Ether Authority

www.EtherAuthority.io
audit@etherauthority.io

SMART CONTRACT

Security Audit Report

Project:	Injective Token
Website:	injective.com
Platform:	Ethereum
Language:	Solidity
Date:	May 1st, 2024

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

As part of EtherAuthority's community smart contracts audit initiatives, the smart contracts of Injective Token from injective.com were 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 May 1st, 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

- This Solidity smart contract defines an ERC20 token named "Injective Token" with the symbol "INJ". Let's break down the main components of the contract:
    - **ERC20 Contract:** This contract implements the ERC20 interface. It includes functions to transfer tokens (transfer()), approve token spending (approve()), transfer tokens on behalf of others (transferFrom()), and manage allowances. It also includes functions to increase and decrease allowances (increaseAllowance() and decreaseAllowance()). Additionally, it includes functions to mint (_mint()) and burn (_burn()) tokens.
    - **InjectiveToken Contract:** This contract inherits from the ERC20 contract and initializes the "Injective Token" with the symbol "INJ". It mints 100 million tokens to the specified custodian address upon deployment.
- Overall, this contract provides a basic implementation of an ERC20 token with standard functionality, along with additional safety features such as overflow/underflow checks and address validation.
- The token is without any other custom functionality and without any ownership control, which makes it truly decentralized.

Audit scope

Name	Code Review and Security Analysis Report for Injective Token Smart Contract
Platform	**Ethereum**
File	InjectiveToken.sol
Smart Contract Code	0xe28b3B32B6c345A34Ff64674606124Dd5Aceca30
Audit Date	May 1st, 2024

Email: audit@EtherAuthority.io

Claimed Smart Contract Features

Claimed Feature Detail	Our Observation
Tokenomics: • Name: Injective Token • Symbol: INJ • Decimals: 18 • Total Supply: 100 Million	**YES, This is valid.**
Ownership control: • There are no owner functions, which makes it 100% decentralized.	**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 not have any ownership control, hence it is **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. A 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 3 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	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	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	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	Passed
	“Short Address” Attack	Passed
	“Double Spend” Attack	Passed

Overall Audit Result: **PASSED**

Email: audit@EtherAuthority.io

Business Risk Analysis

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

Overall Audit Result: **PASSED**

Email: audit@EtherAuthority.io

Code Quality

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

The libraries in Injective 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 Injective 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 contracts. Ethereum's NatSpec commenting style is recommended.

Documentation

We were given an Injective Token smart contract code in the form of an Etherscan 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 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	name	read	Passed	No Issue
3	symbol	read	Passed	No Issue
4	decimals	read	Passed	No Issue
5	totalSupply	read	Passed	No Issue
6	balanceOf	read	Passed	No Issue
7	transfer	write	Passed	No Issue
8	allowance	read	Passed	No Issue
9	approve	write	Passed	No Issue
10	transferFrom	write	Passed	No Issue
11	increaseAllowance	write	Passed	No Issue
12	decreaseAllowance	write	Passed	No Issue
13	_transfer	internal	Passed	No Issue
14	_mint	internal	Passed	No Issue
15	_burn	internal	Passed	No Issue
16	_approve	internal	Passed	No Issue
17	_setupDecimals	internal	Passed	No Issue
18	_beforeTokenTransfer	internal	Passed	No Issue
19	_msgSender	internal	Passed	No Issue
20	_msgData	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.

Email: audit@EtherAuthority.io

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 0.6.12;
```

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

Resolution: Please use versions greater than 0.8.7.

(2) Error message for require is too long:

```
   require(success, "Address: unable to send value, recipient may
have reverted");
```

Ethereum has a gas limit for each block. This limit includes the gas used by all transactions and contract executions within that block. When a required statement fails, it

results in an exception, and the error message, along with the gas used up to that point, is included in the transaction's revert message.

Resolution: We suggest writing short and clear messages in the required statement.

(3) Low Level Calls:

