QuillAudits

Audit Report
November, 2020

OPEN

Contents

Introduction

During the period of October 26th, 2020 to November 1st, 2020 – Quillhash Team performed security audit for OpenDAO Vesting smart contract. The code for audit was taken from following the official github account of OpenDAO:

https://github.com/opendao-protocols/timelock-tokenallocation/blob/master/Vesting/src/contracts/VestingContract.sol

Overview of OpenDAO

OpenDAO enables real world assets such as stocks, bonds and real estate to be used meaningfully in the DeFi ecosystem via permissionless, trust minimized, transparent, secure and automated protocols.
The business logic behind Token Allocation, Sale Details and Token Vesting Schedule can be found in the whitepaper [Page47 - 50]. The whitepaper available on the official website of OPENDAO was used for verify business logic.

https://opendao.io/Whitepaper-OpenDAO.pdf

Scope of Audit

The scope of this audit was to analyze and document OpenDAO vesting contract codebase for quality, security, and correctness.

Check Vulnerabilities

We have scanned OpenDAO Vesting smart contract for commonly known and more specific vulnerabilities. Here are some of the commonly known vulnerabilities that we considered:

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- DoS with (Unexpected) Throw
- DoS with (Unexpected) revert
- DoS with Block Gas Limit
- Transaction-Ordering
- Dependence
- Use of tx.origin
- Exception disorder

- Gasless send
- Balance equality
- Byte array
- Transfer forwards all gas
- ERC20 API violation
- Malicious libraries
- Compiler version not fixed
- Redundant fallback function
- Send instead of transfer
- Style guide violation
- Unchecked external call
- Unchecked math
- Unsafe type inference
- Implicit visibility level
- Address hardcoded
- Using delete for arrays
- Integer overflow/underflow
- Locked money
- Private modifier
- Revert/require functions
- Using var
- Visibility
- Using blockhash
- Using SHA3
- Using suicide
- Using throw
- Using inline assembly

Techniques and Methods

Throughout the audit of smart contracts care was taken to ensure:

- Overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behavior.
- Token distribution and calculations are as per intended behavior mentioned in whitepaper.
- Implementation of ERC-20 token standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods and tools were used to review all the smart contracts.

Structural Analysis

In this step we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.
SmartCheck.

Static Analysis

Static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step a series of automated tools are used to test security of smart contracts.

Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerability or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of automated analysis were manually verified.

Gas Consumption

In this step we have checked the behaviour of smart contracts in production. Checks were done to know how much gas gets consumed and possibilities of optimization of code to reduce gas consumption.

Tools and Platforms used for Audit

Remix IDE, Truffle, Truffle Team, Ganache, Solhint, Mythril, Slither, SmartCheck.

Issue Categories

Every issue in this report has been assigned with a severity level. There are four levels of severity and each of them has been explained below.

High severity issues

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality and we recommend these issues to be fixed before moving to a live environment.

Medium level severity issues

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems and they should still be fixed.

Low level severity issues

Low level severity issues can cause minor impact and or are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

Informational

These are severity four issues which indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Number of issues per severity

	High	Medium	Low	Informational
Open	0	0	8	5
Closed	0	0	0	0

Issues Found – Code Review / Manual Testing

High severity issues

None.

Medium severity issues

None.

Low level severity issues

1. **Compiler version should be fixed**
   Solidity source files indicate the versions of the compiler they can be compiled with. It’s recommended to lock the compiler version in code, as future compiler versions may handle certain language constructions in a way the developer did not foresee. Also, it's recommended to use the latest compiler version.

2. **Coding Style Issues**
   Coding style issues influence code readability and in some cases may lead to bugs in future. Smart Contracts have naming convention, indentation and code layout issues. It's recommended to use the Solidity Style Guide to fix all the issues. Consider following the Solidity guidelines on formatting the code and commenting for all the files. It can improve the overall code quality and readability.

3. **Order of layout**
   The order of functions as well as the rest of the code layout does not follow solidity style guide. Layout contract elements in the following order:
   - Pragma statements
   - Import statements
   - Interfaces
   - Libraries
   - Contracts

   Inside each contract, library or interface, use the following order:
   - Type declarations
   - State variables
   - Events
   - Functions

   Please read following documentation links to understand the correct order:

   - https://solidity.readthedocs.io/en/v0.5.3/style-guide.html#order-of-layout

   - https://solidity.readthedocs.io/en/v0.5.3/style-guide.html#order-of-functions

4. **Give preference for 'bytes32' over 'string'**
   For constant values smaller than 32 bytes, give preference for a bytes32 type, since they are much cheaper than string types, which are dynamically sized.

