Ether Authority

www.EtherAuthority.io
audit@etherauthority.io

SMART CONTRACT

Security Audit Report

Project: KuCoin Token
Website: kucoin.com
Platform: Ethereum
Language: Solidity
Date: February 12th, 2024

Table of contents

`

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.

Introduction

As part of EtherAuthority's community smart contracts audit initiatives, the KuCoin Token smart contract from kucoin.com was audited extensively. 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 February 12th, 2024.

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 `KCSToken` contract is an ERC20 token with the following additional functionalities:
  - **Burnable Tokens:** Allows tokens to be burned, reducing the total supply.
  - **Blacklistable Addresses:** Maintains a blacklist of addresses that are restricted from transferring or receiving tokens.
  - **Custom Decimals:** Sets the decimal places to 6.
- This contract is useful for scenarios where certain addresses need to be restricted from participating in token transfers and where there is a need to burn tokens to reduce the total supply.

Audit scope

Name	Code Review and Security Analysis Report for KuCoin Token Smart Contract
Platform	Ethereum
Language	Solidity
File	KCSToken.sol
Smart Contract Code	0xf34960d9d60be18cc1d5afc1a6f012a723a28811
Audit Date	February 12th, 2024

Claimed Smart Contract Features

Claimed Feature Detail	Our Observation
Tokenomics: • Name: KuCoin Token • Symbol: KCS • Decimals: 6	**YES, This is valid.**
Ownership Control: • Update the blacklister address. • Update completed status value by the contract's owner. • The current owner can transfer the ownership. • The owner can renounce ownership.	**YES, This is valid.** **We suggest renouncing ownership once the ownership functions are not needed. This is to make the smart contract 100% decentralized.**
Blacklister Control: • Update account address to blacklist.	**YES, This is valid.**

Audit Summary

According to the standard audit assessment, the Customer`s solidity-based 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 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, 0 low, and 6 very low level issues.

Investor 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	The solidity version is not specified	Passed
	The solidity version is too old	Moderated
	Integer overflow/underflow	Passed
	Function input parameters lack 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	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

Business Risk Analysis

Category	Result
Buy Tax	0%
Sell Tax	0%
Cannot Buy	No
Cannot Sell	No
Max Tax	0%
Modify Tax	No
Fee Check	Not Detected
Is Honeypot	Not Detected
Trading Cooldown	Not Detected
Can Pause Trade?	Not Detected
Pause Transfer?	Not Detected
Max Tax?	No
Is it Anti-whale?	Not Detected
Is Anti-bot?	Not Detected
Is it a Blacklist?	Yes
Blacklist Check	Yes
Can Mint?	No
Is it a Proxy?	No
Can Take Ownership?	Yes
Hidden Owner?	Not Detected
Self Destruction?	Not Detected
Auditor Confidence	High

Overall Audit Result: PASSED

Code Quality

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

The libraries in KuCoin 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 KuCoin 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 not well commented on in the smart contracts. Ethereum's NatSpec commenting style is recommended.

Documentation

We were given a KuCoin Token smart contract code in the form of an Etherscan web link.

As mentioned above, code parts are not well commented on. but 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 are 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

