EtherAuthority

www.EtherAuthority.io
audit@etherauthority.io

SMART CONTRACT

Security Audit Report

Customer:	KWD Coin
Website:	https://kwdcoin.io
Platform:	Binance Smart Chain
Language:	Solidity
Date:	August 5th, 2021

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 PUBLIC AFTER ISSUES ARE RESOLVED.

Introduction

EtherAuthority was contracted by the KWD Coin team to perform the Security audit of the KWD Coin 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 August 5th, 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

KWD COIN Token is a Decentralized Token (BEP-20 Token) in Binance smart chain.This token contract does not have any owner influenced functions, which means once the tokens are sent initially to owner, then there is no other owner functions to influence the smart contract code.

Audit scope

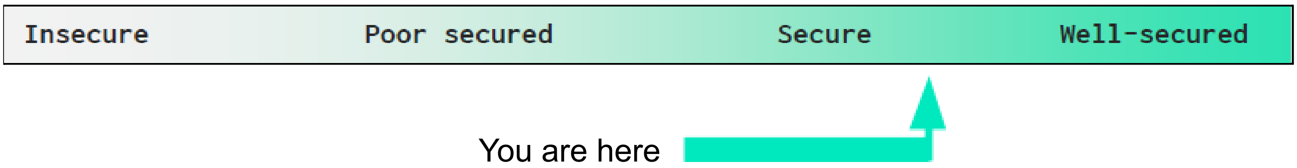
Name	**Code Review and Security Analysis Report for KWD Coin Smart Contract**
Platform	**BSC / Solidity**
File	Token.sol
Smart Contract Online Code	https://bscscan.com/address/0x65453110319e9ce742dd146beFfaC2A5eE655e7E#code
File MD5 Hash	9C790C7218ED48E7D07B670EE5682FFF
Audit Date	August 5th, 2021

Claimed Smart Contract Features

Claimed Feature Detail	Our Observation
Name: KWD	YES, This is valid.
Symbol: KWD	YES, This is valid.
Decimal: 18	YES, This is valid.
Total Supply: 1 Trillion, which was given to the owner at contract deployment.	YES, This is valid.
Redistributed : 5% to LP Liquidity : Locked for 5 years Marketing : 2% Auto Claim : 5% Every 30 minutes (Straight to wallet)	This smart contract does not have any code for this. And this may be handled manually.

Audit Summary

According to the standard audit assessment, Customer`s solidity smart contract is **Secured**. This smart contract issues all the total supply of tokens to the owner and owner must handle those tokens as per the business plan.



We found 0 critical, 0 high, 1 medium and 0 low and some very low level issues.

Investor advice: This smart contract is a standard BEP20 token contract and all the tokens are given to the owner while contract deployment. So, the owner can decide what to do with those tokens. And this technical audit of the code does not guarantee the ethical nature of the project and thus, all investors should do their due diligence before investing into this 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	Passed
	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	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 not set	Passed
	“Short Address” Attack	Passed
	“Double Spend” Attack	Passed

Overall Audit Result: PASSED

Code Quality

This audit scope has 1 smart contract. This is a compact and well written contract. This contract does not contain any library and libraries are not mandatory.

The KWD Coin 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 KWD Coin smart contract code in the form of an BscScan web link. The hash of that code is mentioned above in the table.

As mentioned above, some code parts are **not well** commented. So it is difficult to 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 https://kwdcoin.io/ which provided rich information about the project architecture and tokenomics.

Use of Dependencies

As per our observation, the libraries are not used in this smart contract code.

Apart from libraries, its functions are not used in external smart contract calls.

