QuillAudits

Audit Report

October, 2020

OPEN

Contents

Introduction

This Audit Report mainly focuses on the overall security of Open Governance Token Smart Contract. With this report, we have tried to ensure the reliability and correctness of their smart contract by complete and rigorous assessment of their system's architecture and the smart contract codebase.

Auditing Approach and Methodologies applied

The Quillhash team has performed rigorous testing of the project starting with analysing the code design patterns in which we reviewed the smart contract architecture to ensure it is structured and safe use of third party smart contracts and libraries.

Our team then performed a formal line by line inspection of the Smart Contract to find any potential issue like race conditions, transaction-ordering dependence, timestamp dependence, and denial of service attacks.

The code was tested in collaboration of our multiple team members and this included -

- Testing the functionality of the Smart Contract to determine proper logic has been followed throughout the process.
- Analysing the complexity of the code by thorough, manual review of the code, line-by-line.
- Deploying the code on testnet using multiple clients to run live tests
- Analysing failure preparations to check how the Smart Contract performs in case of bugs and vulnerabilities.
- Checking whether all the libraries used in the code are on the latest version.
- Analysing the security of the on-chain data.

Audit Details

Project Name: OpenDAO
Website/Etherscan Code(Mainnet): 0x69e8b9528CABDA89fe846C67675B5D73d463a916

Languages: Solidity (Smart contract)
Platforms and Tools: Remix IDE, SmartCheck, VScode, Securify, Mythril, Slither

The contract logic was an exact match with Compound Governance Token and therefore can be labelled as well audited and tested contract. Any non severe warning found during automated audit of these contracts will be of low concern.

Audit Goals

The focus of the audit was to verify that the smart contract system is secure, resilient and working according to its specifications. The audit activities can be grouped in the following three categories:

Security
Identifying security related issues within each contract and the system of contracts.

Sound Architecture
Evaluation of the architecture of this system through the lens of established smart contract best practices and general software best practices.

Code Correctness and Quality
A full review of the contract source code. The primary areas of focus include:

- Correctness
- Sections of code with high complexity
- Readability
- Quantity and quality of test coverage

Security

Every issue in this report was assigned a severity level from the following:

High severity issues
Issues on this level are critical to the smart contract's performance/ functionality and should be fixed before moving to a live environment.

Medium severity issues
They could potentially bring problems add should eventually be fixed.

Low severity issues
Issues on this level are minor details and warnings that can remain unfixed but would be better fixed at some point in the future.

Number of issues per severity

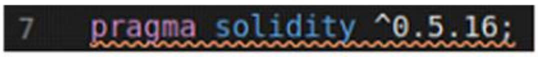
	Low	Medium	High
Open	5	0	0
Closed	0	0	0

Manual Audit

For this section the code was tested/read line by line by our developers. We also used Remix IDE’s JavaScript VM to test the contract functionality.

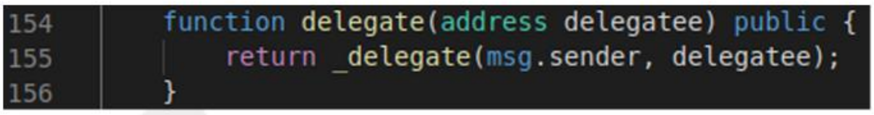
Low Level Severity Issues

1. It is a good practice to lock the solidity version for a live deployment (use 0.5.16 instead of ^0.5.16). Contracts should be deployed with the same compiler version and flags that they have been tested the most with. Locking the pragma helps ensure that contracts do not accidentally get deployed using, for example, the latest compiler which may have higher risks of undiscovered bugs. Contracts may also be deployed by others and the pragma indicates the compiler version intended by the original authors.



```
pragma solidity ^0.5.16;
```

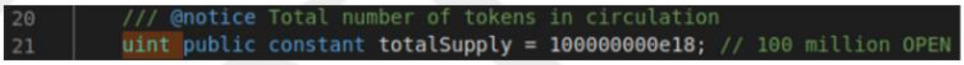
2. Functions which are declared as public and not called internally in the contract can be declared as external for two reasons: 1) Gas usage is lower in an external function compared to public and 2) Increases code readability.



```
    function delegate(address delegatee) public {
        return _delegate(msg.sender, delegatee);
    }
```

Another instance is the delegateBySig function.