Functions

Sl.	Functions	Type	Observation	Conclusion
1	constructor	write	Passed	No Issue
2	onlyBlacklister	modifier	Missing required error message	Refer Audit Findings
3	notBlacklisted	modifier	Unused code, Boolean equal, Missing required error message	Refer Audit Findings
4	isBlacklisted	read	Passed	No Issue
5	blacklist	write	Centralization	Refer Audit Findings
6	unBlacklist	write	Centralization	Refer Audit Findings
7	updateBlacklister	write	Centralization	Refer Audit Findings
8	burn	write	Passed	No Issue
9	burnFrom	write	Passed	No Issue
10	name	read	Passed	No Issue
11	symbol	read	Passed	No Issue
12	decimals	read	Passed	No Issue
13	totalSupply	read	Passed	No Issue
14	balanceOf	read	Passed	No Issue
15	transfer	write	Passed	No Issue
16	allowance	read	Passed	No Issue
17	approve	write	Passed	No Issue
18	transferFrom	write	Passed	No Issue
19	increaseAllowance	write	Passed	No Issue
20	decreaseAllowance	write	Passed	No Issue
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	_setupDecimals	internal	Passed	No Issue
26	_beforeTokenTransfer	internal	Passed	No Issue
27	owner	read	Passed	No Issue
28	onlyOwner	modifier	Passed	No Issue
29	renounceOwnership	write	Centralization	Refer Audit Findings
30	transferOwnership	write	Centralization	Refer Audit Findings
31	_msgSender	internal	Passed	No Issue
32	_msgData	internal	Passed	No Issue
33	restricted	modifier	Passed	No Issue
34	setCompleted	write	restricted	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 lose
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.

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

No Low Severity vulnerabilities were found.

Very Low / Informational / Best practices:

(1) Use the latest solidity version:

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

Using the latest solidity will prevent any compiler-level bugs.

Resolution: We suggest using the latest solidity compiler version.

(2) Unused code:

These functions are not used in contracts.

Blacklistable.sol

- notBlacklisted

Resolution: Remove the notBlacklisted modifier function from the contract.

(3) Boolean equal:

```
/**
 * @dev Throws if argument account is blacklisted
 * @param _account The address to check
*/
modifier notBlacklisted(address _account) {
    require(blacklisted[_account] == false);
    _;
}
```

Boolean constants can be used directly and do not need to be compared to true or false.

Resolution: Remove the equality to the boolean constant.

require(blacklisted[_account])

(4) Unlocked compiler version:

The contract has an unlocked compiler version. An unlocked compiler version in the source code of the contract permits the user to compile it at or above a particular version. This, in turn, leads to differences in the generated bytecode between compilations due to differing compiler version numbers. This can lead to ambiguity when debugging as compiler-specific bugs may occur in the codebase that would be hard to identify over a span of multiple compiler versions rather than a specific one.

Resolution: We suggest that the compiler version is instead locked at the lowest version possible that the contract can be compiled at.

(5) Centralization:

onlyBlacklister has owner authority of the following function:

- blacklist()
- unBlacklist()
- updateBlacklister()

In Blacklistable.sol file onlyOwner has owner authority of the following functions:

- updateBlacklister()
- renounceOwnership()
- transferOwnership()

Resolution: We suggest carefully managing these account's private keys to avoid any potential risks of being hacked. In general, we strongly recommend centralized privileges or roles in the protocol to be improved via a decentralized mechanism or smart-contract-based accounts with enhanced security practices.

(6) Missing required error message:

```
modifier onlyBlacklister() {
    require(msg.sender == blacklister);
    _;
}
```

```
modifier notBlacklisted(address _account) {
    require(blacklisted[_account] == false);
    _;
}
```

There is an error message required.

Resolution: We suggest setting relevant error messages to identify the failure of the transaction.

Centralization

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

Blacklistable.sol

- blacklist: Adds account to blacklist by only blacklister owner.
- unBlacklist: Removes account from blacklist by only blacklister owner.
- updateBlacklister: Update blacklister address by only owner.

Migrations.sol

- setCompleted: Update completed status value by the contract's owner.

Ownable.sol

- renounceOwnership: Deleting ownership will leave the contract without an owner, removing any owner-only functionality.
- transferOwnership: Current owner can transfer ownership of the contract to a new account.

To make the smart contract 100% decentralized, we suggest renouncing ownership in the smart contract once its function is completed.

Conclusion

We were given a contract code in the form of Etherscan web links. And we have used all possible tests based on given objects as files. We observed 6 Informational issues in the smart contracts. but those are not critical. So, **it's good to go for the 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 the 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 white box 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 to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, and 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).