5. **Pure functions should not modify the state[#527-531]**
   Using inline assembly that contains certain opcodes is considered as modifying the state. Functions that modify the state should not be declared as pure functions. Do not declare functions that change the state as pure.

6. **Use external function modifier instead of public [# 379-381, 392-401, 420-452]**
   The public functions that are never called by contract should be declared external to save gas.

7. **Consider using struct instead of multiple return values.**
   Replace multiple return values with a struct. It helps improve readability.

8. **Structs should be named using the CapWords style**
   Struct Checkpoint [#258-261] is not in CapWords.

Informational

These are severity four issues which indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

1. **Delete variables that you don’t need**
   In Ethereum, you get a gas refund for freeing up storage space. If you don’t need a variable anymore, you should delete it using the delete keyword provided by solidity or by setting it to its default value.

2. **Do not use EVM assembly. Use of assembly is error-prone and should be avoided.**

3. **Use local variable instead of state variable like .length in a loop [#608-612]**
   For every iteration of for loop - state variables like .length of non-memory array will consume extra gas. To reduce the gas consumption it’s advisable to use local variable.

4. **Be explicit about which `uint` the code is using**
   `uint` is an alias for `uint256`, but using the full form is preferable. Be consistent and use one of the forms.

5. **Use of block.timestamp should be avoided**
   It is risky to use block.timestamp. As block.timestamp can be manipulated by miners. Therefore block.timestamp is recommended to be supplemented with some other strategy in the case of high-value/risk applications.

Automated Testing

Slither

Slither is an open-source Solidity static analysis framework. This tool provides rich information about Ethereum smart contracts and has the critical properties. It runs a suite of vulnerability detectors, prints visual information about contract details, and provides an API to easily write custom analyses. Slither enables developers to find vulnerabilities, enhance their code comprehension, and quickly prototype custom analyses.

