EtherAuthority

www.EtherAuthority.io
audit@etherauthority.io

SMART CONTRACT

Security Audit Report

Project:	Ondo
Website:	ondo.finance
Platform:	Ethereum
Language:	Solidity
Date:	May 18th, 2024

Table of contents

Introduction 4
Project Background 4
Audit Scope 5
Claimed Smart Contract Features 6
Audit Summary 7
Technical Quick Stats 8
Business Risk Analysis 9
Code Quality 10
Documentation 10
Use of Dependencies 10
AS-IS overview 11
Severity Definitions 13
Audit Findings 14
Conclusion 16
Our Methodology 17
Disclaimers 19
Appendix

- Code Flow Diagram 20
- Slither Results Log 21
- Solidity static analysis 23
- Solhint Linter 25

`

Introduction

As part of EtherAuthority's community smart contracts audit initiatives, the Ondo smart contract from ondo.finance was audited extensively. The audit has been performed using manual analysis as well as using automated software tools. This report presents all the findings regarding the audit performed on May 18th, 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 contract named "Ondo" implements a token with time-based vesting functionalities. Let's break down its key components:
  - **Context:** Abstract contract defining internal functions _msgSender() and _msgData() to retrieve the sender's address and the data of the current message, respectively.
  - **IERC165:** Interface for contract-level support for ERC165.
  - **IAccessControl:** Interface defining functions related to access control such as role management.
  - **AccessControl:** Abstract contract implementing access control functionality based on roles. It manages roles and role membership.
  - **IOndo:** Interface defining functions related to Ondo token features such as updating tranche balances and retrieving balances.
  - **LinearTimelock:** Abstract contract implementing linear time-based vesting logic. It calculates vested balances based on elapsed time since the cliff period.
  - **Ondo:** The main contract implementing the Ondo token. It inherits from AccessControl and LinearTimelock. Key functionalities include:
    - Token metadata such as name, symbol, and decimals.
    - Management of total supply and balances.
    - Delegation functionality allows token holders to delegate voting power.

    - Minting functionality is restricted to accounts with the MINTER_ROLE.
    - Transfer and approval functionalities, with the ability to enable/disable transfers.
    - Time-based vesting functionality for specific tranches of investors.
    - Role-based access control for various operations.
    - Update of cliff timestamp for vesting.
- This contract provides a comprehensive implementation of a token with access control and time-based vesting features, suitable for scenarios where token release needs to be controlled over time.

Audit scope

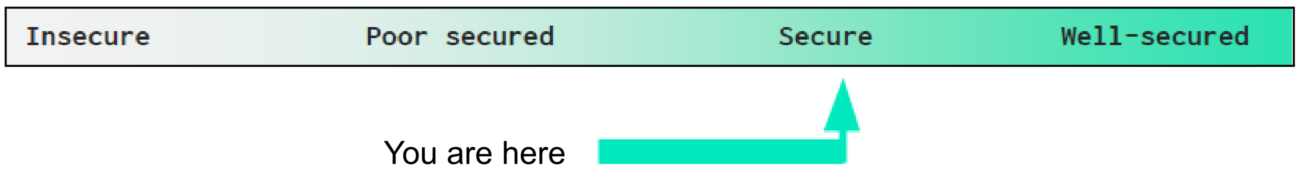
Name	**Code Review and Security Analysis Report for Ondo Smart Contract**
Platform	**Ethereum**
Language	**Solidity**
File	Ondo.sol
Smart Contract Code	0xfaba6f8e4a5e8ab82f62fe7c39859fa577269be3
Audit Date	April 18th, 2024

Claimed Smart Contract Features

Claimed Feature Detail	Our Observation
Tokenomics: • Name: Ondo • Symbol: ONDO • Decimals: 18 • Total Supply: 10 billion Ondo	**YES, This is valid.**

Audit Summary

According to the standard audit assessment, the Customer`s solidity-based smart contracts are **"Secured"**.Also, these contracts contain owner control, which does not make them 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. The general overview is presented in the AS-IS section and all identified issues can be found in the Audit overview section.

We found 0 critical, 0 high, 0 medium, 0 low and 2 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	The solidity version is not specified	Passed
	Solidity version is too old	Passed
	Integer overflow/underflow	Passed
	Function input parameters lack check	Passed
	Function input parameters check bypass	Passed
	Function access control lacks management	Passed
	Critical operation lacks event log	Passed
	Human/contract checks bypass	Passed
	Random number generation/use vulnerability	N/A
	Fallback function misuse	Passed
	Race condition	Passed
	Logical vulnerability	Passed
	Features claimed	Passed
	Other programming issues	Moderated
Code Specification	Function visibility not explicitly declared	Passed
	Var. storage location not explicitly declared	Passed
	Use keywords/functions to be deprecated	Passed
	Unused code	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 is not set	Passed
	“Short Address” Attack	Passed
	“Double Spend” Attack	Passed

Overall Audit Result: **PASSED**

Business Risk Analysis

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

Overall Audit Result: **PASSED**

Code Quality

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

The libraries in Ondo 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 Ondo.

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 Ondo 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 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

Ondo.sol

Functions

Sl.	Functions	Type	Observation	Conclusion
1	constructor	write	Passed	No Issue
2	allowance	external	Passed	No Issue
3	approve	external	Passed	No Issue
4	balanceOf	external	Passed	No Issue
5	getFreedBalance	external	Passed	No Issue
6	transfer	external	Passed	No Issue
7	transferFrom	external	Passed	No Issue
8	delegate	write	Passed	No Issue
9	delegateBySig	write	Passed	No Issue
10	getCurrentVotes	external	Passed	No Issue
11	getPriorVotes	read	Passed	No Issue
12	mint	external	Passed	No Issue
13	_delegate	internal	Passed	No Issue
14	_transferTokens	internal	Passed	No Issue
15	_moveDelegates	internal	Passed	No Issue
16	_writeCheckpoint	internal	Passed	No Issue
17	getChainId	internal	Assembly	Refer Audit Findings
18	enableTransfer	external	Passed	No Issue
19	updateTrancheBalance	external	Passed	No Issue
20	_getFreedBalance	internal	Passed	No Issue
21	updateCliffTimestamp	external	Centralization	Refer Audit Findings
22	passedCliff	read	Passed	No Issue
23	passedAllVestingPeriods	read	Passed	No Issue
24	getVestedBalance	external	Passed	No Issue
25	_getTrancheInfo	internal	Passed	No Issue
26	_proportionAvailable	internal	Passed	No Issue
27	safe32	internal	Passed	No Issue
28	safe96	internal	Passed	No Issue
29	add96	internal	Passed	No Issue
30	sub96	internal	Passed	No Issue
31	supportsInterface	read	Passed	No Issue
32	hasRole	read	Passed	No Issue
33	getRoleAdmin	read	Passed	No Issue
34	grantRole	write	Passed	No Issue
35	revokeRole	write	Passed	No Issue
36	renounceRole	write	Passed	No Issue
37	_setupRole	internal	Passed	No Issue
38	_setRoleAdmin	internal	Passed	No Issue
39	_grantRole	write	Passed	No Issue