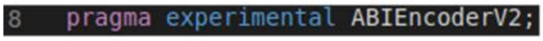
3. Throughout the code base, some variables are declared as uint. To favor explicitness, consider changing all instances of uint to uint256. For example:



```
    /// @notice Total number of tokens in circulation
    uint public constant totalSupply = 100000000e18; // 100 million OPEN
```

4. Trust in smart contract can be better established if their source code is available. Since making source code available always touches on legal problems with regards to copyright, it is recommended to use SPDX license identifiers. Every source file should start with a comment indicating its license:
// SPDX-License-Identifier: MIT

5. It is not a good practice to use experimental features in live deployments. The ABIEncoderV2 is not considered experimental as of solidity version 0.6.0 but you still have to explicitly activate it using pragma experimental ABIEncoderV2 (you can read about it in detail here).



```
pragma experimental ABIEncoderV2;
```

Medium Severity Issues

No medium severity issues

High severity issues

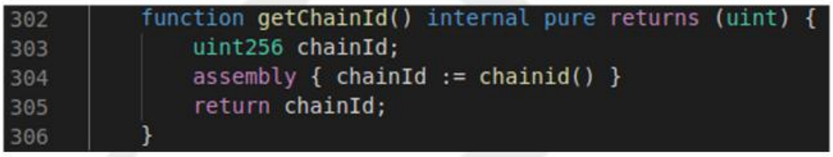
No high severity issues

Recommendations

1. Consider adding Solidity syntax highlighting to the GitHub repository. This helps improve readability for users inspecting contract code on GitHub.

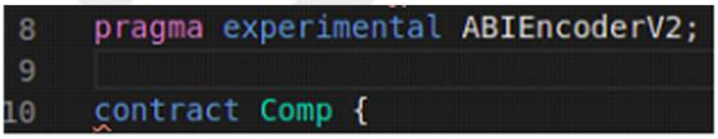
2. The contract name can be updated to Open or any other name of your choice to suit the organization/token it represents.

3. Use of inline assembly should be avoided whenever possible. Inline assembly is a way to access the Ethereum Virtual Machine at a low level. This bypasses several important safety features and checks of Solidity. You should only use it for tasks that need it, and only if you are confident with using it as per the Solidity's documentation.



```
    function getChainId() internal pure returns (uint) {
        uint256 chainId;
        assembly { chainId := chainid() }
        return chainId;
    }
```

4. Top level declarations must be surrounded by two blank lines as per olidity's style guide (you can read more in detail here).

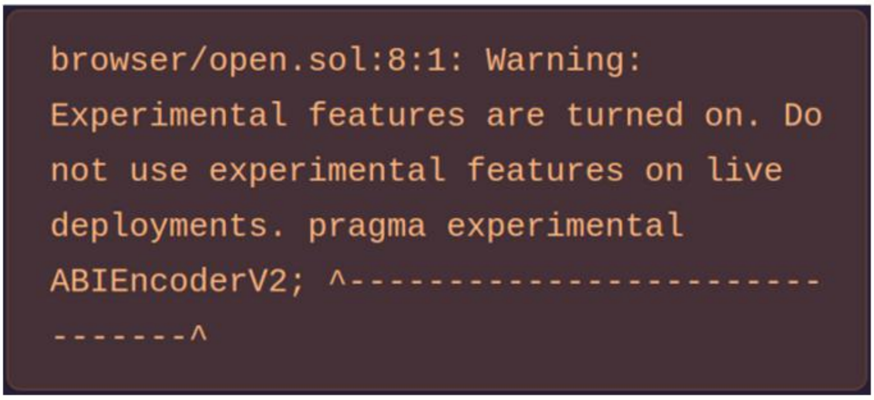


```
pragma experimental ABIEncoderV2;

contract Comp {
```

Automated Audit

Remix Compiler Warnings

It throws warnings by Solidity's compiler. If it encounters any errors the contract cannot be compiled and deployed.

