement Violation	CWE-573: Improper Following of Specification by Caller	

ID	Title		Test Result
SWC-113	DoS with Failed Call	CWE-703: Improper Check or Handling of Exceptional Conditions	
SWC-112	Delegatecall to Untrusted Callee	CWE-829: Inclusion of Functionality from Untrusted Control Sphere	
SWC-111	Use of Deprecated Solidity Functions	CWE-477: Use of Obsolete Function	
SWC-110	Assert Violation	CWE-670: Always-Incorrect Control Flow Implementation	
SWC-109	Uninitialized Storage Pointer	CWE-824: Access of Uninitialized Pointer	
SWC-108	State Variable Default Visibility	CWE-710: Improper Adherence to Coding Standards	
SWC-107	Reentrancy	CWE-841: Improper Enforcement of Behavioral Workflow	
SWC-106	Unprotected SELFDESTRUCT Instruction	CWE-284: Improper Access Control	
SWC-105	Unprotected Ether Withdrawal	CWE-284: Improper Access Control	
SWC-104	Unchecked Call Return Value	CWE-252: Unchecked Return Value	

Owner privileges

- Verify Claims
  The contract is using the ERC-20 token standard
  Status: tested and verified
  Owner/Deployer cannot mint new tokens
  Status: tested and verified
  Owner/Deployer cannot burn or lock user funds
  Status: tested and verified
  Owner/Deployer cannot pause the contract
  Status: tested and verified
  The smart contract is coded according to the newest standards and in a secure way.
  Status: tested and verified

Executive Summary

Two (2) independent SkillSCodified experts performed an unbiased and isolated audit of the smart contract codebase. The final debriefs
The overall code quality is good and not overloaded with unnecessary functions, these is greatly
benefiting the security of the contract. It correctly implemented widely used and reviewed contracts from OpenZeppelin.
The main goal of the audit was to verify the claims regarding the security of the smart contract and the claims inside the scope of work.
During the audit, no issues were found after the manual and automated security testing.

Deployed Smart Contract

VERIFIED

https://www.bscscan.com/address/0x3b556DEe236E220DEdb8549a732580CdAE879df6#code