Ether Authority

www.EtherAuthority.io
audit@etherauthority.io

SMART CONTRACT

Security Audit Report

Customer:	Aeka Defi
Website:	https://aeka.cc
Platform:	Binance Smart Chain
Language:	Solidity
Date:	September 27th, 2021

Table of contents

Introduction

EtherAuthority was contracted by the Aeka Defi team to perform the Security audit of the Aeka 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 September 27th, 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

AEKA Token is a BEP20 standard token smart contract running on Binance Smart Chain. It will be acting as a backbone of the Aeka Defi ecosystem.

Audit scope

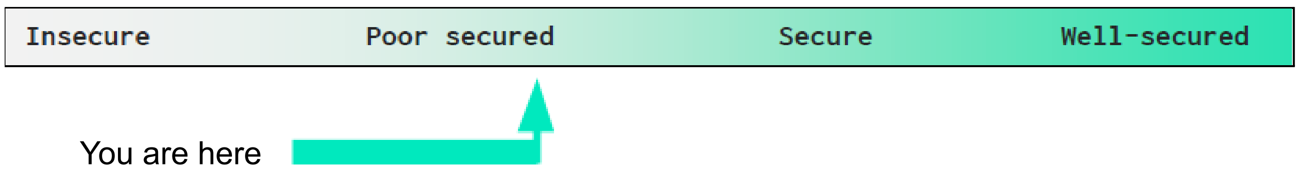
Name	Code Review and Security Analysis Report for Aeka Token Smart Contract
Platform	**BSC / Solidity**
File	Token.sol
File MD5 Hash	7F68A7FECF3A3826C2D351E58BD080DB
Online Code	https://bscscan.com/address/0xbd4a96b5a14415BBB6DE9010A07CbB2b2D555e23#code
Audit Date	September 27th, 2021

Claimed Smart Contract Features

Claimed Feature Detail	Our Observation
Tokenomics: ● Name: Aeka ● Symbol: AEKA ● Decimals: 18 ● TotalSupply: 10,000,000 ● No more token minting possible	**YES, This is valid.**

Audit Summary

According to the standard audit assessment, Customer`s solidity smart contracts are **"Poorly Secured"**. This token contract does not contain owner control, which makes 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 1 critical, 1 high, 0 medium and 1 low and some very low level issues.

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	Moderated
	Human/contract checks bypass	Passed
	Random number generation/use vulnerability	Passed
	Fallback function misuse	Passed
	Race condition	Passed
	Logical vulnerability	Not 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	Passed
	“Short Address” Attack	Passed
	“Double Spend” Attack	Passed

Overall Audit Result: NOT PASSED

Code Quality

This audit scope has 1 smart contract file. Smart contracts do not contain Libraries, Smart contracts, inherits and Interfaces. This is a compact and well written smart contract.

The Aeka 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 **not well** commented on smart contracts.

Documentation

We were given a Aeka Token smart contract 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 **not well** commented. So it is not 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 **not** 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	Missing event	Refer audit finding section
2	balanceOf	write	This should be view function	Refer audit finding section
3	transfer	write	Passed	No Issue
4	transferFrom	write	Unlimited token spending	Refer audit finding section
5	approve	write	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.

Audit Findings

Critical Severity

(1) Unlimited token spending

```
function transferFrom(address from, address to, uint value) public returns(bool) {
    require(balanceOf(from) >= value, 'balance too low');
    require(allowance[from][msg.sender] >= value, 'allowance too low');
    balances[to] += value;
    balances[from] -= value;
    emit Transfer(from, to, value);
    return true;
}
```

The transferFrom function does not have a code to deduct the allowance. So, once the owner gives approval, the spender can keep spending his tokens multiple times.

For example, Alice gives approval to Bob to spend 100 tokens. And Bob enjoys withdrawing 100 tokens multiple times!

Resolution: We suggest adding one more line of code as below:

allowance[from][msg.sender] -= value;

High Severity

(2) BEP20 standard violation

```
function balanceOf(address owner) public returns(uint) {
    return balances[owner];
}
```

The balanceOf function is missing the “view” keyword. This will make this to be interpreted as a “write function” instead of a “read function”. As a standard BEP20 requirement, this function must be a read or view function.