```
 (bool success, ) = recipient.call{ value: amount }("");
   (bool success, bytes memory returndata) = target.call{ value:
weiValue }(data);
```

This contract uses low-level calls, which may be unsafe.

Resolution: Enhance safety by reviewing low-level calls, considering high-level alternatives, and consulting security experts. Prioritize code security and integrity.

Centralization Risk

The Injective Token smart contract does not have any ownership control, **hence it is 100% decentralized.**

Therefore, there is **no** centralization risk.

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 3 informational issues in the smart contracts. And those issues 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, 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).

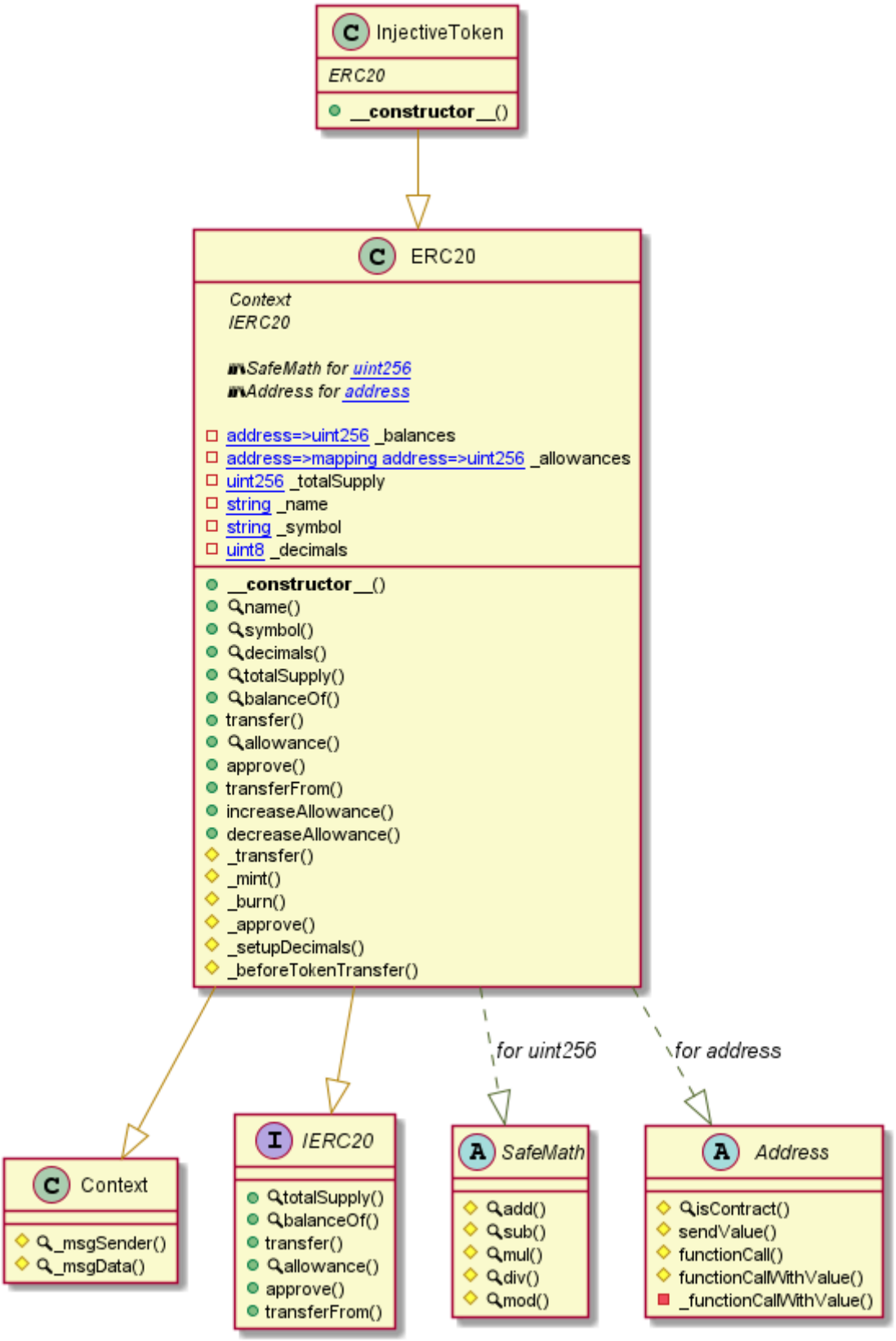
Due to the fact that the total number of test cases are unlimited, the audit makes no statements or warranties on security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bugfree status or any other statements of the contract. While we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

```
ERC20.constructor(string,string).name (InjectiveToken.sol#451) shadows:
        - ERC20.name() (InjectiveToken.sol#460-462) (function)
ERC20.constructor(string,string).symbol (InjectiveToken.sol#451) shadows:
        - ERC20.symbol() (InjectiveToken.sol#468-470) (function)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#local-variable-shadowing