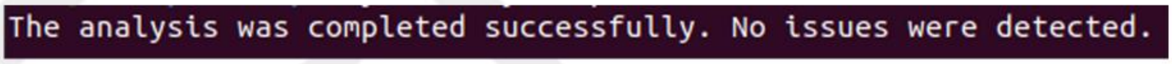


```
browser/open.sol:8:1: Warning:
Experimental features are turned on. Do
not use experimental features on live
deployments. pragma experimental
ABIEncoderV2; ^------------------------
-------^
```

Remix returned a warning about using experimental features which has already been covered in the manual audit.

Mythril

Mythril is a security analysis tool for EVM bytecode. It detects security vulnerabilities in smart contracts built for Ethereum, Hedera, Quorum, Vechain, Roostock, Tron and other EVM-compatible blockchains. It uses symbolic execution, SMT solving and taint analysis to detect a variety of security vulnerabilities. Mythril was used to analyse the contract code using runtime Bytecode of the contract.

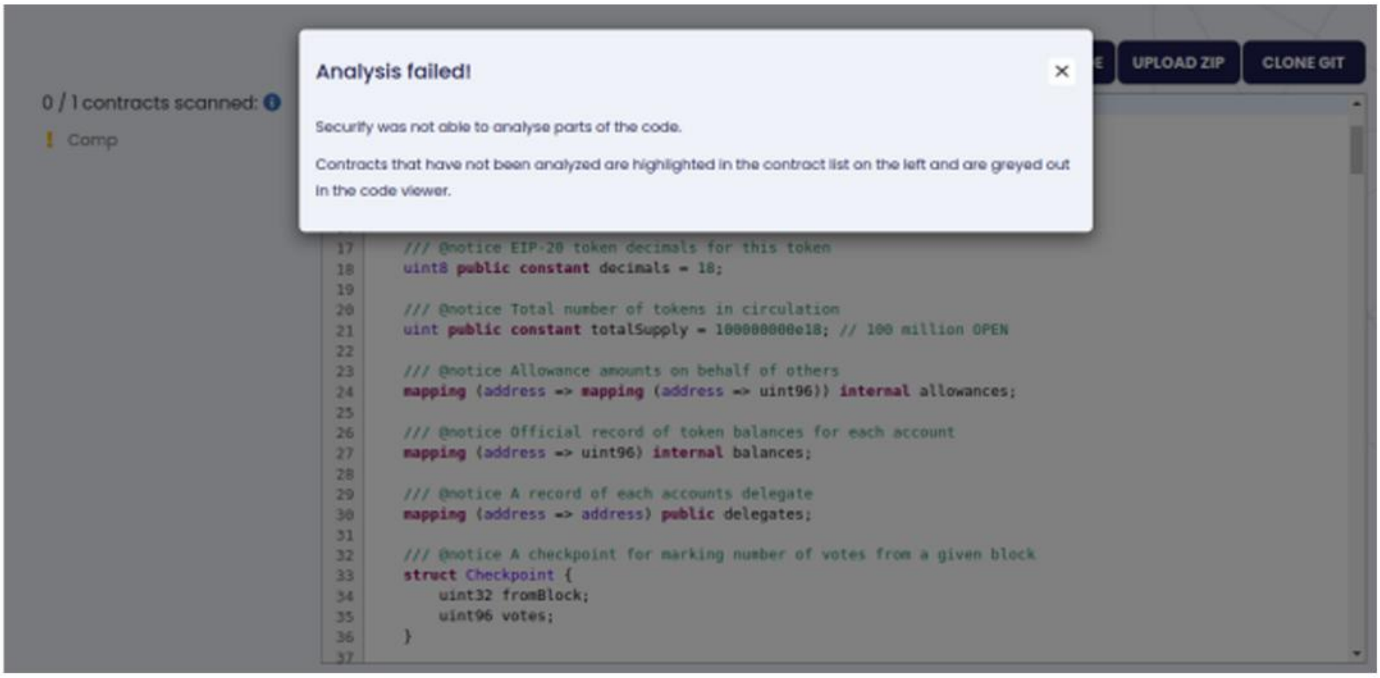


```
The analysis was completed successfully. No issues were detected.
```