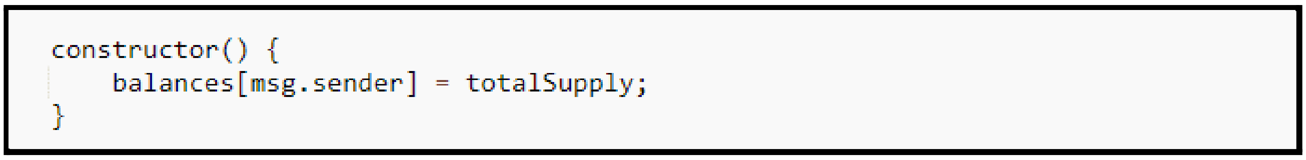
Resolution: Please add “view” keyword

Medium

No Medium severity vulnerabilities were found.

Low

(1) Missing event:



```
constructor() {
    balances[msg.sender] = totalSupply;
}
```

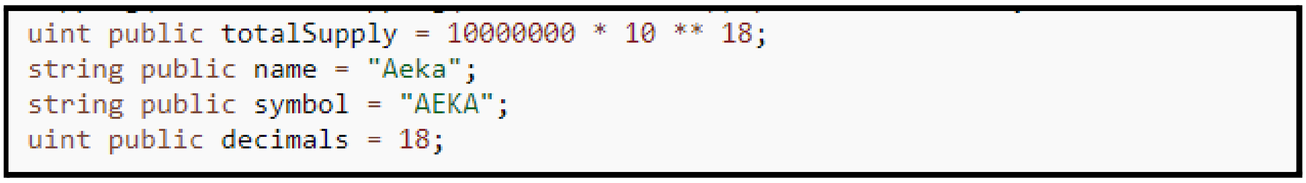
Tokens are transferred from address(0) to msg.sender. Thus, it is important to add a Transfer event to properly track the token movement by the clients.

Resolution: Please add a Transfer event in the constructor as below:

emit Transfer(address(0), msg.sender, totalSupply);

Very Low / Informational / Best practices:

(1) Variables can be constant



```
uint public totalSupply = 10000000 * 10 ** 18;
string public name = "Aeka";
string public symbol = "AEKA";
uint public decimals = 18;
```

The variables name, decimals, symbol, totalSupply values will be unchanged. So, It is recommended to make them constant. It will save some gas..

Resolution: Please add “constant” keyword

(2) Missing SPDX license identifier:

Solidity’s new specification requires a valid SPDX licence identifier to be included in every smart contract file.

Resolution: Please add a comment for appropriate SPDX licence identifier

(3) Consider adding the latest solidity version:

v0.8.2+commit.661d1103

The current version is an upgraded one, but consider using 0.8.7 which is the latest at the time of this audit.

(4) Visibility external instead of public:

It is recommended to use the function visibility as external instead of public, if it is not called from inside the contract. This will save some gas.

https://ethereum.stackexchange.com/questions/19380/external-vs-public-best-practices

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 and once those issues are fixed/acknowledged, **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 **"Poorly 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).

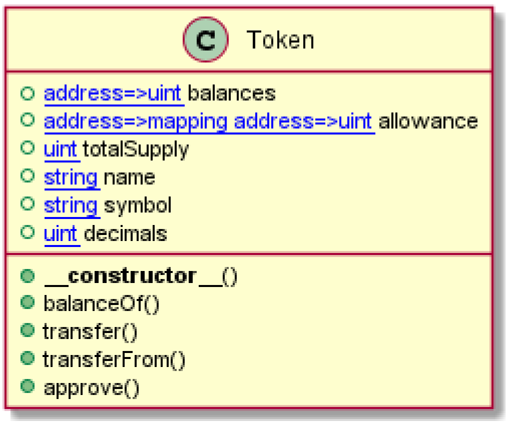
Due to the fact that the total number of test cases are unlimited, the audit makes no statements or warranties on security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bugfree status or any other statements of the contract. While we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

