EtherAuthority

www.EtherAuthority.io
audit@etherauthority.io

SMART CONTRACT

Security Audit Report

Project:	MetaStakeToken
Website:	metastakeswap.com
Platform:	Binance Smart Chain
Language:	Solidity
Date:	April 11th, 2023

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

EtherAuthority was contracted by the Metastake Token team to perform the Security audit of the Metastake 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 April 11th, 2023.

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

- Metastake Token is a bep20 token which has mint, burn functionalities.
- Metastake Token is used for MetastakeSwap runs on the Binance Smart Chain.

Audit scope

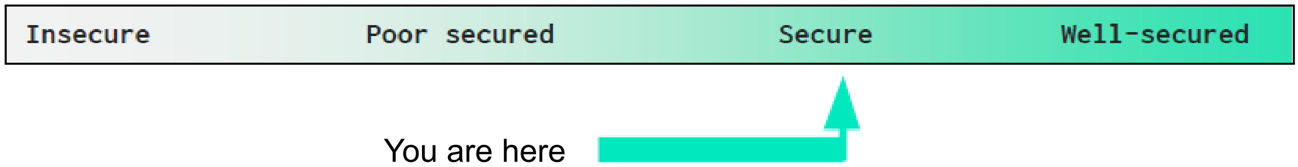
Name	**Code Review and Security Analysis Report for Metastake Token Smart Contract**
Platform	**BSC / Solidity**
File	MetaStakeToken.sol
File MD5 Hash	ADCDAD5FA196273039E0087649F307DA
Online code link	0x5addeb3b6b128f5ba5b70542dedf678d78382970
Audit Date	April 11th, 2023

Claimed Smart Contract Features

Claimed Feature Detail	Our Observation
Tokenomics: ● Name: METASTAKE EXCHANGE ● Symbol: MSE ● Decimals: 18 ● Maximum Supply: 100 Million	YES, This is valid.
Owner Specifications: ● The MasterChef owner can mint tokens.	YES, This is valid.

Audit Summary

According to the standard audit assessment, Customer`s solidity based smart contracts are **“Secured”.** This token contract does contain owner control, which does not make it fully decentralized.



Insecure Poor secured Secure Well-secured

You are here

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 0 critical, 0 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	Moderate
	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	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: PASSED

Code Quality

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

The libraries in the Metastake 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 Metastake Token.

The Metastake 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 in the smart contracts. Ethereum's NatSpec commenting style is used, which is a good thing.

Documentation

We were given a Metastake 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. 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.

Another source of information was its official website:https://metastakeswap.com which provided rich information about the project architecture and tokenomics.

Use of Dependencies

As per our observation, the libraries are 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	getOwner	external	Passed	No Issue
3	name	read	Passed	No Issue
4	decimals	read	Passed	No Issue
5	symbol	read	Passed	No Issue
6	totalSupply	read	Passed	No Issue
7	balanceOf	read	Passed	No Issue
8	transfer	write	Passed	No Issue
9	allowance	read	Passed	No Issue
10	approve	write	Passed	No Issue
11	transferFrom	write	Passed	No Issue
12	increaseAllowance	write	Passed	No Issue
13	decreaseAllowance	write	Passed	No Issue
14	mint	write	access only Owner	No Issue
15	_transfer	internal	Passed	No Issue
16	_mint	internal	Passed	No Issue
17	_burn	internal	Passed	No Issue
18	_approve	internal	Passed	No Issue
19	_burnFrom	internal	Passed	No Issue
20	mintFor	write	access only Owner	No Issue
21	mint	write	access only Owner	No Issue
22	delegates	external	Passed	No Issue
23	delegate	external	Passed	No Issue
24	delegateBySig	external	Function Not able to delegate from the signer account	Refer to audit findings
25	getCurrentVotes	external	Passed	No Issue
26	getPriorVotes	external	Passed	No Issue
27	_delegate	external	Passed	No Issue
28	_moveDelegates	internal	Passed	No Issue
29	_writeCheckpoint	internal	Passed	No Issue
30	safe32	internal	Passed	No Issue
31	getChainId	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.

Audit Findings

Critical Severity

No Critical severity vulnerabilities were found.

High Severity

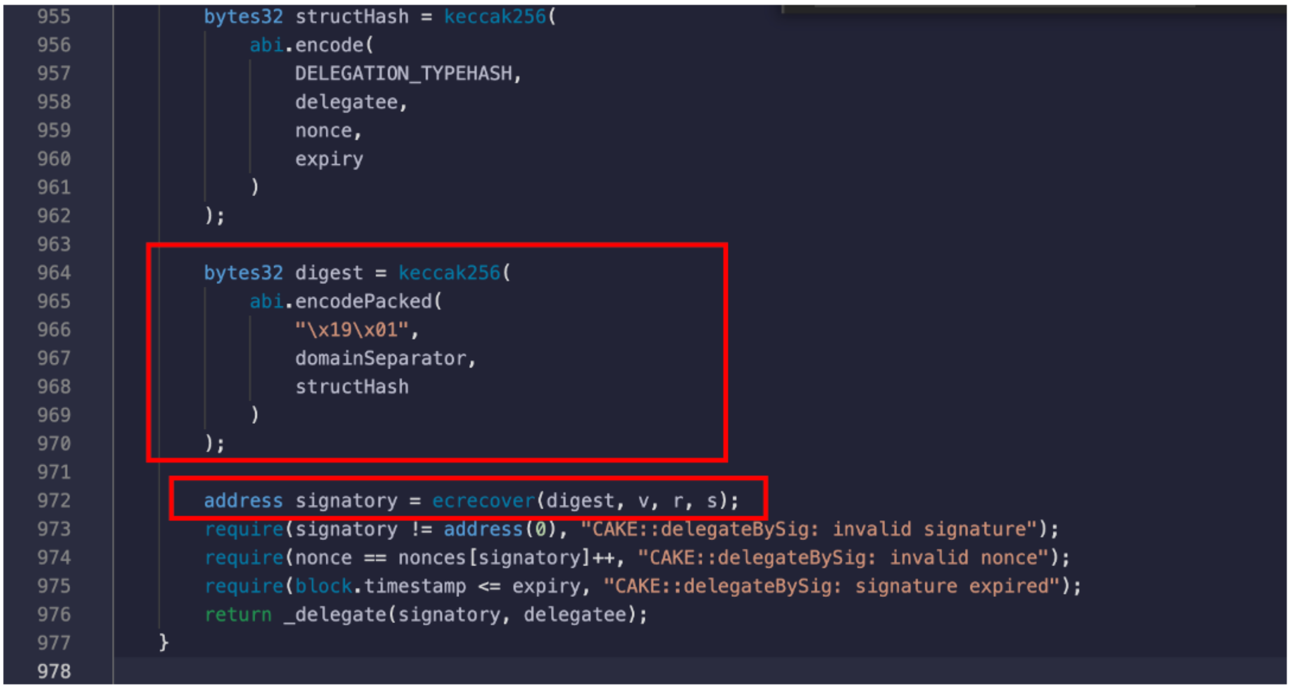
No high severity vulnerabilities were found.