Mythril did not return any issues whatsoever.

Securify

Securify is a tool that scans Ethereum smart contracts for critical security vulnerabilities. Securify statically analyzes the EVM code of the smart contract to infer important semantic information (including control-flow and data-flow facts) about the contract. It was unable to Audit the OpenDAO contract due to incompatible compiler versions.



0 / 1 contracts scanned:

Analysis failed!

Securify was not able to analyse parts of the code.

Contracts that have not been analyzed are highlighted in the contract list on the left and are greyed out in the code viewer.

```
    /// @notice EIP-20 token decimals for this token
    uint8 public constant decimals = 18;

    /// @notice Total number of tokens in circulation
    uint public constant totalSupply = 100000000e18; // 100 million OPEN

    /// @notice Allowance amounts on behalf of others
    mapping (address => mapping (address => uint96)) internal allowances;

    /// @notice Official record of token balances for each account
    mapping (address => uint96) internal balances;

    /// @notice A record of each accounts delegate
    mapping (address => address) public delegates;

    /// @notice A checkpoint for marking number of votes from a given block
    struct Checkpoint {
        uint32 fromBlock;
        uint96 votes;
    }
```

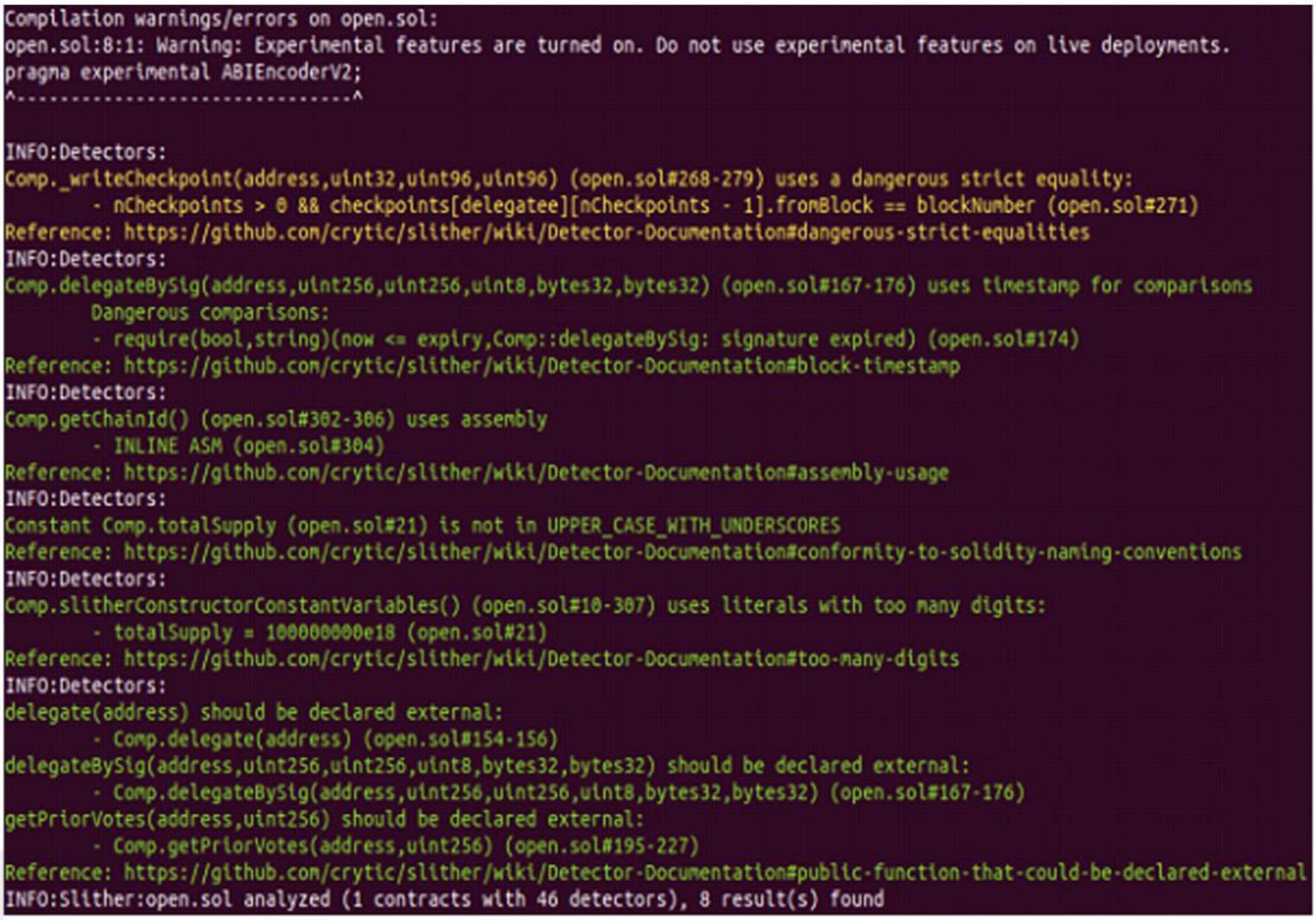
SmartCheck

Smartcheck is a tool for automated static analysis of Solidity source code for security vulnerabilities and best practices. It gave the following result for the OpenDAO contract.
https://tool.smartdec.net/scan/b3db7167713b44afaee473f3ba0126df.

SmartCheck did not detect any high severity issue. All the considerable issues raised by SmartCheck are already covered in the Manual Audit section of this report.

Slither

Slither, an open-source static analysis framework. This tool provides rich information about Ethereum smart contracts and has the critical properties. While Slither is built as a security-oriented static analysis framework, it is also used to enhance the user's understanding of smart contracts, assist in code reviews, and detect missing optimizations.