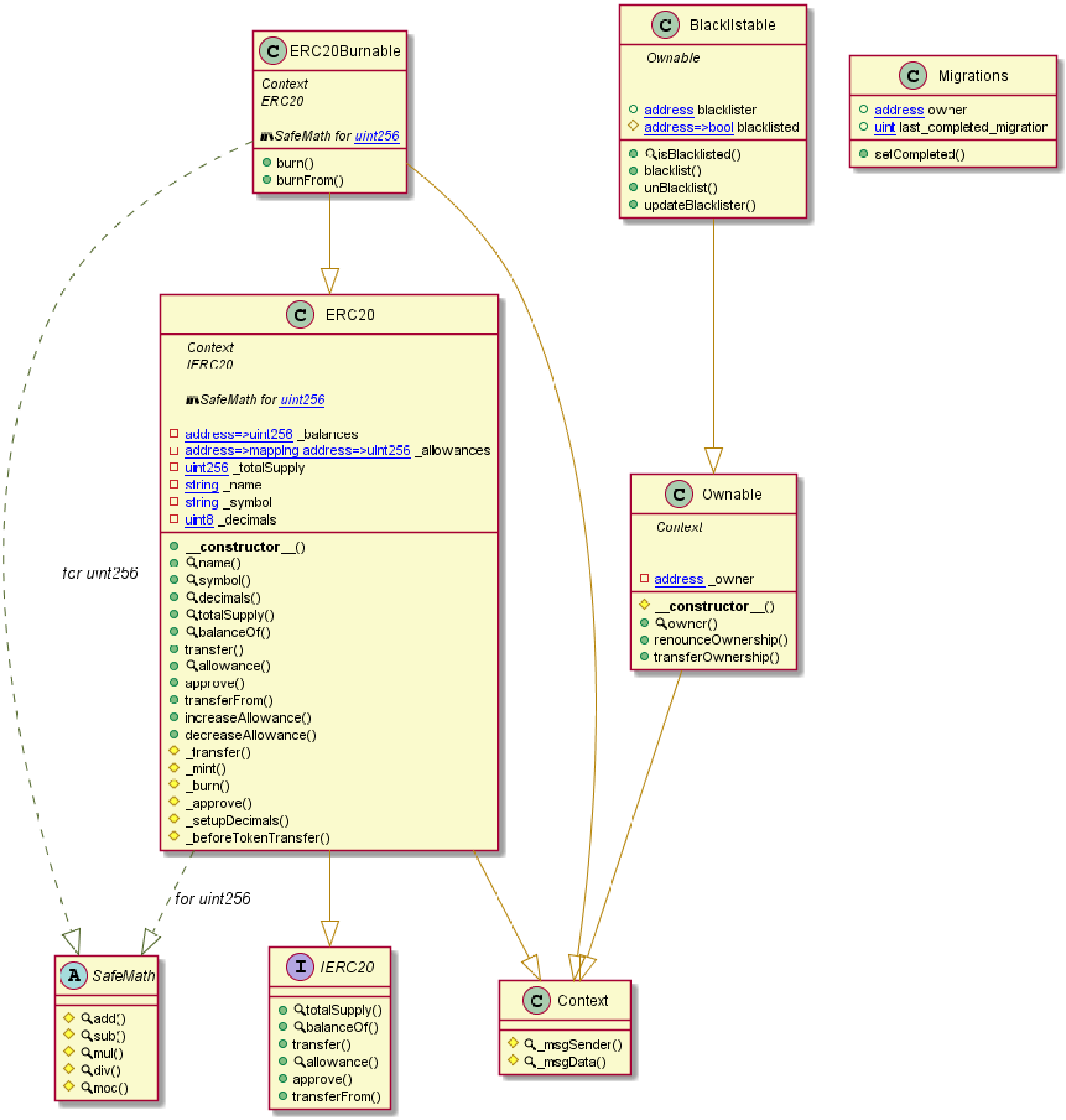
Due to the fact that the total number of test cases is unlimited, the audit makes no statements or warranties on the security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. While 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 the explicit security of the audited smart contracts.