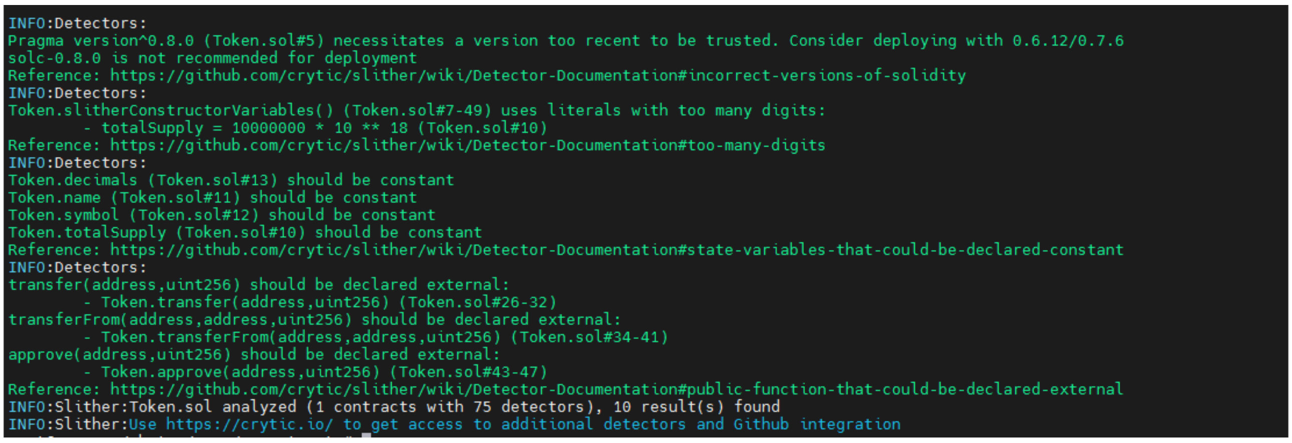
Appendix

Code Flow Diagram - Token



Slither Results Log

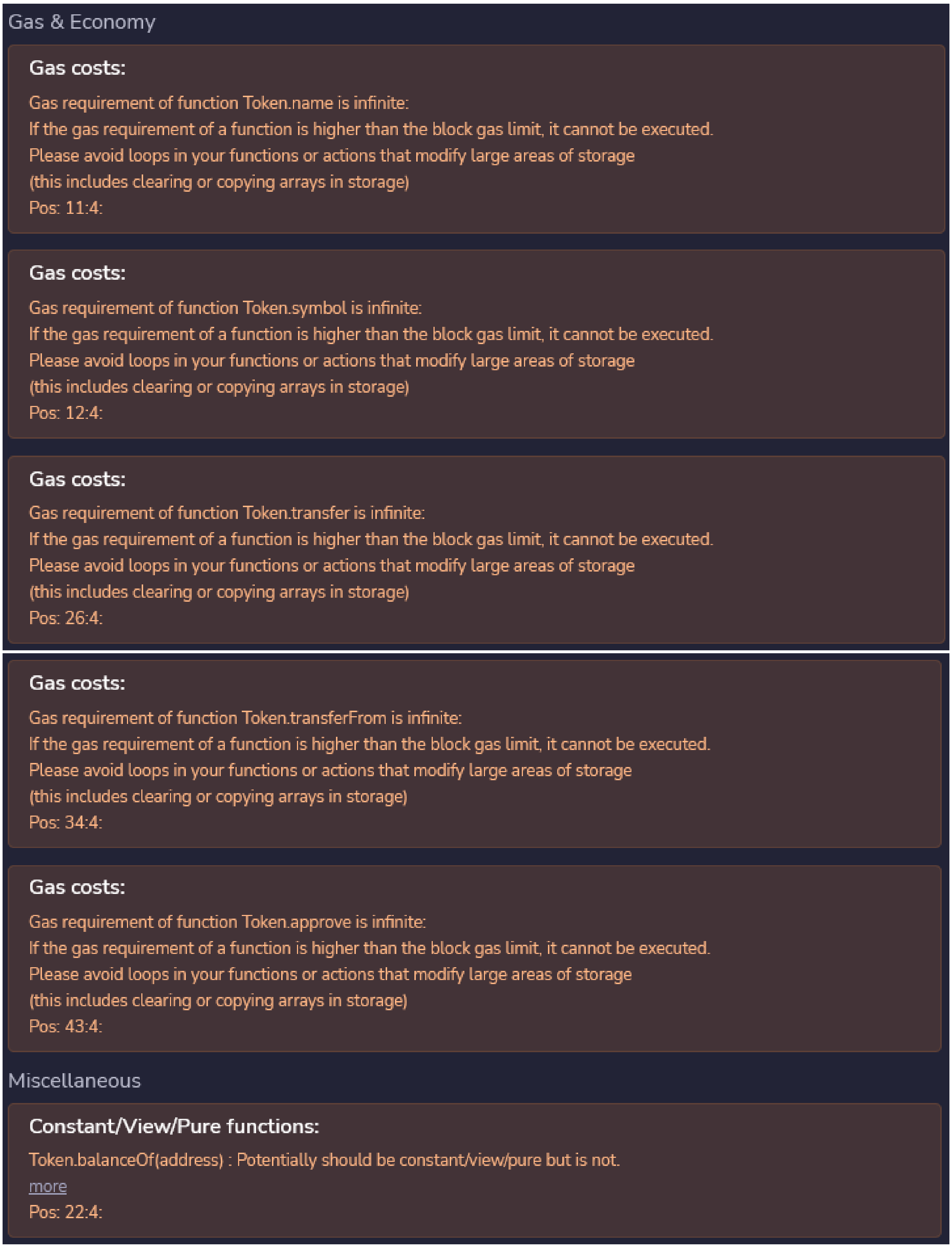
Slither log >> Token.sol



```
INFO:Detectors:
Pragma version^0.8.0 (Token.sol#5) necessitates a version too recent to be trusted. Consider deploying with 0.6.12/0.7.6
solc-0.8.0 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
INFO:Detectors:
Token.slitherConstructorVariables() (Token.sol#7-49) uses literals with too many digits:
        - totalSupply = 10000000 * 10 ** 18 (Token.sol#10)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#too-many-digits
INFO:Detectors:
Token.decimals (Token.sol#13) should be constant
Token.name (Token.sol#11) should be constant
Token.symbol (Token.sol#12) should be constant
Token.totalSupply (Token.sol#10) should be constant
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#state-variables-that-could-be-declared-constant
INFO:Detectors:
transfer(address,uint256) should be declared external:
        - Token.transfer(address,uint256) (Token.sol#26-32)
transferFrom(address,address,uint256) should be declared external:
        - Token.transferFrom(address,address,uint256) (Token.sol#34-41)
approve(address,uint256) should be declared external:
        - Token.approve(address,uint256) (Token.sol#43-47)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#public-function-that-could-be-declared-external
INFO:Slither:Token.sol analyzed (1 contracts with 75 detectors), 10 result(s) found
INFO:Slither:Use https://crytic.io/ to get access to additional detectors and Github integration
```