40	_revokeRole	write	Passed	No Issue
41	supportsInterface	read	Passed	No Issue
42	_msgSender	internal	Passed	No Issue
43	_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 a significant impact on smart contract execution, e.g. public access to crucial
Medium	Medium-level vulnerabilities are important to fix; however, they can't lead to tokens lose
Low	Low-level vulnerabilities are mostly related to outdated, unused etc. code snippets, that can't have a significant impact on execution
Lowest / Code Style / Best Practice	Lowest-level vulnerabilities, code style violations and info statements can't affect smart contract execution and can be ignored.

Audit Findings

Critical Severity

No Critical severity vulnerabilities were found.

High Severity

No High severity vulnerabilities were found.

Medium

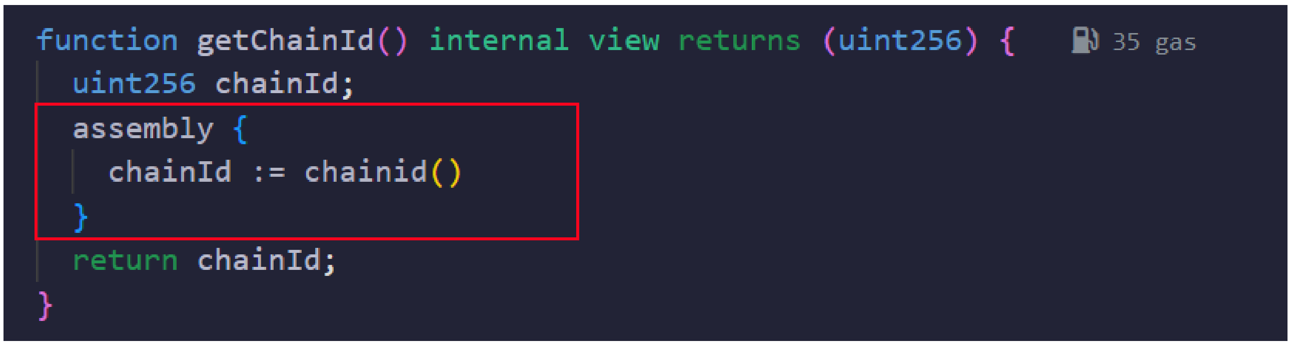
No Medium Severity vulnerabilities were found.

Low

No Low Severity vulnerabilities were found.

Very Low / Informational / Best practices:

(1) Assembly:

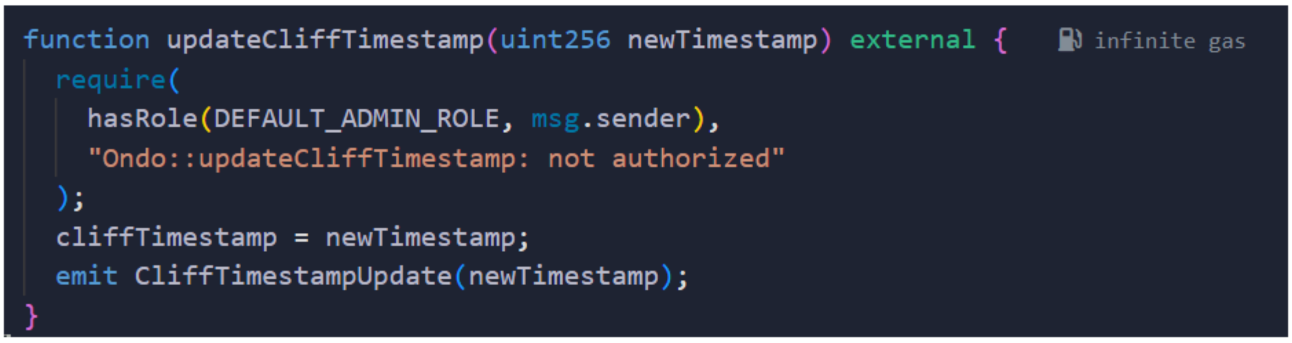


```
function getChainId() internal view returns (uint256) {    35 gas
  uint256 chainId;
  assembly {
    chainId := chainid()
  }
  return chainId;
}
```

Using assembly can be useful for optimizing code, but it can also be error-prone. It's important to carefully test and debug assembly code to ensure that it is correct and does not contain any errors.

Resolution: It is recommended to use assembly only when necessary.

(2) Centralization:



```
function updateCliffTimestamp(uint256 newTimestamp) external {    infinite gas
  require(
    hasRole(DEFAULT_ADMIN_ROLE, msg.sender),
    "Ondo::updateCliffTimestamp: not authorized"
  );
  cliffTimestamp = newTimestamp;
  emit CliffTimestampUpdate(newTimestamp);
}
```

DEFAULT_ADMIN_ROLE can change cliffTimestamp.

Resolution: This function can be by DEFAULT_ADMIN_ROLE. So we strongly recommend removing this function.

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. The following are Admin functions:

AccessControl.sol

- grantRole: Grants `role` to `account` can be set by the admin.
- revokeRole: Revokes `role` from `account` by the admin.
- renounceRole: Renounce Role from `account` by the admin.

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

Conclusion

We were given a contract code in the form of Etherscan web links. And we have used all possible tests based on given objects as files. We observed 2 Informational issues in the smart contracts. but those are not critical. So, **it's good to go for the production**.