Medium

No medium severity vulnerabilities were found.

Low

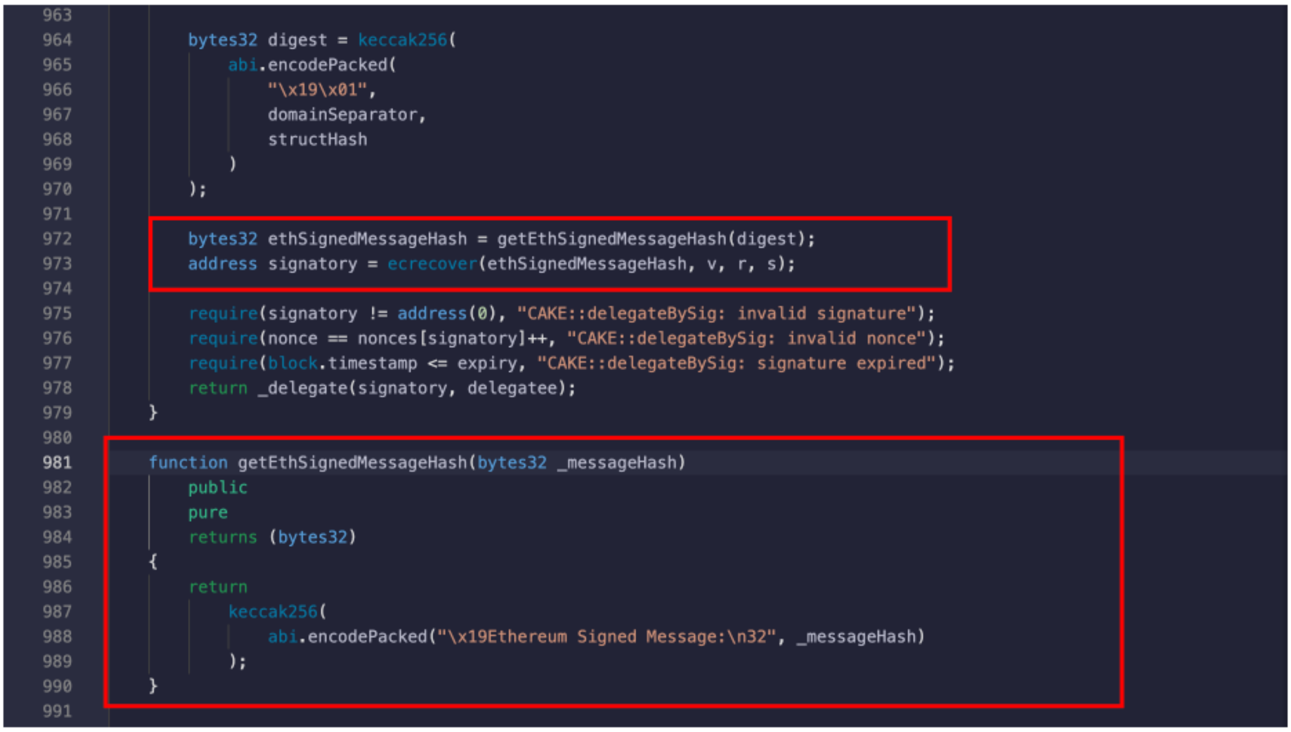
(1) Using the delegateBySig function, not able to delegate from the signer account:



```
        bytes32 structHash = keccak256(
            abi.encode(
                DELEGATION_TYPEHASH,
                delegatee,
                nonce,
                expiry
            )
        );

        bytes32 digest = keccak256(
            abi.encodePacked(
                "\x19\x01",
                domainSeparator,
                structHash
            )
        );

        address signatory = ecrecover(digest, v, r, s);
        require(signatory != address(0), "CAKE::delegateBySig: invalid signature");
        require(nonce == nonces[signatory]++, "CAKE::delegateBySig: invalid nonce");
        require(block.timestamp <= expiry, "CAKE::delegateBySig: signature expired");
        return _delegate(signatory, delegatee);
    }
```

In the delegateBySig function, a hash (digest) of the domain separator is created and passed to ecrecover, which returns an unknown address. So, this function is failing (unable to pass delegation from the signer account).

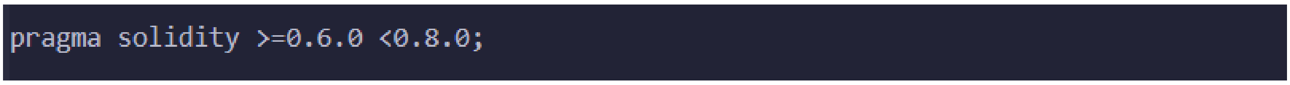
Resolution: We suggest converting to Ethereum Signed Message of length 32 and pass to ecrecover then able to recover the correct signer and pass the delegatee from the signer's account.



```
        bytes32 digest = keccak256(
            abi.encodePacked(
                "\x19\x01",
                domainSeparator,
                structHash
            )
        );

        bytes32 ethSignedMessageHash = getEthSignedMessageHash(digest);
        address signatory = ecrecover(ethSignedMessageHash, v, r, s);

        require(signatory != address(0), "CAKE::delegateBySig: invalid signature");
        require(nonce == nonces[signatory]++, "CAKE::delegateBySig: invalid nonce");
        require(block.timestamp <= expiry, "CAKE::delegateBySig: signature expired");
        return _delegate(signatory, delegatee);
    }

    function getEthSignedMessageHash(bytes32 _messageHash)
        public
        pure
        returns (bytes32)
    {
        return
            keccak256(
                abi.encodePacked("\x19Ethereum Signed Message:\n32", _messageHash)
            );
    }
```

Very Low / Informational / Best practices:

(1) Use latest solidity version:

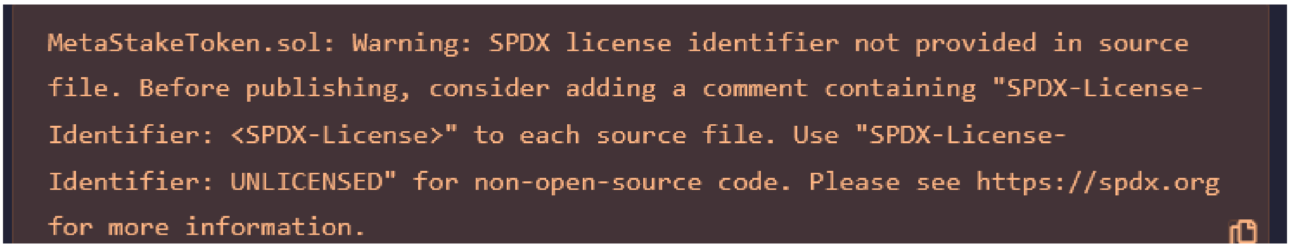


```
pragma solidity >=0.6.0 <0.8.0;
```

Using the latest solidity will prevent any compiler level bugs.