AS-IS overview

(1) Events

(a) event Transfer(address indexed from, address indexed to, uint value);

(b) event Approval(address indexed owner, address indexed spender, uint value);

(2) Functions

Sl.	Functions	Type	Observation	Conclusion
1	constructor	write	Passed	No Issue
2	balanceOf	read	Missing View keyword	Alternatively, a balance method can be used.
3	transfer	write	Passed	No Issue
4	transferFrom	write	Passed	No Issue
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 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

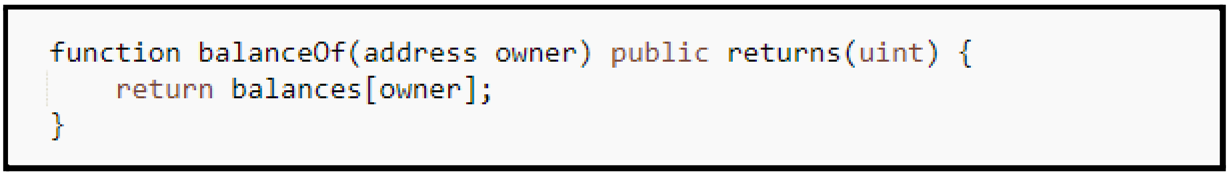
No critical severity vulnerabilities were found.

High

No high severity vulnerabilities were found.

Medium

(1) ERC20 / BEP20 standard violation



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

balanceOf function is not specified as a "view" and thus this will not be considered as a read function, but instead it will be considered as a write function.

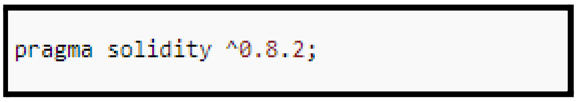
On another hand, this smart contract contains a "balances" method which will be used as an alternative. But any other dapps depending on the balanceOf read method might break.

Low

No Low severity vulnerabilities were found.

Very Low / Discussion / Best practices:

(1) Use latest solidity version:

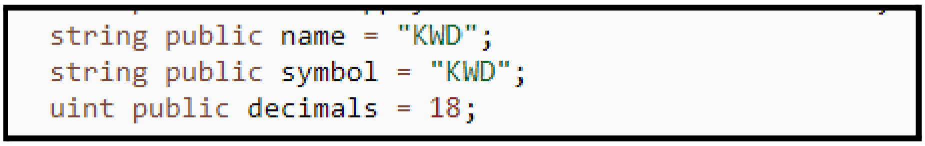


```
pragma solidity ^0.8.2;
```

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

Resolution: Please use 0.8.6 which is the latest version at the time of this audit

(2) Use constant keyword



```
  string public name = "KWD";
  string public symbol = "KWD";
  uint public decimals = 18;
```

If the variables are not being changed, then it is recommended to make them “constant”. Although, current code does not raise any security vulnerability, making constant saves some gas.

(3) approve of ERC20/BEP20 standard

To prevent attack vectors regarding approve() like the one described here: https://docs.google.com/document/d/1YLPtQxZu1UAvO9cZ1O2RPXBbT0mooh4DYKjA_jp-RLM/edit , clients SHOULD make sure to create user interfaces in such a way that they set the allowance first to 0 before setting it to another value for the same spender. THOUGH the contract itself shouldn't enforce it, to allow backwards compatibility with contracts deployed before

(4) Use visibility external over public

If the function is not being called from inside the smart contract, then it is recommended to specify it as an external. It saves 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 those are fixed/acknowledged in the smart contracts. **So it is 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 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 **"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).

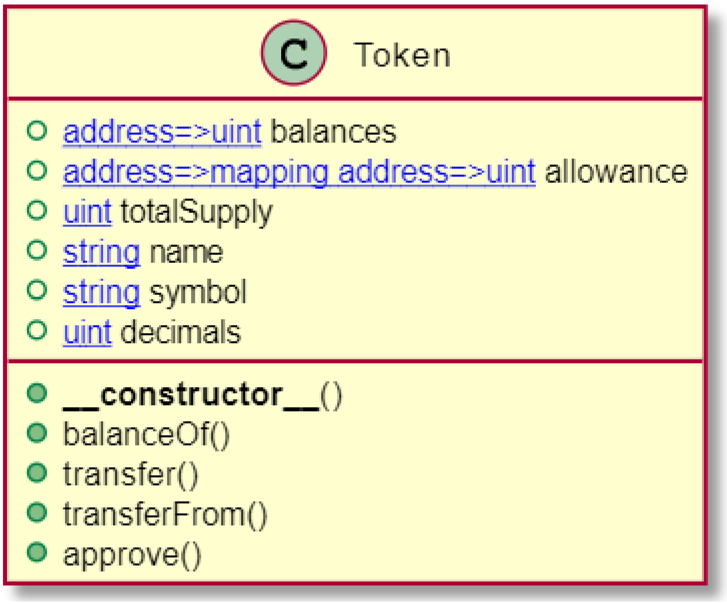
Due to the fact that the total number of test cases are unlimited, the audit makes no statements or warranties on security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bugfree status or any other statements of the contract. While we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