Since possible test cases can be unlimited for such smart contracts protocol, we provide no such guarantee of future outcomes. We have used all the latest static tools and manual observations to cover the maximum possible test cases to scan everything.

Smart contracts within the scope were manually reviewed and analyzed with static analysis tools. Smart Contract's high-level description of functionality was presented in the As-is overview section of the report.

The audit report contains all found security vulnerabilities and other issues in the reviewed code.

The security state of the reviewed smart contract, based on standard audit procedure scope, is **"Secured".**

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of the systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review:
In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis:
Our audit techniques included manual code analysis, user interface interaction, and white box penetration testing. We look at the project's 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).

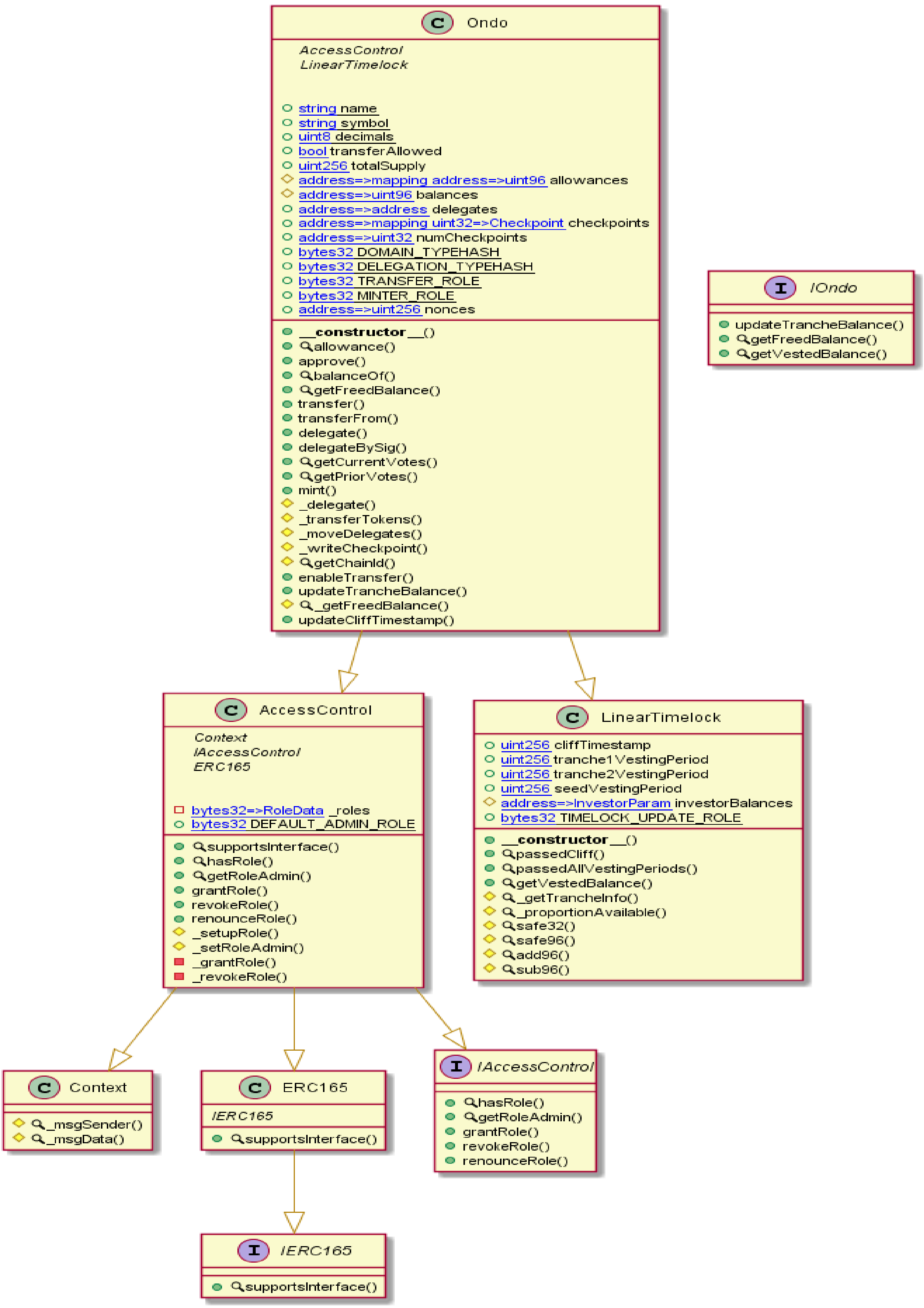
Due to the fact that the total number of test cases is unlimited, the audit makes no statements or warranties on security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bugfree status, or any other statements of the contract. While we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