Resolution: We suggest using version > 0.8.0.

(2) Warning: SPDX license identifier:



```
MetaStakeToken.sol: Warning: SPDX license identifier not provided in source file. Before publishing, consider adding a comment containing "SPDX-License-Identifier: <SPDX-License>" to each source file. Use "SPDX-License-Identifier: UNLICENSED" for non-open-source code. Please see https://spdx.org for more information.
```

Warning: SPDX license identifier not provided in source file.

Resolution: We suggest adding SPDX-License-Identifier.

(3) Multiple pragma:

There are multiple pragmas with different compiler versions.

Resolution: We suggest using only one pragma and removing the other.

Centralization

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

MetaStakeToken.sol

- mintFor: The MasterChef owner can create tokens.
- mint: Create the amounts of tokens and assign them to the account, increasing the total supply.

BEP20.sol

- mint: Create the amounts of tokens and assign them to Owner, increasing the total supply.

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 a bscscan.com link and we have used all possible tests based on given objects as files. We have observed 1 low severity issue and 3 informational severity issues in the token smart contract. But those issues are not critical. So, **it's good to go for the mainnet deployment**.

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.

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 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).

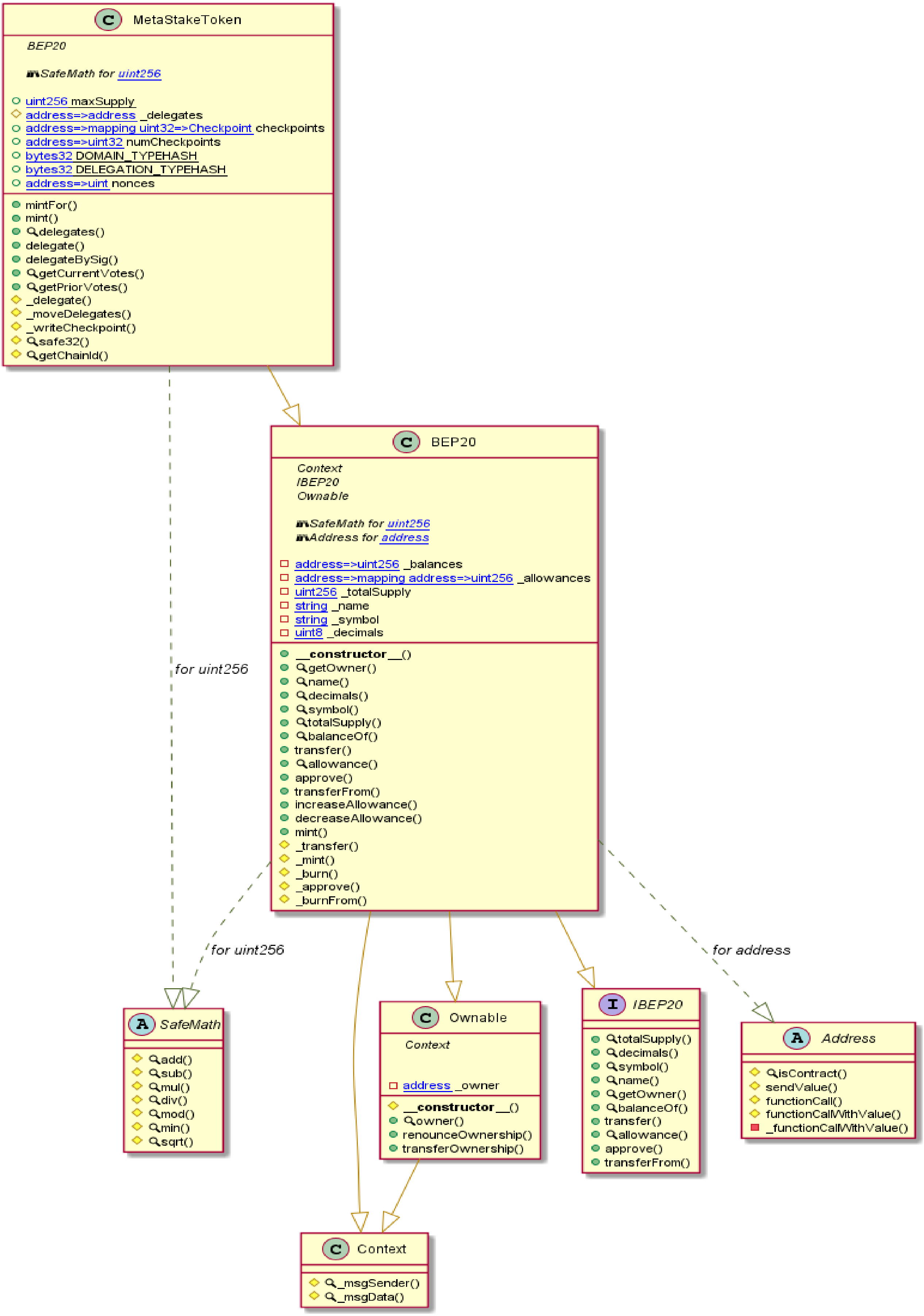
Due to the fact that the total number of test cases are unlimited, the audit makes no statements or warranties on security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bugfree status or any other statements of the contract. While we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