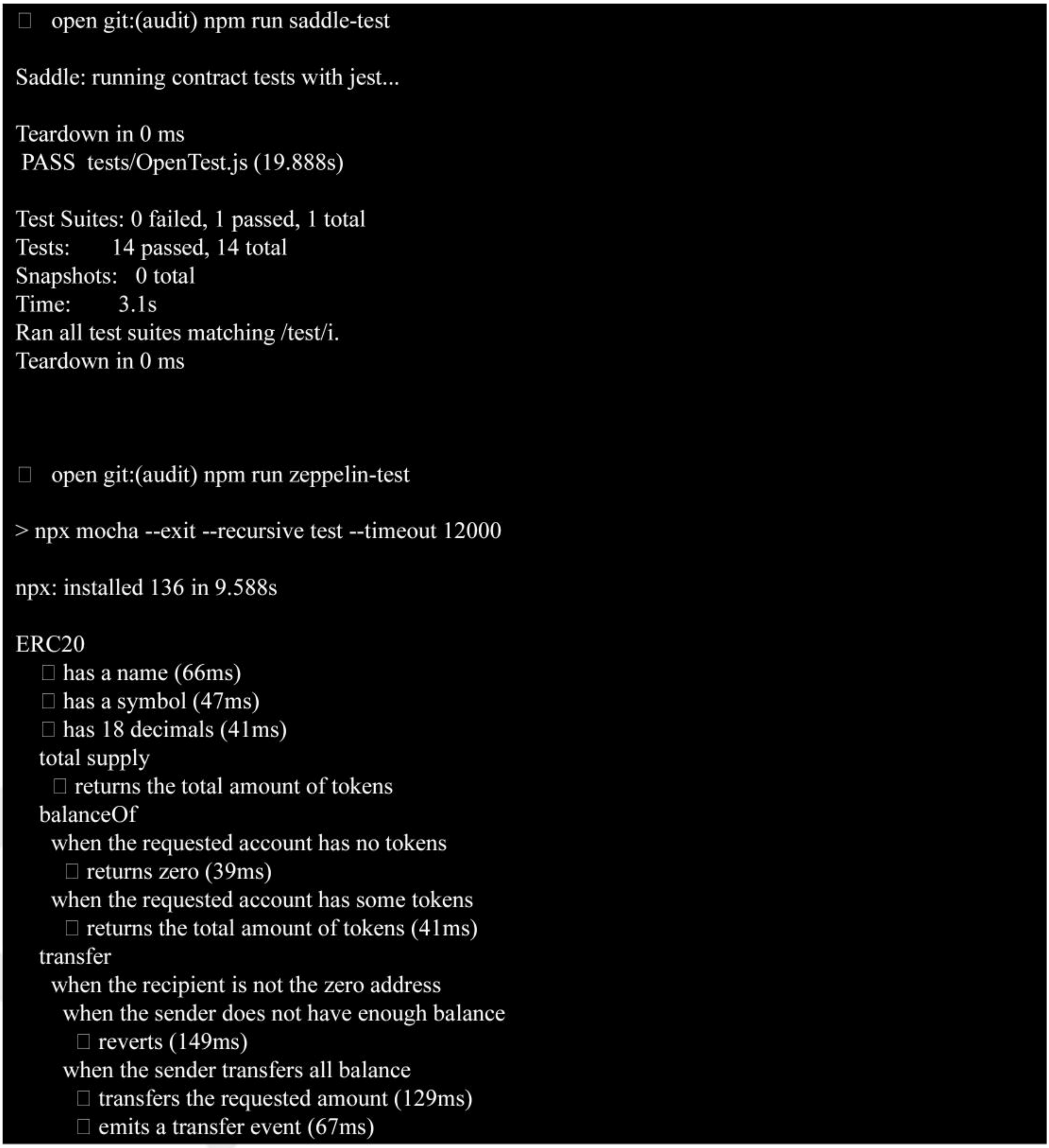


```
Compilation warnings/errors on open.sol:
open.sol:8:1: Warning: Experimental features are turned on. Do not use experimental features on live deployments.
pragma experimental ABIEncoderV2;
^-------------------------------^

INFO:Detectors:
Comp._writeCheckpoint(address,uint32,uint96,uint96) (open.sol#268-279) uses a dangerous strict equality:
        - nCheckpoints > 0 && checkpoints[delegatee][nCheckpoints - 1].fromBlock == blockNumber (open.sol#271)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dangerous-strict-equalities
INFO:Detectors:
Comp.delegateBySig(address,uint256,uint256,uint8,bytes32,bytes32) (open.sol#167-176) uses timestamp for comparisons
        Dangerous comparisons:
        - require(bool,string)(now <= expiry,Comp::delegateBySig: signature expired) (open.sol#174)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#block-timestamp
INFO:Detectors:
Comp.getChainId() (open.sol#302-306) uses assembly
        - INLINE ASM (open.sol#304)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#assembly-usage
INFO:Detectors:
Constant Comp.totalSupply (open.sol#21) is not in UPPER_CASE_WITH_UNDERSCORES
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformity-to-solidity-naming-conventions
INFO:Detectors:
Comp.slitherConstructorConstantVariables() (open.sol#10-307) uses literals with too many digits:
        - totalSupply = 100000000e18 (open.sol#21)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#too-many-digits
INFO:Detectors:
delegate(address) should be declared external:
        - Comp.delegate(address) (open.sol#154-156)
delegateBySig(address,uint256,uint256,uint8,bytes32,bytes32) should be declared external:
        - Comp.delegateBySig(address,uint256,uint256,uint8,bytes32,bytes32) (open.sol#167-176)
getPriorVotes(address,uint256) should be declared external:
        - Comp.getPriorVotes(address,uint256) (open.sol#195-227)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#public-function-that-could-be-declared-external
INFO:Slither:open.sol analyzed (1 contracts with 46 detectors), 8 result(s) found
```