Appendix

Code Flow Diagram - Ondo

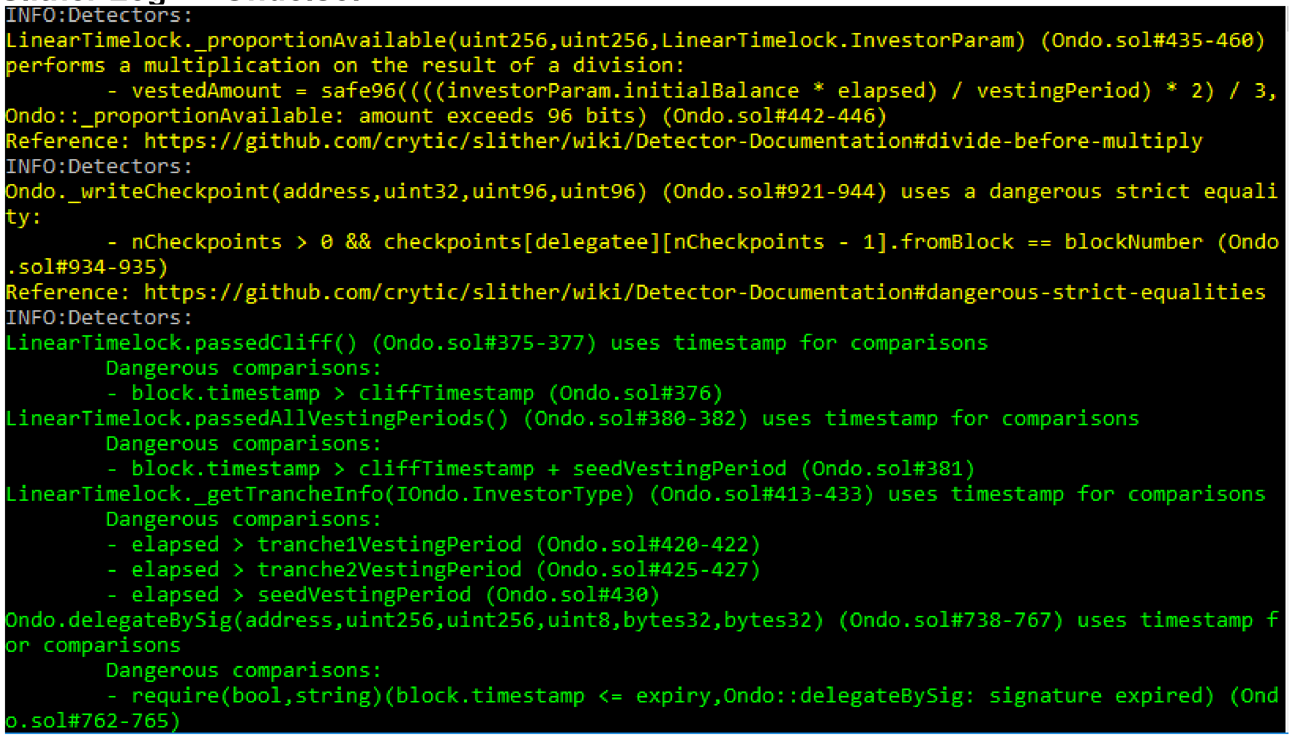


Slither Results Log

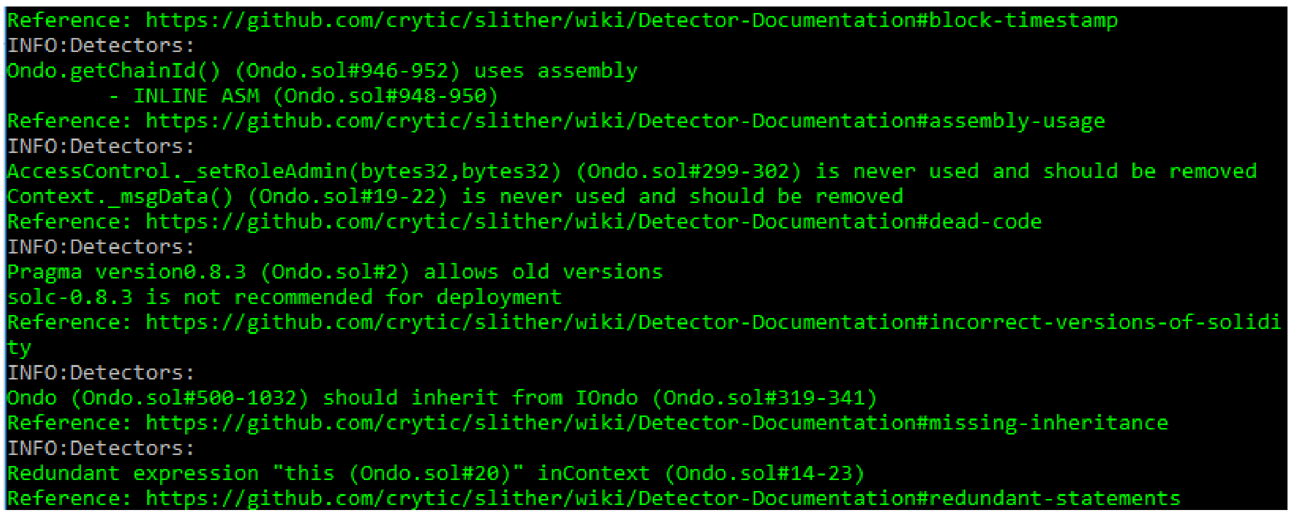
Slither is a Solidity static analysis framework that uses vulnerability detectors, displays contract details, and provides an API for writing custom analyses. It helps developers identify vulnerabilities, improve code comprehension, and prototype custom analyses quickly. The analysis includes a report with warnings and errors, allowing developers to quickly prototype and fix issues.

We did the analysis of the project altogether. Below are the results.