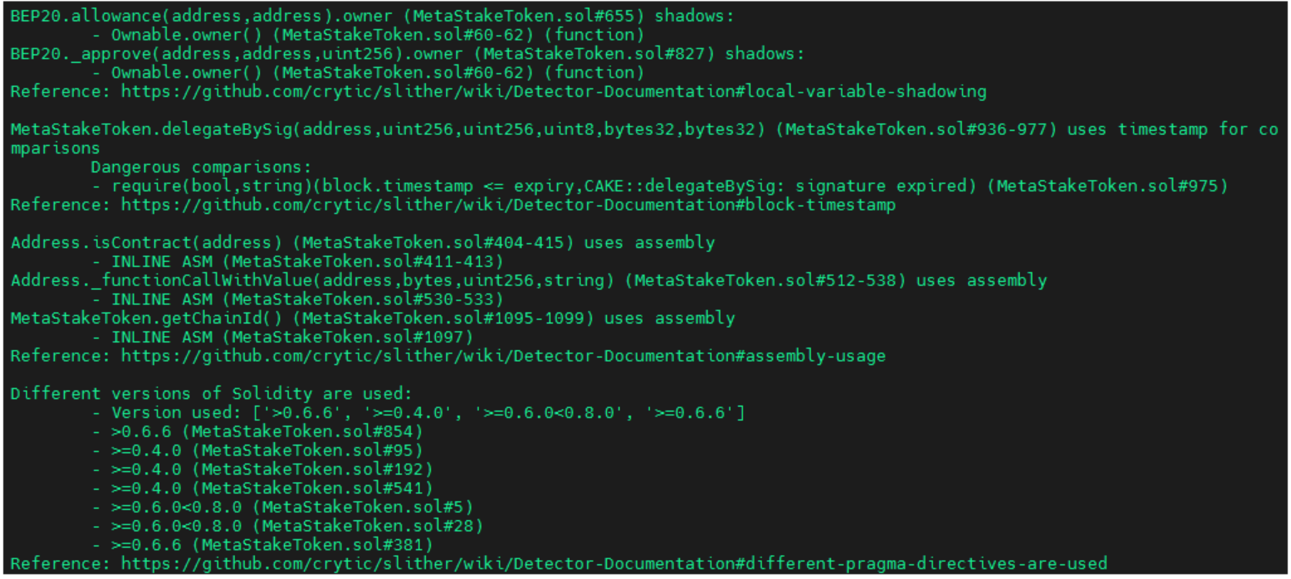
Appendix

Code Flow Diagram - Metastake Token

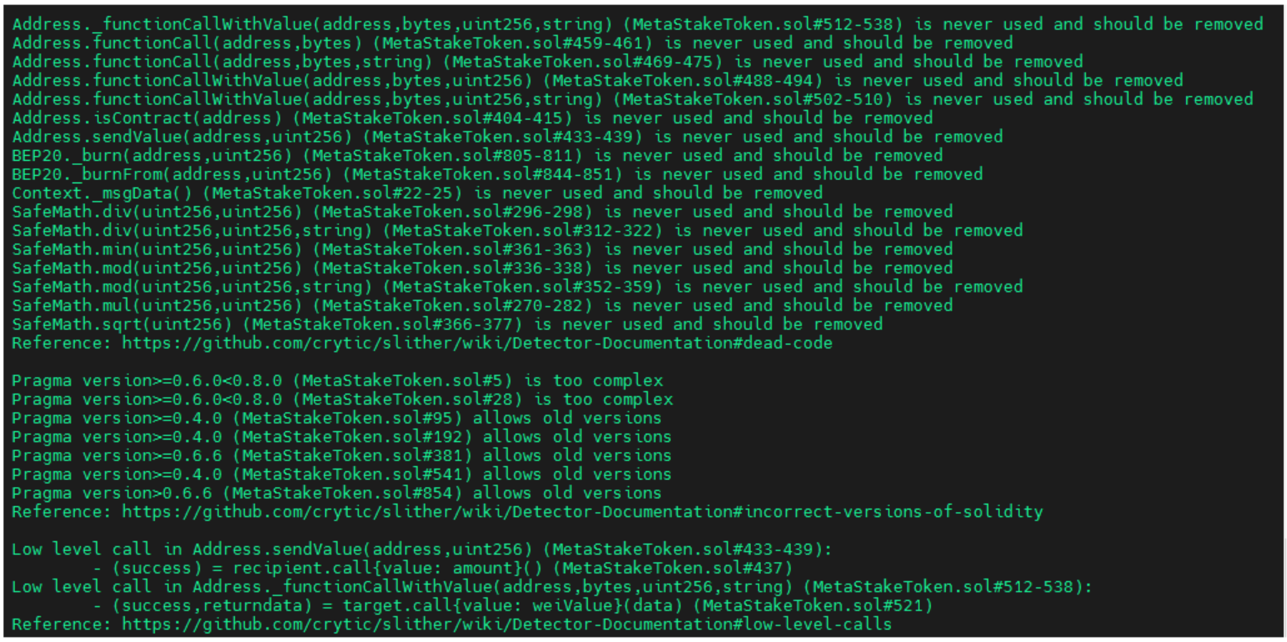


Slither Results Log

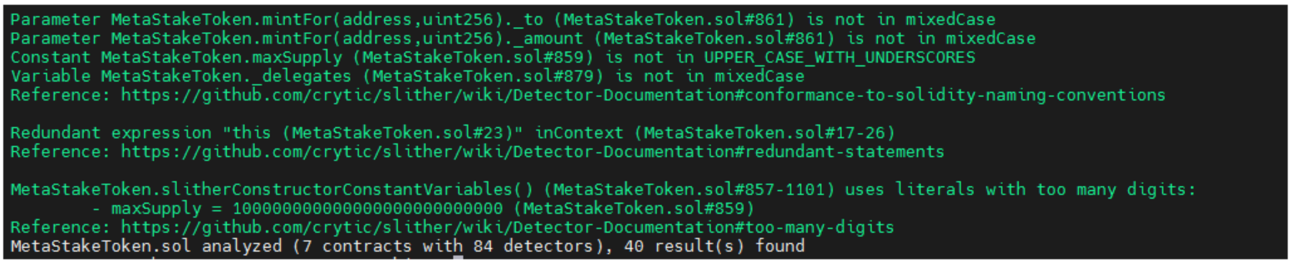
Slither Log >> MetaStakeToken.sol



```
BEP20.allowance(address,address).owner (MetaStakeToken.sol#655) shadows:
        - Ownable.owner() (MetaStakeToken.sol#60-62) (function)
BEP20._approve(address,address,uint256).owner (MetaStakeToken.sol#827) shadows:
        - Ownable.owner() (MetaStakeToken.sol#60-62) (function)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#local-variable-shadowing

MetaStakeToken.delegateBySig(address,uint256,uint256,uint8,bytes32,bytes32) (MetaStakeToken.sol#936-977) uses timestamp for co
mparisons
        Dangerous comparisons:
        - require(bool,string)(block.timestamp <= expiry,CAKE::delegateBySig: signature expired) (MetaStakeToken.sol#975)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#block-timestamp

Address.isContract(address) (MetaStakeToken.sol#404-415) uses assembly
        - INLINE ASM (MetaStakeToken.sol#411-413)
Address._functionCallWithValue(address,bytes,uint256,string) (MetaStakeToken.sol#512-538) uses assembly
        - INLINE ASM (MetaStakeToken.sol#530-533)
MetaStakeToken.getChainId() (MetaStakeToken.sol#1095-1099) uses assembly
        - INLINE ASM (MetaStakeToken.sol#1097)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#assembly-usage

Different versions of Solidity are used:
        - Version used: ['>0.6.6', '>=0.4.0', '>=0.6.0<0.8.0', '>=0.6.6']
        - >0.6.6 (MetaStakeToken.sol#854)
        - >=0.4.0 (MetaStakeToken.sol#95)
        - >=0.4.0 (MetaStakeToken.sol#192)
        - >=0.4.0 (MetaStakeToken.sol#541)
        - >=0.6.0<0.8.0 (MetaStakeToken.sol#5)
        - >=0.6.0<0.8.0 (MetaStakeToken.sol#28)
        - >=0.6.6 (MetaStakeToken.sol#381)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#different-pragma-directives-are-used
```