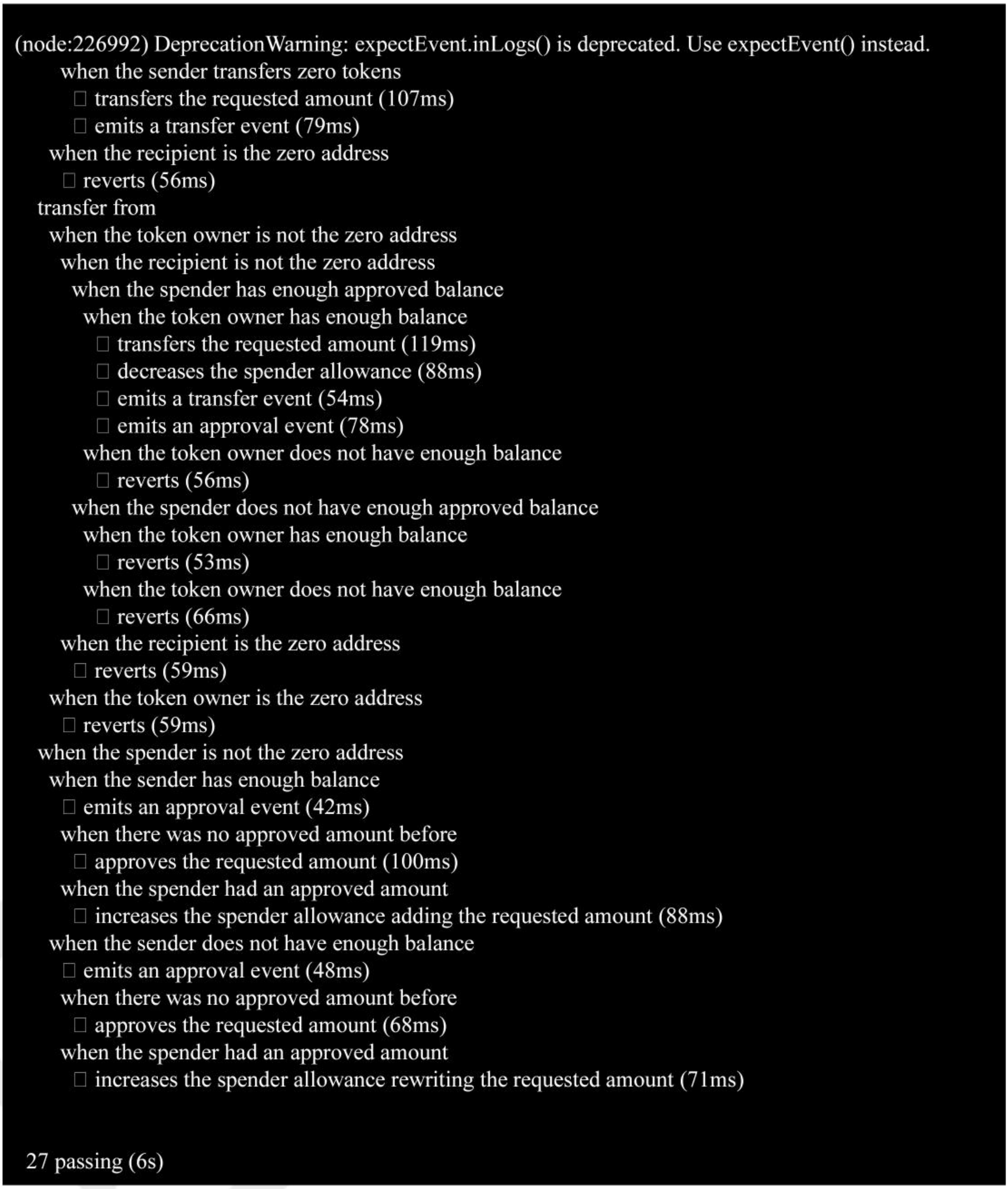
Slither did not detect any high severity issue. All the considerable issues raised by Slither are already covered in the Manual Audit section of this report.