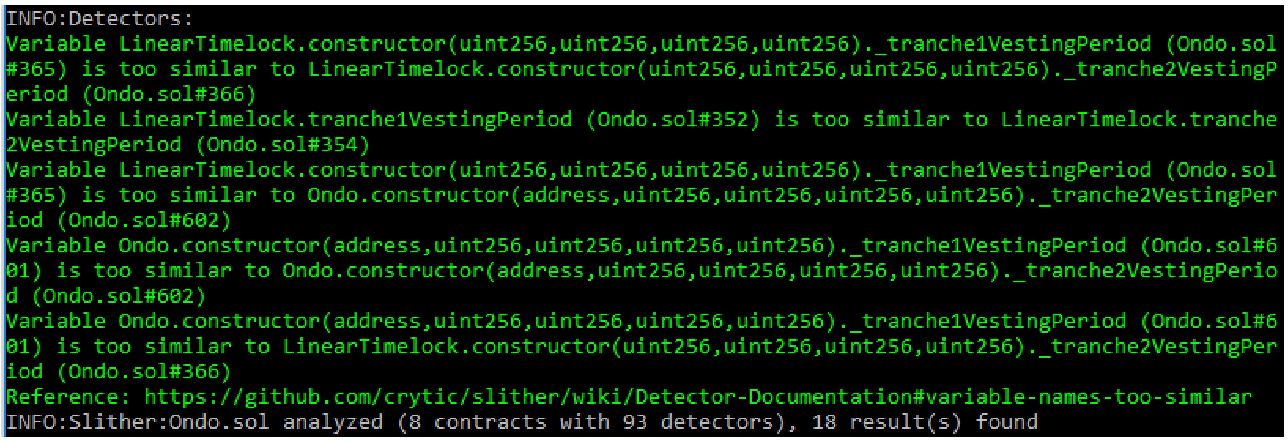
Slither Log >> Ondo.sol



```
INFO:Detectors:
LinearTimelock._proportionAvailable(uint256,uint256,LinearTimelock.InvestorParam) (Ondo.sol#435-460)
performs a multiplication on the result of a division:
        - vestedAmount = safe96((((investorParam.initialBalance * elapsed) / vestingPeriod) * 2) / 3,
Ondo::_proportionAvailable: amount exceeds 96 bits) (Ondo.sol#442-446)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#divide-before-multiply
INFO:Detectors:
Ondo._writeCheckpoint(address,uint32,uint96,uint96) (Ondo.sol#921-944) uses a dangerous strict equali
ty:
        - nCheckpoints > 0 && checkpoints[delegatee][nCheckpoints - 1].fromBlock == blockNumber (Ondo
.sol#934-935)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dangerous-strict-equalities
INFO:Detectors:
LinearTimelock.passedCliff() (Ondo.sol#375-377) uses timestamp for comparisons
        Dangerous comparisons:
        - block.timestamp > cliffTimestamp (Ondo.sol#376)
LinearTimelock.passedAllVestingPeriods() (Ondo.sol#380-382) uses timestamp for comparisons
        Dangerous comparisons:
        - block.timestamp > cliffTimestamp + seedVestingPeriod (Ondo.sol#381)
LinearTimelock._getTrancheInfo(IOndo.InvestorType) (Ondo.sol#413-433) uses timestamp for comparisons
        Dangerous comparisons:
        - elapsed > tranche1VestingPeriod (Ondo.sol#420-422)
        - elapsed > tranche2VestingPeriod (Ondo.sol#425-427)
        - elapsed > seedVestingPeriod (Ondo.sol#430)
Ondo.delegateBySig(address,uint256,uint256,uint8,bytes32,bytes32) (Ondo.sol#738-767) uses timestamp f
or comparisons
        Dangerous comparisons:
        - require(bool,string)(block.timestamp <= expiry,Ondo::delegateBySig: signature expired) (Ond
o.sol#762-765)
```



```
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#block-timestamp
INFO:Detectors:
Ondo.getChainId() (Ondo.sol#946-952) uses assembly
        - INLINE ASM (Ondo.sol#948-950)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#assembly-usage
INFO:Detectors:
AccessControl._setRoleAdmin(bytes32,bytes32) (Ondo.sol#299-302) is never used and should be removed
Context._msgData() (Ondo.sol#19-22) is never used and should be removed
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dead-code
INFO:Detectors:
Pragma version0.8.3 (Ondo.sol#2) allows old versions
solc-0.8.3 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidi
ty
INFO:Detectors:
Ondo (Ondo.sol#500-1032) should inherit from IOndo (Ondo.sol#319-341)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#missing-inheritance
INFO:Detectors:
Redundant expression "this (Ondo.sol#20)" inContext (Ondo.sol#14-23)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#redundant-statements
```

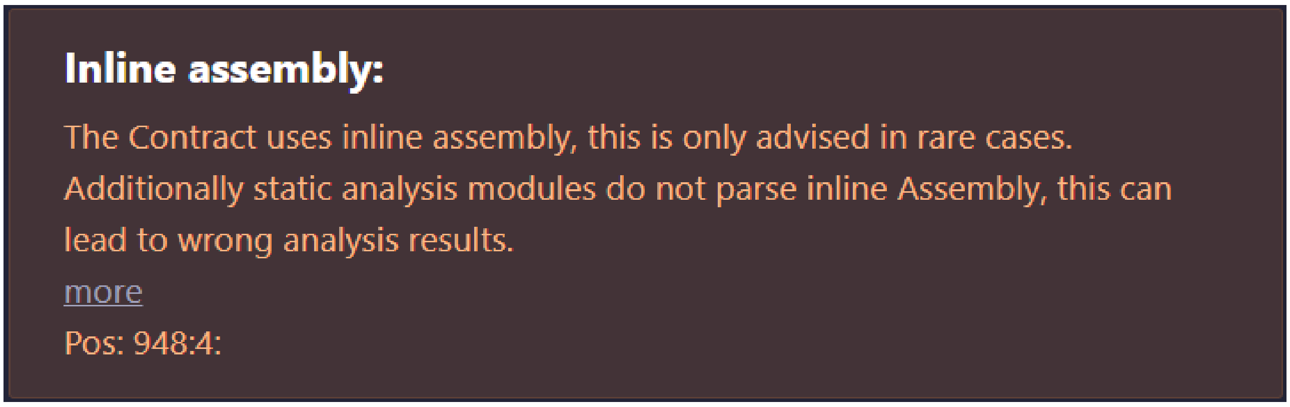


```
INFO:Detectors:
Variable LinearTimelock.constructor(uint256,uint256,uint256,uint256)._tranche1VestingPeriod (Ondo.sol
#365) is too similar to LinearTimelock.constructor(uint256,uint256,uint256,uint256)._tranche2VestingP
eriod (Ondo.sol#366)
Variable LinearTimelock.tranche1VestingPeriod (Ondo.sol#352) is too similar to LinearTimelock.tranche
2VestingPeriod (Ondo.sol#354)
Variable LinearTimelock.constructor(uint256,uint256,uint256,uint256)._tranche1VestingPeriod (Ondo.sol
#365) is too similar to Ondo.constructor(address,uint256,uint256,uint256,uint256)._tranche2VestingPer
iod (Ondo.sol#602)
Variable Ondo.constructor(address,uint256,uint256,uint256,uint256)._tranche1VestingPeriod (Ondo.sol#6
01) is too similar to Ondo.constructor(address,uint256,uint256,uint256,uint256)._tranche2VestingPerio
d (Ondo.sol#602)
Variable Ondo.constructor(address,uint256,uint256,uint256,uint256)._tranche1VestingPeriod (Ondo.sol#6
01) is too similar to LinearTimelock.constructor(uint256,uint256,uint256,uint256)._tranche2VestingPer
iod (Ondo.sol#366)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#variable-names-too-similar
INFO:Slither:Ondo.sol analyzed (8 contracts with 93 detectors), 18 result(s) found
```