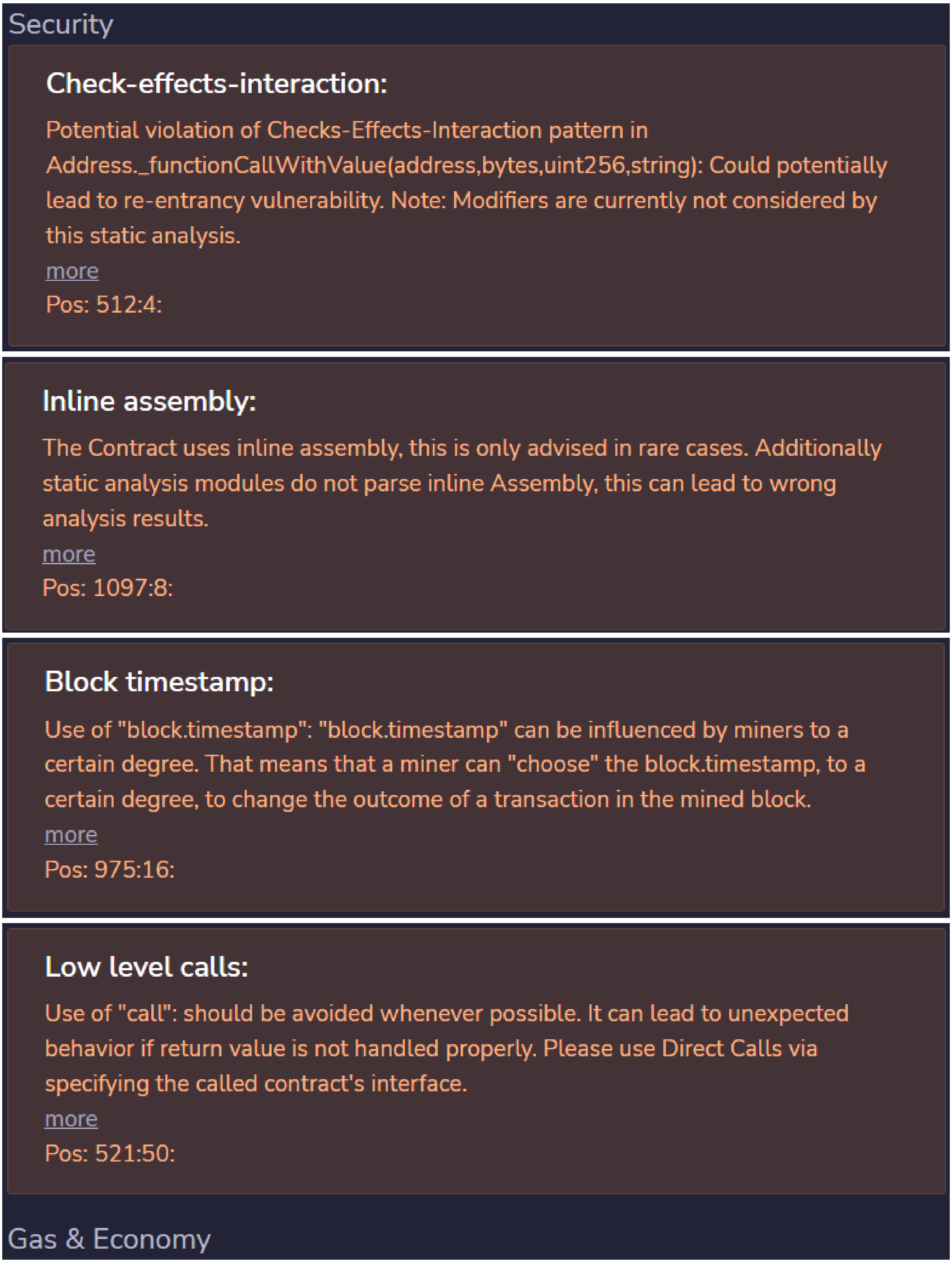
```
Address._functionCallWithValue(address,bytes,uint256,string) (MetaStakeToken.sol#512-538) is never used and should be removed
Address.functionCall(address,bytes) (MetaStakeToken.sol#459-461) is never used and should be removed
Address.functionCall(address,bytes,string) (MetaStakeToken.sol#469-475) is never used and should be removed
Address.functionCallWithValue(address,bytes,uint256) (MetaStakeToken.sol#488-494) is never used and should be removed
Address.functionCallWithValue(address,bytes,uint256,string) (MetaStakeToken.sol#502-510) is never used and should be removed
Address.isContract(address) (MetaStakeToken.sol#404-415) is never used and should be removed
Address.sendValue(address,uint256) (MetaStakeToken.sol#433-439) is never used and should be removed
BEP20._burn(address,uint256) (MetaStakeToken.sol#805-811) is never used and should be removed
BEP20._burnFrom(address,uint256) (MetaStakeToken.sol#844-851) is never used and should be removed
Context._msgData() (MetaStakeToken.sol#22-25) is never used and should be removed
SafeMath.div(uint256,uint256) (MetaStakeToken.sol#296-298) is never used and should be removed
SafeMath.div(uint256,uint256,string) (MetaStakeToken.sol#312-322) is never used and should be removed
SafeMath.min(uint256,uint256) (MetaStakeToken.sol#361-363) is never used and should be removed
SafeMath.mod(uint256,uint256) (MetaStakeToken.sol#336-338) is never used and should be removed
SafeMath.mod(uint256,uint256,string) (MetaStakeToken.sol#352-359) is never used and should be removed
SafeMath.mul(uint256,uint256) (MetaStakeToken.sol#270-282) is never used and should be removed
SafeMath.sqrt(uint256) (MetaStakeToken.sol#366-377) is never used and should be removed
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dead-code

Pragma version>=0.6.0<0.8.0 (MetaStakeToken.sol#5) is too complex
Pragma version>=0.6.0<0.8.0 (MetaStakeToken.sol#28) is too complex
Pragma version>=0.4.0 (MetaStakeToken.sol#95) allows old versions
Pragma version>=0.4.0 (MetaStakeToken.sol#192) allows old versions
Pragma version>=0.6.6 (MetaStakeToken.sol#381) allows old versions
Pragma version>=0.4.0 (MetaStakeToken.sol#541) allows old versions
Pragma version>0.6.6 (MetaStakeToken.sol#854) allows old versions
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity

Low level call in Address.sendValue(address,uint256) (MetaStakeToken.sol#433-439):
        - (success) = recipient.call{value: amount}() (MetaStakeToken.sol#437)
Low level call in Address._functionCallWithValue(address,bytes,uint256,string) (MetaStakeToken.sol#512-538):
        - (success,returndata) = target.call{value: weiValue}(data) (MetaStakeToken.sol#521)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#low-level-calls
```



```
Parameter MetaStakeToken.mintFor(address,uint256)._to (MetaStakeToken.sol#861) is not in mixedCase
Parameter MetaStakeToken.mintFor(address,uint256)._amount (MetaStakeToken.sol#861) is not in mixedCase
Constant MetaStakeToken.maxSupply (MetaStakeToken.sol#859) is not in UPPER_CASE_WITH_UNDERSCORES
Variable MetaStakeToken._delegates (MetaStakeToken.sol#879) is not in mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions

Redundant expression "this (MetaStakeToken.sol#23)" inContext (MetaStakeToken.sol#17-26)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#redundant-statements

MetaStakeToken.slitherConstructorConstantVariables() (MetaStakeToken.sol#857-1101) uses literals with too many digits:
        - maxSupply = 100000000000000000000000000 (MetaStakeToken.sol#859)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#too-many-digits
MetaStakeToken.sol analyzed (7 contracts with 84 detectors), 40 result(s) found
```