Solidity Static Analysis

Token.sol



Gas & Economy

Gas costs:

Gas requirement of function Token.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: 11:4:

Gas costs:

Gas requirement of function Token.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: 12:4:

Gas costs:

Gas requirement of function Token.transfer 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: 26:4:

Gas costs:

Gas requirement of function Token.transferFrom 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: 34:4:

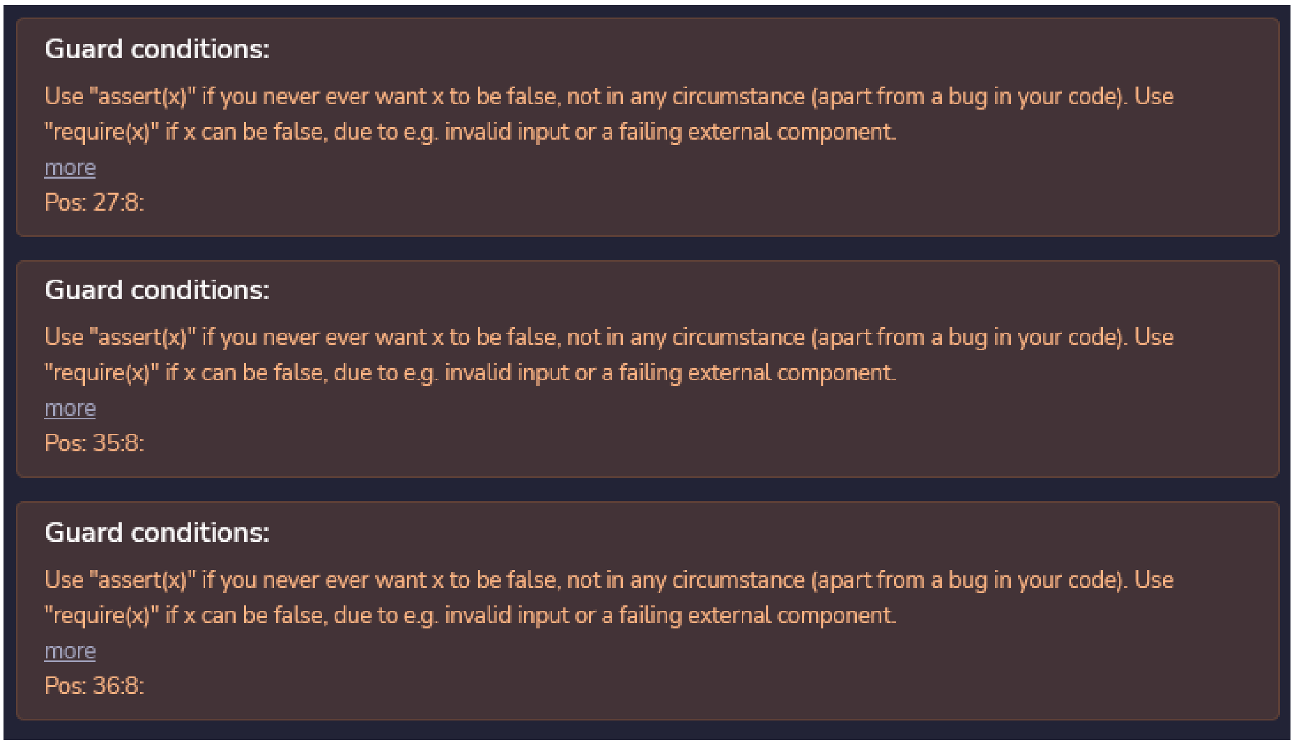
Gas costs:

Gas requirement of function Token.approve 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: 43:4:

Miscellaneous

Constant/View/Pure functions:

Token.balanceOf(address) : Potentially should be constant/view/pure but is not.
more
Pos: 22:4:


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: 27:8:

Guard conditions:

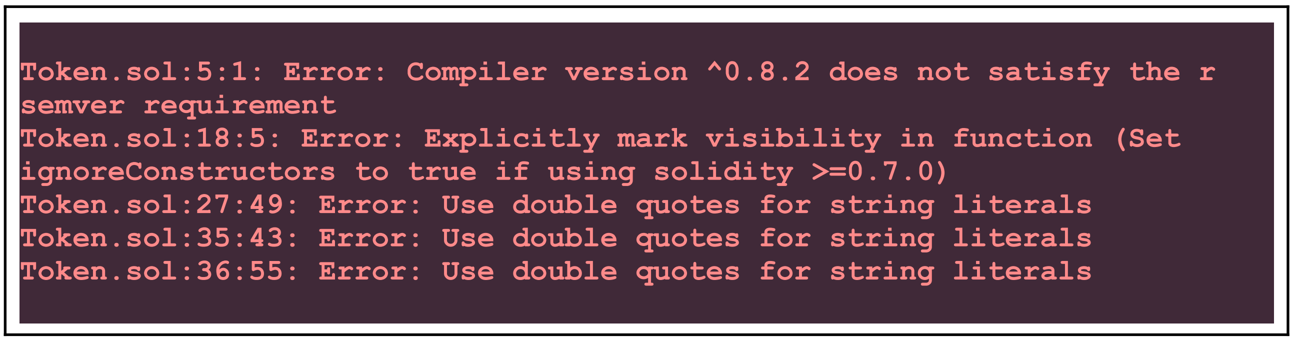
Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.
more
Pos: 35:8:

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: 36:8:

Solhint Linter

Token.sol



```
Token.sol:5:1: Error: Compiler version ^0.8.2 does not satisfy the r
semver requirement
Token.sol:18:5: Error: Explicitly mark visibility in function (Set
ignoreConstructors to true if using solidity >=0.7.0)
Token.sol:27:49: Error: Use double quotes for string literals
Token.sol:35:43: Error: Use double quotes for string literals
Token.sol:36:55: Error: Use double quotes for string literals
```

Software analysis result:

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