Address.isContract(address) (InjectiveToken.sol#286-295) uses assembly
        - INLINE ASM (InjectiveToken.sol#293)
Address._functionCallWithValue(address,bytes,uint256,string) (InjectiveToken.sol#379-400) uses assembly
        - INLINE ASM (InjectiveToken.sol#392-395)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#assembly-usage

Address._functionCallWithValue(address,bytes,uint256,string) (InjectiveToken.sol#379-400) is never used and should be removed
Address.functionCall(address,bytes) (InjectiveToken.sol#339-341) is never used and should be removed
Address.functionCall(address,bytes,string) (InjectiveToken.sol#349-351) is never used and should be removed
Address.functionCallWithValue(address,bytes,uint256) (InjectiveToken.sol#364-366) is never used and should be removed
Address.functionCallWithValue(address,bytes,uint256,string) (InjectiveToken.sol#374-377) is never used and should be removed
Address.isContract(address) (InjectiveToken.sol#286-295) is never used and should be removed
Address.sendValue(address,uint256) (InjectiveToken.sol#313-319) is never used and should be removed
Context._msgData() (InjectiveToken.sol#26-29) is never used and should be removed
ERC20._burn(address,uint256) (InjectiveToken.sol#644-652) is never used and should be removed
ERC20._setupDecimals(uint8) (InjectiveToken.sol#682-684) is never used and should be removed
SafeMath.div(uint256,uint256) (InjectiveToken.sol#206-208) is never used and should be removed
SafeMath.div(uint256,uint256,string) (InjectiveToken.sol#222-228) is never used and should be removed
SafeMath.mod(uint256,uint256) (InjectiveToken.sol#242-244) is never used and should be removed
SafeMath.mod(uint256,uint256,string) (InjectiveToken.sol#258-261) is never used and should be removed
SafeMath.mul(uint256,uint256) (InjectiveToken.sol#180-192) is never used and should be removed
SafeMath.sub(uint256,uint256) (InjectiveToken.sol#149-151) is never used and should be removed
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dead-code

Low level call in Address.sendValue(address,uint256) (InjectiveToken.sol#313-319):
        - (success) = recipient.call{value: amount}() (InjectiveToken.sol#317)
Low level call in Address._functionCallWithValue(address,bytes,uint256,string) (InjectiveToken.sol#379-400):
        - (success,returndata) = target.call{value: weiValue}(data) (InjectiveToken.sol#383)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#low-level-calls

Redundant expression "this (InjectiveToken.sol#27)" inContext (InjectiveToken.sol#21-30)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#redundant-statements

InjectiveToken.constructor(address) (InjectiveToken.sol#705-707) uses literals with too many digits:
        - _mint(custodian,100000000000000000000000000) (InjectiveToken.sol#706)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#too-many-digits
InjectiveToken.sol analyzed (6 contracts with 84 detectors), 24 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.

InjectiveToken.sol

Check-effects-interaction:

Potential violation of Checks-Effects-Interaction pattern in Address._functionCallWithValue(address,bytes,uint256,string): Could potentially lead to re-entrancy vulnerability.
more
Pos: 379:4:

Inline assembly:

The Contract uses inline assembly, this is only advised in rare cases. Additionally static analysis modules do not parse inline Assembly, this can lead to wrong analysis results.
more
Pos: 293:8:

Low level calls:

Use of "call": should be avoided whenever possible. It can lead to unexpected behavior if return value is not handled properly. Please use Direct Calls via specifying the called contract's interface.
more
Pos: 383:50:

Gas costs:

Gas requirement of function InjectiveToken.decreaseAllowance 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: 584:4:

Constant/View/Pure functions:

ERC20._beforeTokenTransfer(address,address,uint256) : Potentially should be constant/view/pure but is not.
more
Pos: 700:4:

Similar variable names:

ERC20._mint(address,uint256) : Variables have very similar names "account" and "amount".
Pos: 630:34:

Similar variable names:

ERC20._burn(address,uint256) : Variables have very similar names "account" and "amount".
Pos: 649:52:

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: 318:8:

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: 224:20:

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.

InjectiveToken.sol

```
Compiler version 0.6.12 does not satisfy the ^0.5.8 semver
requirement
Pos: 1:6
Error message for require is too long
Pos: 9:188
Error message for require is too long
Pos: 9:317
Error message for require is too long
Pos: 9:374
Error message for require is too long
Pos: 9:603
Error message for require is too long
Pos: 9:604
Error message for require is too long
Pos: 9:644
Error message for require is too long
Pos: 9:667
Error message for require is too long
Pos: 9:668
Code contains empty blocks
Pos: 22:700
```

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