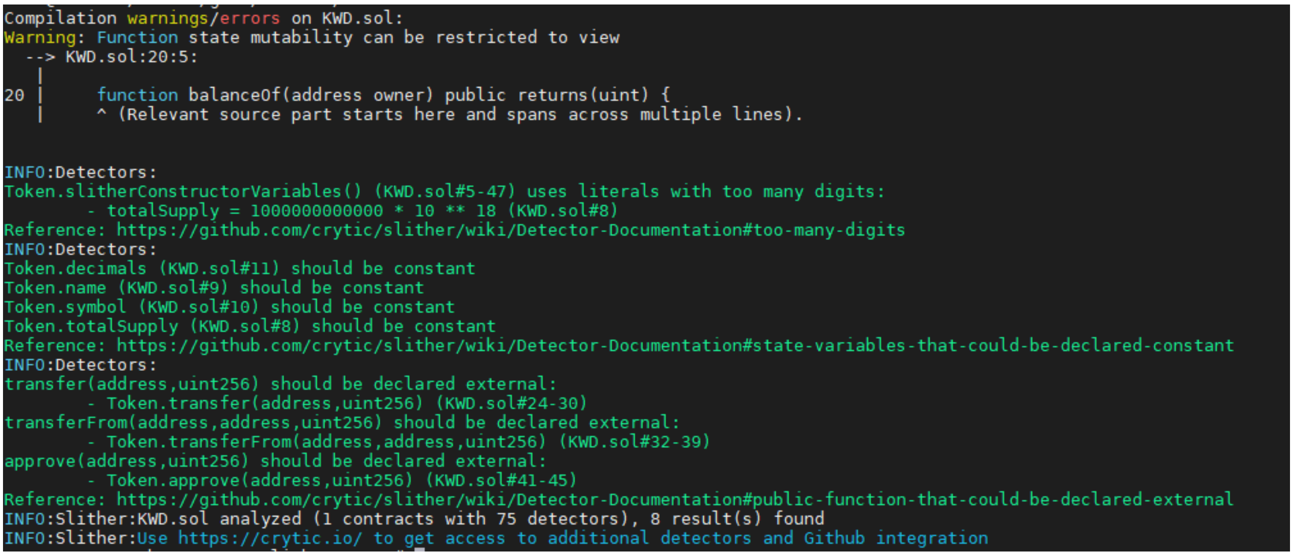
Smart contracts are deployed and executed on the blockchain platform. The platform, its programming language, and other software related to the smart contract can have their own vulnerabilities that can lead to hacks. Thus, the audit can't guarantee explicit security of the audited smart contracts.