Solidity Static Analysis

MetaStakeToken.sol



Security

Check-effects-interaction:

Potential violation of Checks-Effects-Interaction pattern in Address._functionCallWithValue(address,bytes,uint256,string): Could potentially lead to re-entrancy vulnerability. Note: Modifiers are currently not considered by this static analysis.

more

Pos: 512:4:

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: 1097:8:

Block timestamp:

Use of "block.timestamp": "block.timestamp" can be influenced by miners to a certain degree. That means that a miner can "choose" the block.timestamp, to a certain degree, to change the outcome of a transaction in the mined block.

more

Pos: 975:16:

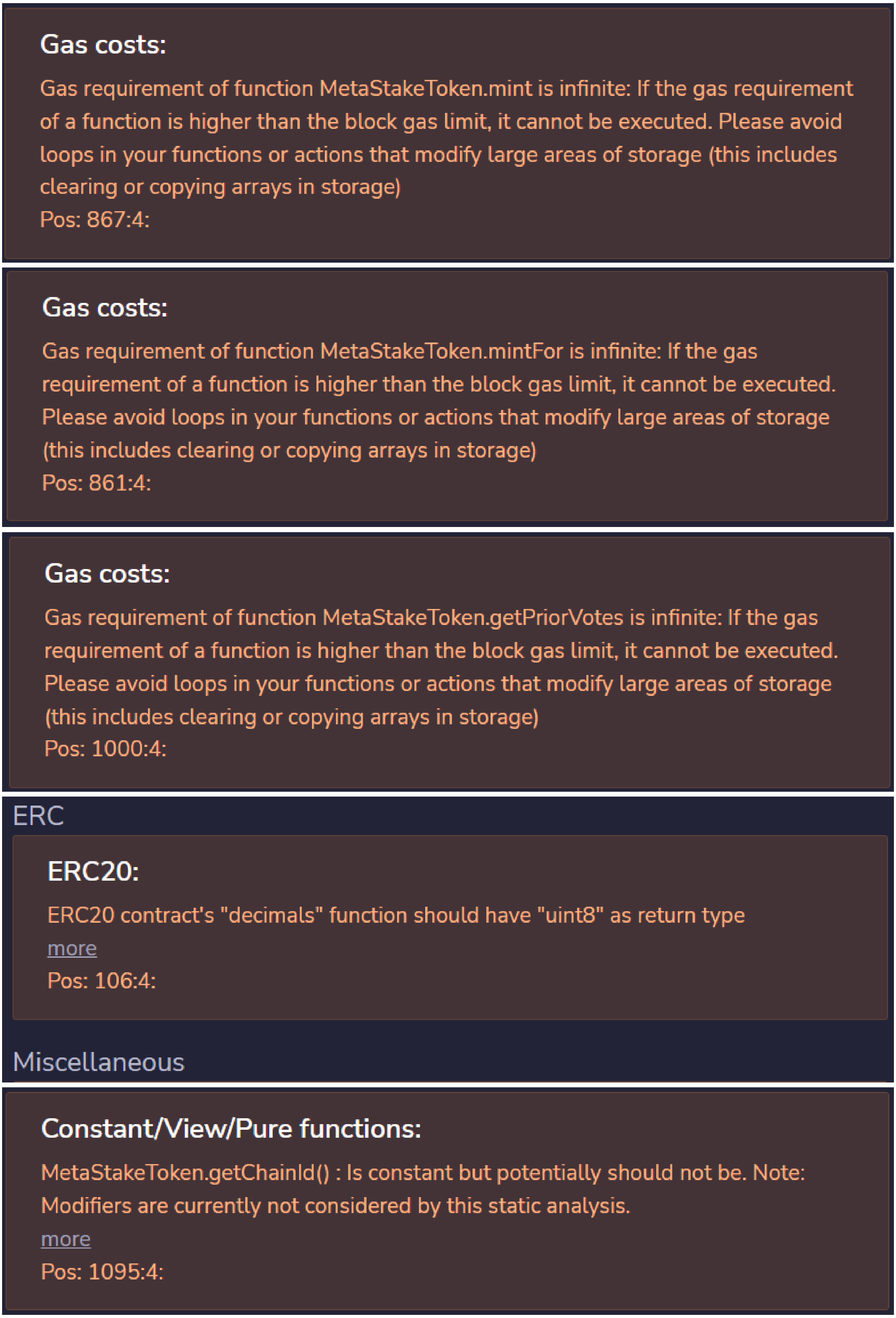
Low level calls:

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: 521:50:

Gas & Economy



Gas costs:

Gas requirement of function MetaStakeToken.mint 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: 867:4:

Gas costs:

Gas requirement of function MetaStakeToken.mintFor 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: 861:4:

Gas costs:

Gas requirement of function MetaStakeToken.getPriorVotes 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: 1000:4:

ERC

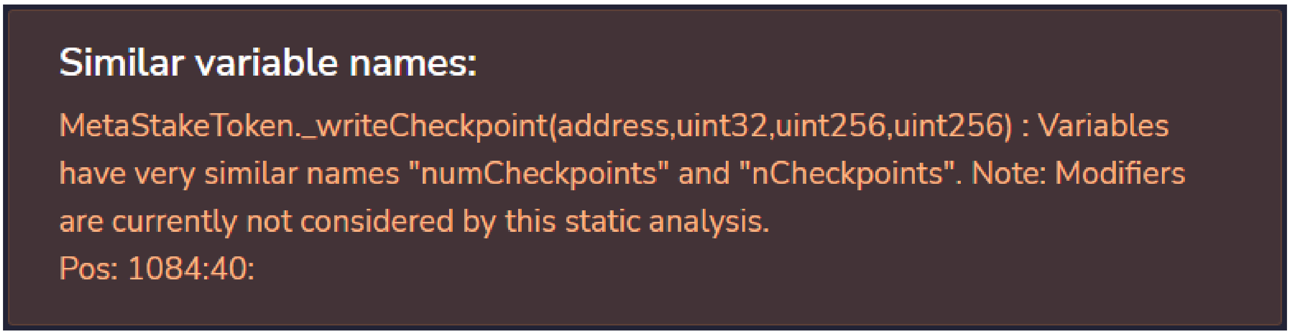
ERC20:

ERC20 contract's "decimals" function should have "uint8" as return type
more
Pos: 106:4:

Miscellaneous

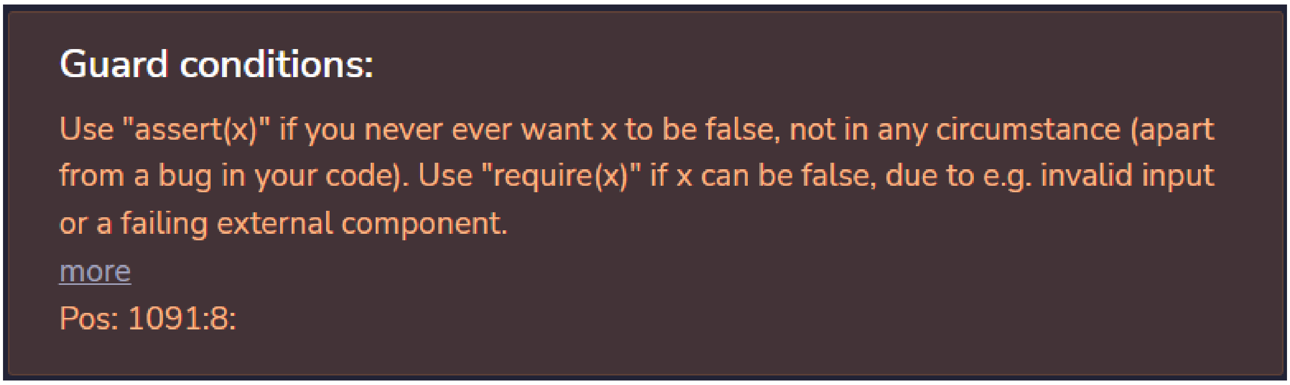
Constant/View/Pure functions:

MetaStakeToken.getChainId() : Is constant but potentially should not be. Note: Modifiers are currently not considered by this static analysis.
more
Pos: 1095:4:



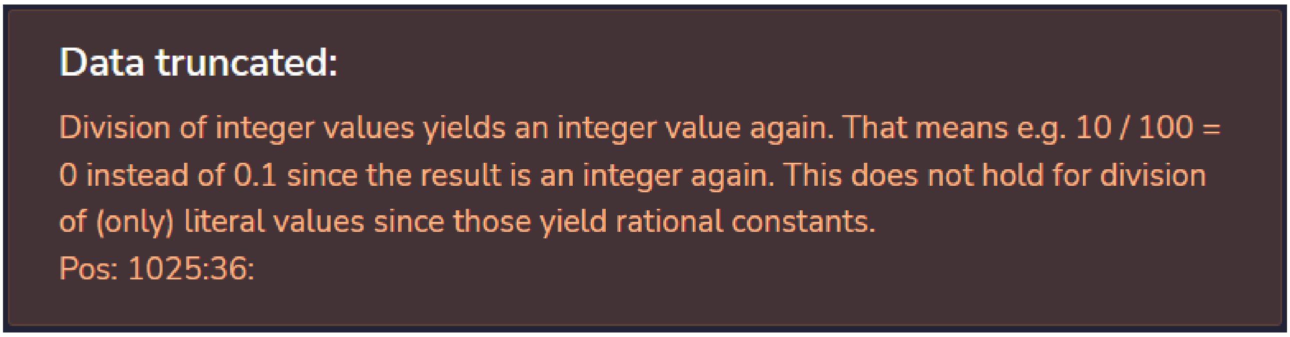
Similar variable names:

MetaStakeToken._writeCheckpoint(address,uint32,uint256,uint256) : Variables have very similar names "numCheckpoints" and "nCheckpoints". Note: Modifiers are currently not considered by this static analysis.
Pos: 1084:40:



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: 1091:8:

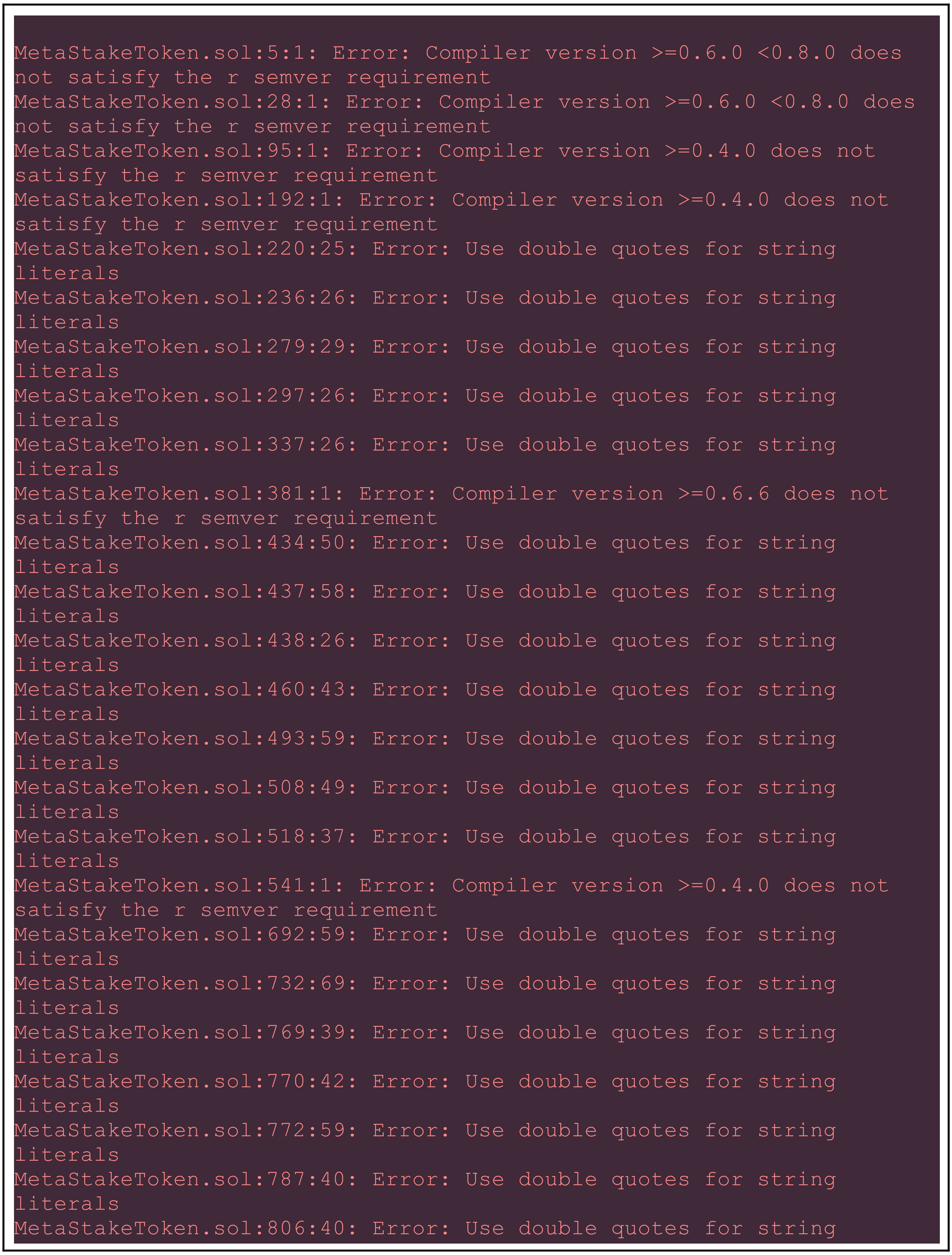


Data truncated:

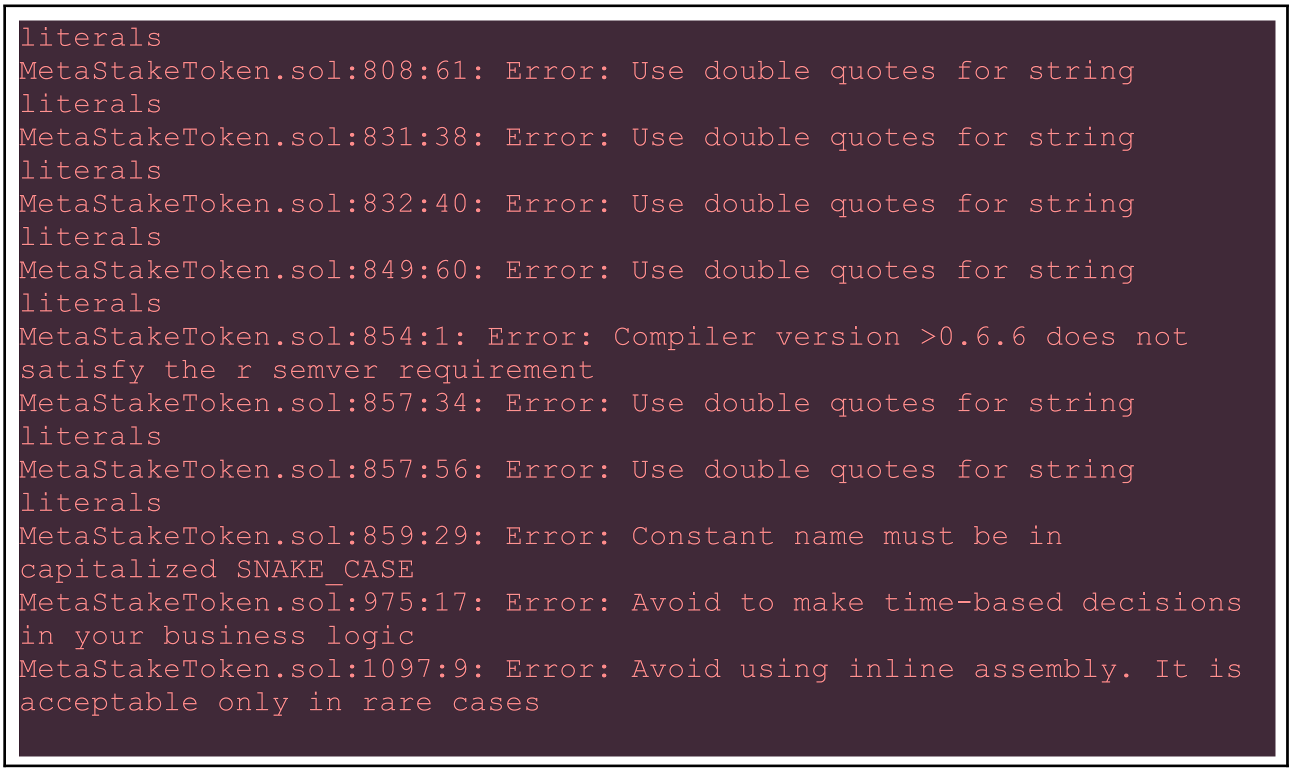
Division of integer values yields an integer value again. That means e.g. 10 / 100 = 0 instead of 0.1 since the result is an integer again. This does not hold for division of (only) literal values since those yield rational constants.
Pos: 1025:36:

Solhint Linter

MetaStakeToken.sol



```
MetaStakeToken.sol:5:1: Error: Compiler version >=0.6.0 <0.8.0 does
not satisfy the r semver requirement
MetaStakeToken.sol:28:1: Error: Compiler version >=0.6.0 <0.8.0 does
not satisfy the r semver requirement
MetaStakeToken.sol:95:1: Error: Compiler version >=0.4.0 does not
satisfy the r semver requirement
MetaStakeToken.sol:192:1: Error: Compiler version >=0.4.0 does not
satisfy the r semver requirement
MetaStakeToken.sol:220:25: Error: Use double quotes for string
literals
MetaStakeToken.sol:236:26: Error: Use double quotes for string
literals
MetaStakeToken.sol:279:29: Error: Use double quotes for string
literals
MetaStakeToken.sol:297:26: Error: Use double quotes for string
literals
MetaStakeToken.sol:337:26: Error: Use double quotes for string
literals
MetaStakeToken.sol:381:1: Error: Compiler version >=0.6.6 does not
satisfy the r semver requirement
MetaStakeToken.sol:434:50: Error: Use double quotes for string
literals
MetaStakeToken.sol:437:58: Error: Use double quotes for string
literals
MetaStakeToken.sol:438:26: Error: Use double quotes for string
literals
MetaStakeToken.sol:460:43: Error: Use double quotes for string
literals
MetaStakeToken.sol:493:59: Error: Use double quotes for string
literals
MetaStakeToken.sol:508:49: Error: Use double quotes for string
literals
MetaStakeToken.sol:518:37: Error: Use double quotes for string
literals
MetaStakeToken.sol:541:1: Error: Compiler version >=0.4.0 does not
satisfy the r semver requirement
MetaStakeToken.sol:692:59: Error: Use double quotes for string
literals
MetaStakeToken.sol:732:69: Error: Use double quotes for string
literals
MetaStakeToken.sol:769:39: Error: Use double quotes for string
literals
MetaStakeToken.sol:770:42: Error: Use double quotes for string
literals
MetaStakeToken.sol:772:59: Error: Use double quotes for string
literals
MetaStakeToken.sol:787:40: Error: Use double quotes for string
literals
MetaStakeToken.sol:806:40: Error: Use double quotes for string
```



```
literals
MetaStakeToken.sol:808:61: Error: Use double quotes for string
literals
MetaStakeToken.sol:831:38: Error: Use double quotes for string
literals
MetaStakeToken.sol:832:40: Error: Use double quotes for string
literals
MetaStakeToken.sol:849:60: Error: Use double quotes for string
literals
MetaStakeToken.sol:854:1: Error: Compiler version >0.6.6 does not
satisfy the r semver requirement
MetaStakeToken.sol:857:34: Error: Use double quotes for string
literals
MetaStakeToken.sol:857:56: Error: Use double quotes for string
literals
MetaStakeToken.sol:859:29: Error: Constant name must be in
capitalized SNAKE_CASE
MetaStakeToken.sol:975:17: Error: Avoid to make time-based decisions
in your business logic
MetaStakeToken.sol:1097:9: Error: Avoid using inline assembly. It is
acceptable only in rare cases
```

Software analysis result:

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

EtherAuthority

Email: audit@EtherAuthority.io