Unit Tests

We checked 14 unit test cases using Saddle and another 27 test cases using OpenZeppelin test environment.



```
□  open git:(audit) npm run saddle-test

Saddle: running contract tests with jest...

Teardown in 0 ms
 PASS  tests/OpenTest.js (19.888s)

Test Suites: 0 failed, 1 passed, 1 total
Tests:       14 passed, 14 total
Snapshots:   0 total
Time:        3.1s
Ran all test suites matching /test/i.
Teardown in 0 ms



□  open git:(audit) npm run zeppelin-test

> npx mocha --exit --recursive test --timeout 12000

npx: installed 136 in 9.588s

ERC20
    □ has a name (66ms)
    □ has a symbol (47ms)
    □ has 18 decimals (41ms)
    total supply
      □ returns the total amount of tokens
    balanceOf
      when the requested account has no tokens
        □ returns zero (39ms)
      when the requested account has some tokens
        □ returns the total amount of tokens (41ms)
    transfer
      when the recipient is not the zero address
        when the sender does not have enough balance
          □ reverts (149ms)
        when the sender transfers all balance
          □ transfers the requested amount (129ms)
          □ emits a transfer event (67ms)
```



```
(node:226992) DeprecationWarning: expectEvent.inLogs() is deprecated. Use expectEvent() instead.
        when the sender transfers zero tokens
          □ transfers the requested amount (107ms)
          □ emits a transfer event (79ms)
      when the recipient is the zero address
        □ reverts (56ms)
    transfer from
      when the token owner is not the zero address
        when the recipient is not the zero address
          when the spender has enough approved balance
            when the token owner has enough balance
              □ transfers the requested amount (119ms)
              □ decreases the spender allowance (88ms)
              □ emits a transfer event (54ms)
              □ emits an approval event (78ms)
            when the token owner does not have enough balance
              □ reverts (56ms)
          when the spender does not have enough approved balance
            when the token owner has enough balance
              □ reverts (53ms)
            when the token owner does not have enough balance
              □ reverts (66ms)
        when the recipient is the zero address
          □ reverts (59ms)
      when the token owner is the zero address
        □ reverts (59ms)
    when the spender is not the zero address
      when the sender has enough balance
        □ emits an approval event (42ms)
        when there was no approved amount before
          □ approves the requested amount (100ms)
        when the spender had an approved amount
          □ increases the spender allowance adding the requested amount (88ms)
      when the sender does not have enough balance
        □ emits an approval event (48ms)
        when there was no approved amount before
          □ approves the requested amount (68ms)
        when the spender had an approved amount
          □ increases the spender allowance rewriting the requested amount (71ms)


  27 passing (6s)
```

Disclaimer

Quillhash audit is not a security warranty, investment advice, or an endorsement of the Open contract. Securing smart contracts is a multistep process, therefore running a bug bounty program as a complement to this audit is strongly recommended.

Summary

Use case of the smart contract is very well designed and implemented. Altogether, the code is written and demonstrates effective use of abstraction, separation of concerns, and modularity. Overall the code is well written and readable with no high and medium level concerns, but can be improved according to Solidity's style guide which is recommended to be fixed before implementing a live version.

OPEN

QuillAudits

India, Singapore, United Kingdom, Canada

audits.quillhash.com

hello@quillhash.com