Solidity Static Analysis

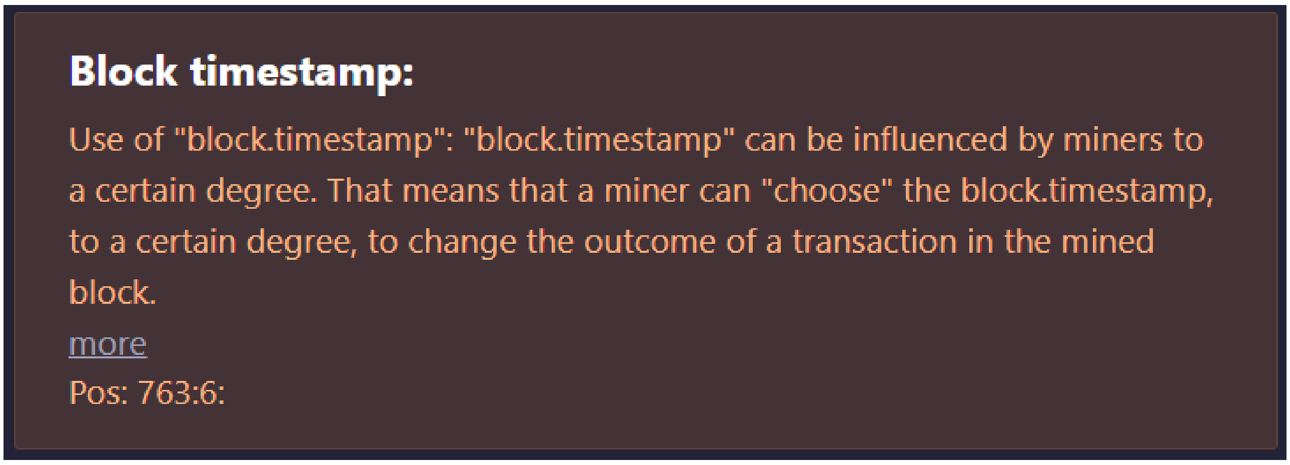
Static code analysis is used to identify many common coding problems before a program is released. It involves examining the code manually or using tools to automate the process. Static code analysis tools can automatically scan the code without executing it.

Ondo.sol



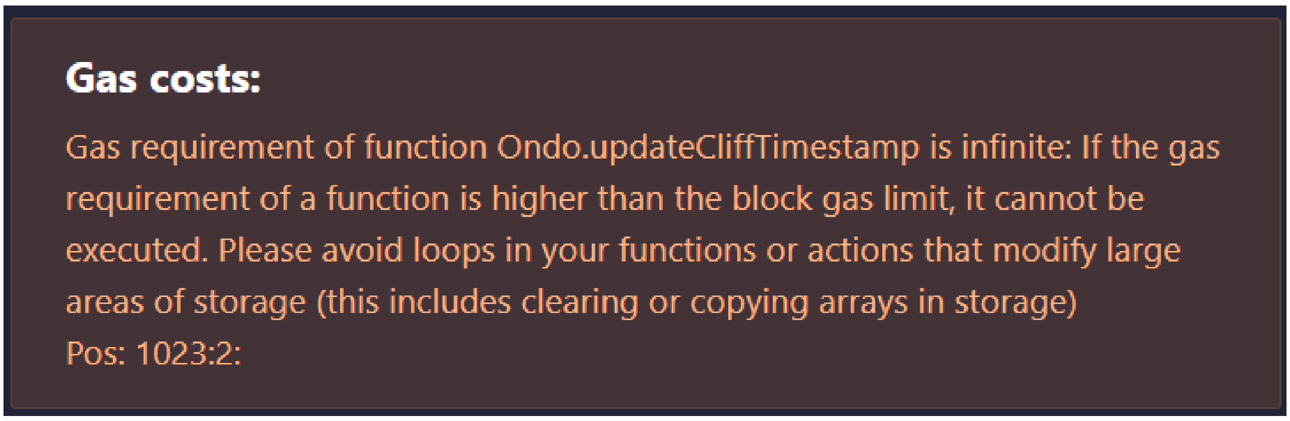
Inline assembly:

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: 948:4:



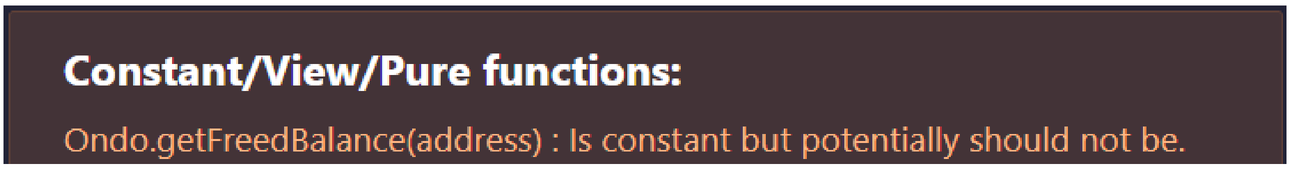
Block timestamp:

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: 763:6:



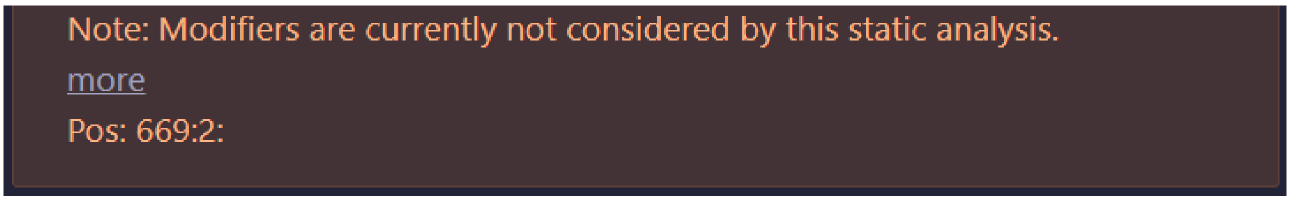
Gas costs:

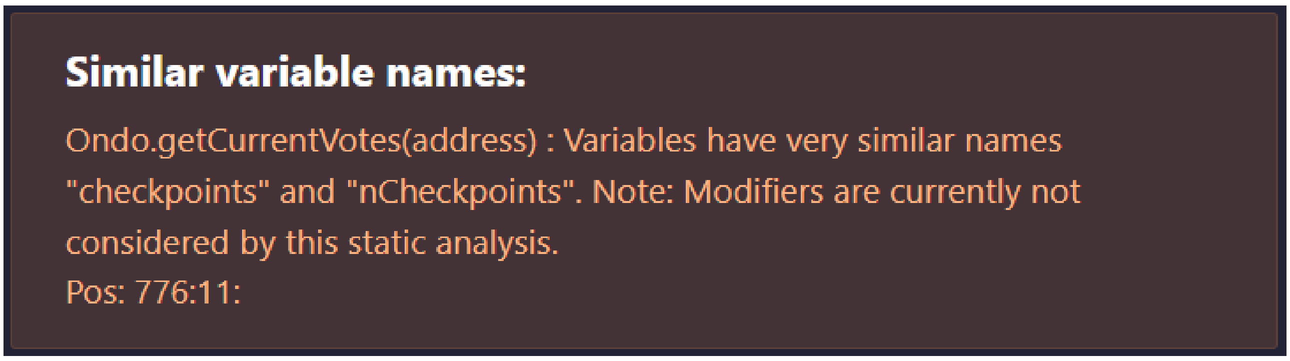
Gas requirement of function Ondo.updateCliffTimestamp is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)
Pos: 1023:2:



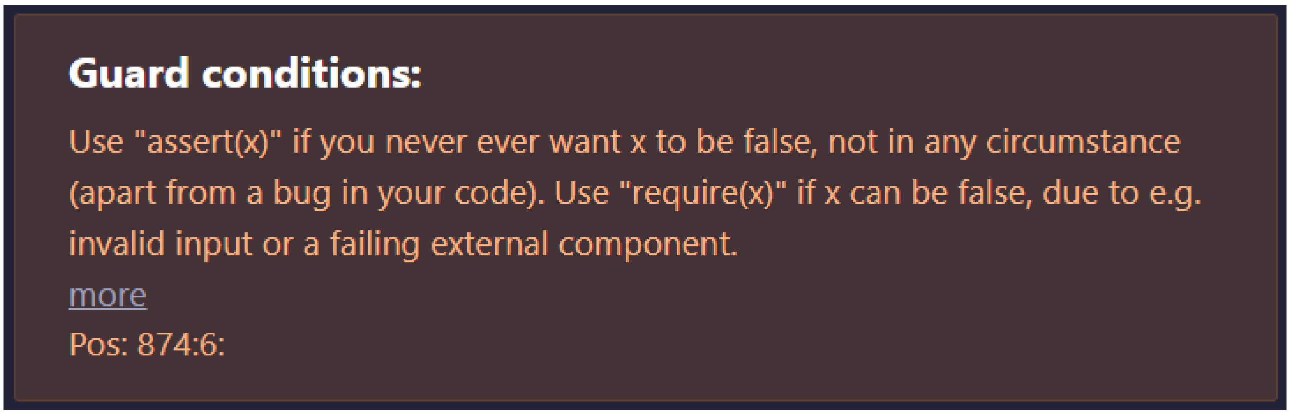
Constant/View/Pure functions:

Ondo.getFreedBalance(address) : Is constant but potentially should not be.

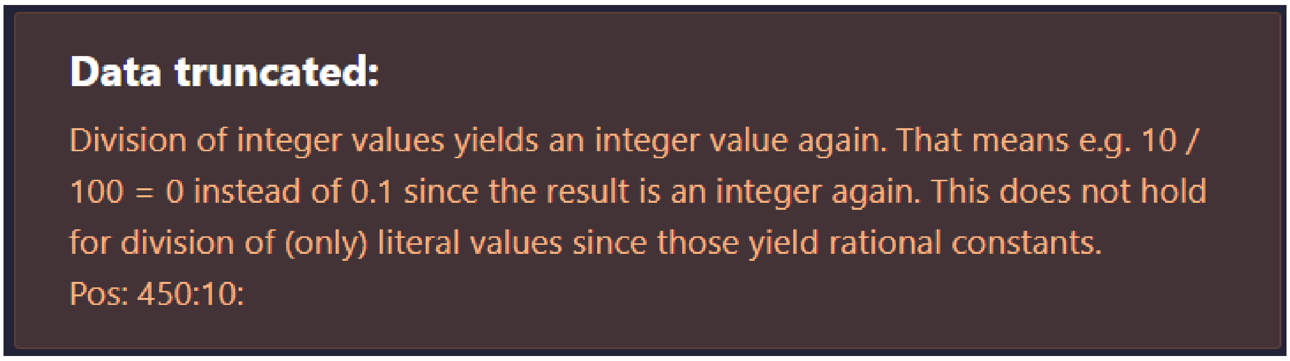

Note: Modifiers are currently not considered by this static analysis.
more
Pos: 669:2:


Similar variable names:

Ondo.getCurrentVotes(address) : Variables have very similar names "checkpoints" and "nCheckpoints". Note: Modifiers are currently not considered by this static analysis.
Pos: 776:11:


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: 874:6:

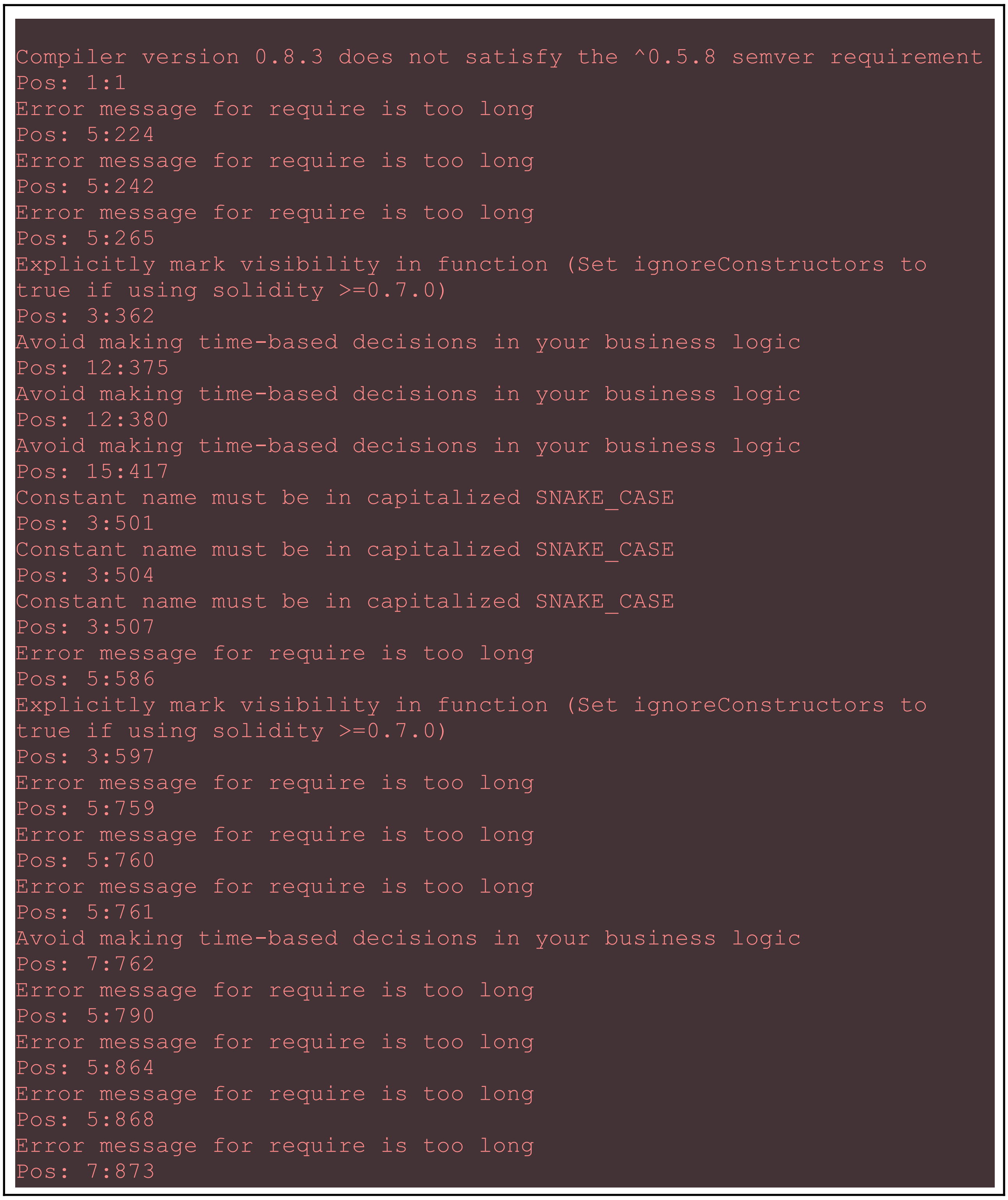

Data truncated:

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: 450:10:

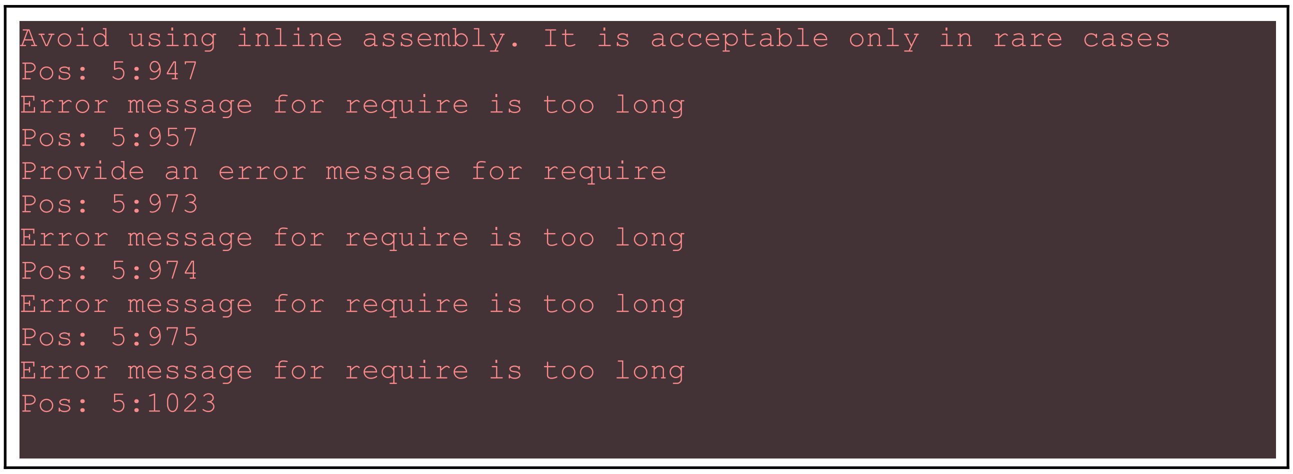
Solhint Linter

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.

Ondo.sol



```
Compiler version 0.8.3 does not satisfy the ^0.5.8 semver requirement
Pos: 1:1
Error message for require is too long
Pos: 5:224
Error message for require is too long
Pos: 5:242
Error message for require is too long
Pos: 5:265
Explicitly mark visibility in function (Set ignoreConstructors to
true if using solidity >=0.7.0)
Pos: 3:362
Avoid making time-based decisions in your business logic
Pos: 12:375
Avoid making time-based decisions in your business logic
Pos: 12:380
Avoid making time-based decisions in your business logic
Pos: 15:417
Constant name must be in capitalized SNAKE_CASE
Pos: 3:501
Constant name must be in capitalized SNAKE_CASE
Pos: 3:504
Constant name must be in capitalized SNAKE_CASE
Pos: 3:507
Error message for require is too long
Pos: 5:586
Explicitly mark visibility in function (Set ignoreConstructors to
true if using solidity >=0.7.0)
Pos: 3:597
Error message for require is too long
Pos: 5:759
Error message for require is too long
Pos: 5:760
Error message for require is too long
Pos: 5:761
Avoid making time-based decisions in your business logic
Pos: 7:762
Error message for require is too long
Pos: 5:790
Error message for require is too long
Pos: 5:864
Error message for require is too long
Pos: 5:868
Error message for require is too long
Pos: 7:873
```



```
Avoid using inline assembly. It is acceptable only in rare cases
Pos: 5:947
Error message for require is too long
Pos: 5:957
Provide an error message for require
Pos: 5:973
Error message for require is too long
Pos: 5:974
Error message for require is too long
Pos: 5:975
Error message for require is too long
Pos: 5:1023
```

Software analysis result:

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