Appendix

Code Flow Diagram - KuCoin Token



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 >> KCSToken.sol

```
INFO:Detectors:
Blacklistable.notBlacklisted(address) (KCSToken.sol#633-636) compares to a boolean constant:
        -require(bool)(blacklisted[_account] == false) (KCSToken.sol#634)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#boolean-equality
INFO:Detectors:
Context._msgData() (KCSToken.sol#245-249) is never used and should be removed
ERC20._mint(address,uint256) (KCSToken.sol#501-509) is never used and should be removed
ERC20._setupDecimals(uint8) (KCSToken.sol#560-562) is never used and should be removed
SafeMath.div(uint256,uint256) (KCSToken.sol#164-166) is never used and should be removed
SafeMath.div(uint256,uint256,string) (KCSToken.sol#180-186) is never used and should be removed
SafeMath.mod(uint256,uint256) (KCSToken.sol#200-202) is never used and should be removed
SafeMath.mod(uint256,uint256,string) (KCSToken.sol#216-219) is never used and should be removed
SafeMath.mul(uint256,uint256) (KCSToken.sol#138-150) is never used and should be removed
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dead-code
INFO:Detectors:
Pragma version>=0.6.0<0.8.0 (KCSToken.sol#3) is too complex
solc-0.7.6 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
```

```
INFO:Detectors:
Variable Migrations.last_completed_migration (KCSToken.sol#225) is not in mixedCase
Parameter Blacklistable.isBlacklisted(address)._account (KCSToken.sol#642) is not in mixedCase
Parameter Blacklistable.blacklist(address)._account (KCSToken.sol#650) is not in mixedCase
Parameter Blacklistable.unBlacklist(address)._account (KCSToken.sol#659) is not in mixedCase
Parameter Blacklistable.updateBlacklister(address)._newBlacklister (KCSToken.sol#664) is not in mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions
INFO:Detectors:
Redundant expression "this (KCSToken.sol#246)" inContext (KCSToken.sol#240-250)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#redundant-statements
INFO:Detectors:
Variable Blacklistable.blacklisted (KCSToken.sol#615) is too similar to Blacklistable.blacklister (KCSToken.sol#614)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#variable-names-too-similar
INFO:Detectors:
Migrations.owner (KCSToken.sol#224) should be immutable
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#state-variables-that-could-be-declared-immutabl
e
INFO:Slither:KCSToken.sol analyzed (8 contracts with 93 detectors), 19 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.

KCSToken.sol

Gas costs:

Gas requirement of function Migrations.setCompleted 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: 235:4:

Constant/View/Pure functions:

ERC20._beforeTokenTransfer(address,address,uint256) : Potentially should be constant/view/pure but is not. Note: Modifiers are currently not considered by this static analysis.
more
Pos: 578:4:

Similar variable names:

Blacklistable.blacklist(address) : Variables have very similar names "blacklister" and "blacklisted". Note: Modifiers are currently not considered by this static analysis.
Pos: 651:8:

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: 665:8:

Solhint Linter

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.

KCSToken.sol

```
Compiler version >=0.6.0 <0.8.0 does not satisfy the ^0.5.8 semver
requirement
Pos: 1:2
Error message for require is too long
Pos: 9:146
Variable name must be in mixedCase
Pos: 5:224
Error message for require is too long
Pos: 9:227
Error message for require is too long
Pos: 9:298
Error message for require is too long
Pos: 9:481
Error message for require is too long
Pos: 9:482
Error message for require is too long
Pos: 9:522
Error message for require is too long
Pos: 9:545
Error message for require is too long
Pos: 9:546
Code contains empty blocks
Pos: 94:577
Provide an error message for require
Pos: 9:624
Provide an error message for require
Pos: 9:633
Provide an error message for require
Pos: 9:664
```

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

Email: audit@EtherAuthority.io