Appendix

Code Flow Diagram - KWD Coin



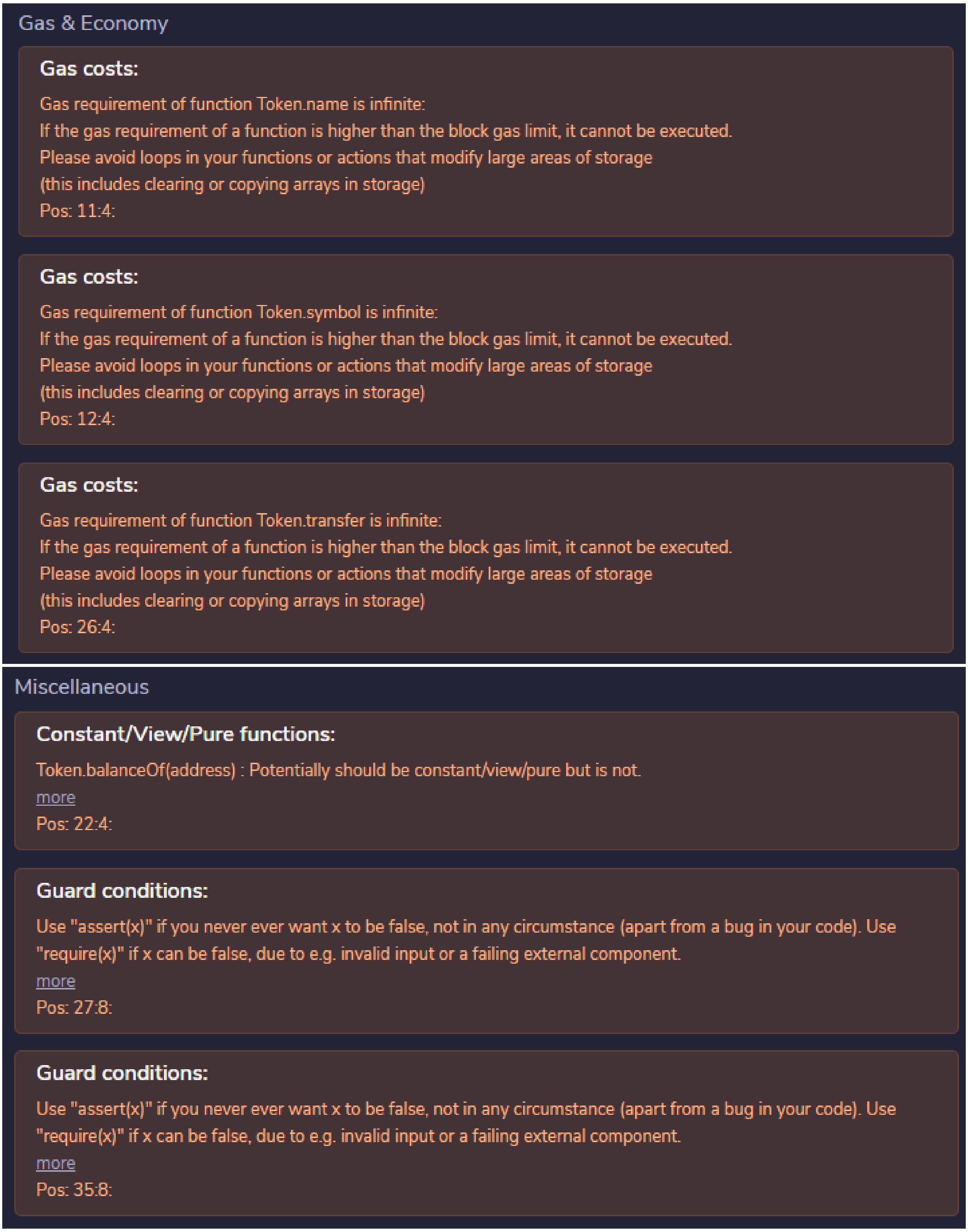
Slither Results Log



```
Compilation warnings/errors on KWD.sol:
Warning: Function state mutability can be restricted to view
  --> KWD.sol:20:5:
   |
20 |     function balanceOf(address owner) public returns(uint) {
   |     ^ (Relevant source part starts here and spans across multiple lines).


INFO:Detectors:
Token.slitherConstructorVariables() (KWD.sol#5-47) uses literals with too many digits:
        - totalSupply = 1000000000000 * 10 ** 18 (KWD.sol#8)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#too-many-digits
INFO:Detectors:
Token.decimals (KWD.sol#11) should be constant
Token.name (KWD.sol#9) should be constant
Token.symbol (KWD.sol#10) should be constant
Token.totalSupply (KWD.sol#8) 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) (KWD.sol#24-30)
transferFrom(address,address,uint256) should be declared external:
        - Token.transferFrom(address,address,uint256) (KWD.sol#32-39)
approve(address,uint256) should be declared external:
        - Token.approve(address,uint256) (KWD.sol#41-45)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#public-function-that-could-be-declared-external
INFO:Slither:KWD.sol analyzed (1 contracts with 75 detectors), 8 result(s) found
INFO:Slither:Use https://crytic.io/ to get access to additional detectors and Github integration
```

Solidity static analyser



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:

Miscellaneous

Constant/View/Pure functions:

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

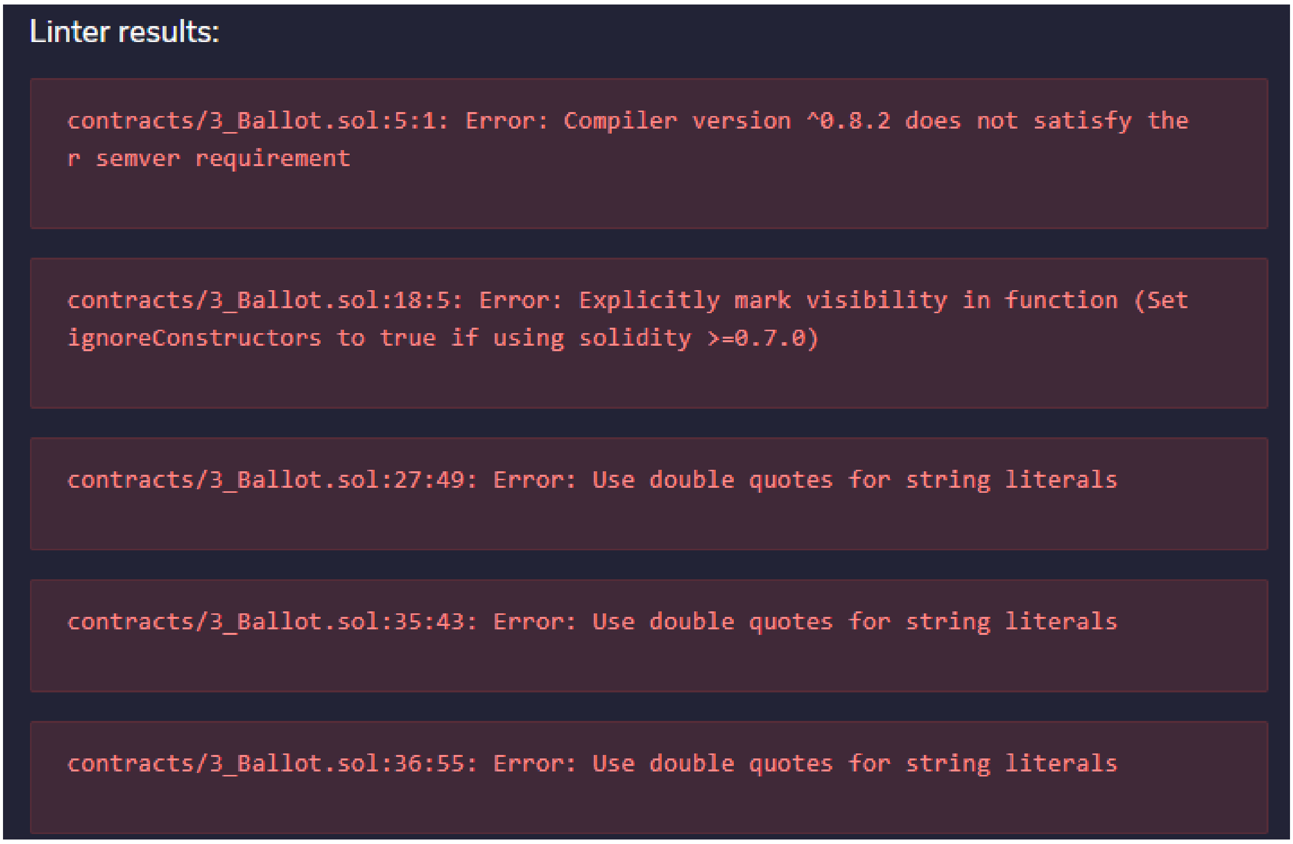
Guard conditions:

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:

Solhint Linter



Linter results:

```
contracts/3_Ballot.sol:5:1: Error: Compiler version ^0.8.2 does not satisfy the
r semver requirement
```

```
contracts/3_Ballot.sol:18:5: Error: Explicitly mark visibility in function (Set
ignoreConstructors to true if using solidity >=0.7.0)
```

```
contracts/3_Ballot.sol:27:49: Error: Use double quotes for string literals
```

```
contracts/3_Ballot.sol:35:43: Error: Use double quotes for string literals
```

```
contracts/3_Ballot.sol:36:55: Error: Use double quotes for string literals
```

EtherAuthority