- Detects vulnerable Solidity code with low false positives
- Identifies where the error condition occurs in the source code
- Easily integrates into continuous integration and Truffle builds
- Built-in 'printers' quickly report crucial contract information
- Detector API to write custom analyses in Python
- Ability to analyze contracts written with Solidity >= 0.4
- Intermediate representation (SlithIR) enables simple, high-precision analysis
- Correctly parses 99.9% of all public Solidity code
- Average execution time of less than 1 second per contract

```
INFO:Detectors:
VestingContractWithoutDelegation._availableDrawDownAmount(address) (VestingContract.sol#748-774) performs a multiplication on the result of a division:
        -drawDownRate = vestedAmount[_beneficiary].div(end.sub(start)) (VestingContract.sol#770)
        -amount = timePassedSinceLastInvocation.mul(drawDownRate) (VestingContract.sol#771)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#divide-before-multiply
INFO:Detectors:
Comp._writeCheckpoint(address,uint32,uint96,uint96) (VestingContract.sol#493-504) uses a dangerous strict equality:
        - nCheckpoints > 0 && checkpoints[delegatee][nCheckpoints - 1].fromBlock == blockNumber (VestingContract.sol#496)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dangerous-strict-equalities
INFO:Detectors:
Reentrancy in VestingContractWithoutDelegation._createVestingSchedule(address,uint256) (VestingContract.sol#698-716):
        External calls:
        - require(bool,string)(token.transferFrom(msg.sender,address(this),_amount),VestingContract::createVestingSchedule: Unable to escrow tokens) (VestingContract.so
l#708-711)
        Event emitted after the call(s):
        - ScheduleCreated(_beneficiary) (VestingContract.sol#713)
Reentrancy in VestingContractWithoutDelegation._drawDown(address) (VestingContract.sol#718-742):
        External calls:
        - require(bool,string)(token.transfer(_beneficiary,amount),VestingContract::_drawDown: Unable to transfer tokens) (VestingContract.sol#737)
        Event emitted after the call(s):
        - DrawDown(_beneficiary,amount) (VestingContract.sol#739)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-3
INFO:Detectors:
Comp.delegateBySig(address,uint256,uint256,uint8,bytes32,bytes32) (VestingContract.sol#392-401) uses timestamp for comparisons
        Dangerous comparisons:
        - require(bool,string)(now <= expiry,Comp::delegateBySig: signature expired) (VestingContract.sol#399)
VestingContractWithoutDelegation._drawDown(address) (VestingContract.sol#718-742) uses timestamp for comparisons
        Dangerous comparisons:
        - require(bool,string)(amount > 0,VestingContract::_drawDown: No allowance left to withdraw) (VestingContract.sol#722)
        - require(bool,string)(totalDrawn[_beneficiary] <= vestedAmount[_beneficiary],VestingContract::_drawDown: Safety Mechanism - Drawn exceeded Amount Vested) (Vest
ingContract.sol#731-734)
        - require(bool,string)(token.transfer(_beneficiary,amount),VestingContract::_drawDown: Unable to transfer tokens) (VestingContract.sol#737)
VestingContractWithoutDelegation._availableDrawDownAmount(address) (VestingContract.sol#748-774) uses timestamp for comparisons
        Dangerous comparisons:
        - _getNow() <= start.add(cliffDuration) (VestingContract.sol#751)
        - _getNow() > end (VestingContract.sol#757)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#block-timestamp
```

```
INFO:Detectors:
Comp.getChainId() (VestingContract.sol#527-531) uses assembly
        - INLINE ASM (VestingContract.sol#529)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#assembly-usage
INFO:Detectors:
Constant Comp.totalSupply (VestingContract.sol#246) is not in UPPER_CASE_WITH_UNDERSCORES
Parameter VestingContractWithoutDelegation.createVestingSchedules(address[],uint256[])._beneficiaries (VestingContract.sol#596) is not in mixedCase
Parameter VestingContractWithoutDelegation.createVestingSchedules(address[],uint256[])._amounts (VestingContract.sol#597) is not in mixedCase
Parameter VestingContractWithoutDelegation.createVestingSchedule(address,uint256)._beneficiary (VestingContract.sol#623) is not in mixedCase
Parameter VestingContractWithoutDelegation.createVestingSchedule(address,uint256)._amount (VestingContract.sol#623) is not in mixedCase
Parameter VestingContractWithoutDelegation.transferOwnership(address)._newOwner (VestingContract.sol#634) is not in mixedCase
Parameter VestingContractWithoutDelegation.vestingScheduleForBeneficiary(address)._beneficiary (VestingContract.sol#667) is not in mixedCase
Parameter VestingContractWithoutDelegation.availableDrawDownAmount(address)._beneficiary (VestingContract.sol#683) is not in mixedCase
Parameter VestingContractWithoutDelegation.remainingBalance(address)._beneficiary (VestingContract.sol#692) is not in mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformity-to-solidity-naming-conventions
INFO:Detectors:
Comp.slitherConstructorConstantVariables() (VestingContract.sol#235-532) uses literals with too many digits:
        - totalSupply = 100000000e18 (VestingContract.sol#246)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#too-many-digits
INFO:Detectors:
delegate(address) should be declared external:
        - Comp.delegate(address) (VestingContract.sol#379-381)
delegateBySig(address,uint256,uint256,uint8,bytes32,bytes32) should be declared external:
        - Comp.delegateBySig(address,uint256,uint256,uint8,bytes32,bytes32) (VestingContract.sol#392-401)
getPriorVotes(address,uint256) should be declared external:
        - Comp.getPriorVotes(address,uint256) (VestingContract.sol#420-452)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#public-function-that-could-be-declared-external
INFO:Slither:VestingContract.sol analyzed (5 contracts with 46 detectors), 21 result(s) found
INFO:Slither:Use https://crytic.io/ to get access to additional detectors and Github integration
```

Slither didn't raise any critical issue with the OpenDAO Vesting contract. The contract was well tested and all the minor issues that were raised have been documented in the report. All other vulnerabilities of importance have already been covered in the manual audit section of the report.

SmartCheck

SmartCheck is a tool for automated static analysis of Solidity source code for security vulnerabilities and best practices. SmartCheck translates Solidity source code into an XML-based intermediate representation and checks i t against XPath patterns. SmartCheck shows significant improvements over existing alternatives in terms of false discovery rate (FDR) and false negative rate (FNR). It gave the following result for the OpenDAO Vesting contract:

https://tool.smartdec.net/scan/49c2774715f84e8f97d797c463d7e8c0

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

Mythril

Mythril is a security analysis tool for EVM bytecode. It detects security vulnerabilities in smart contracts built for Ethereum. It uses symbolic execution, SMT solving and taint analysis to detect a variety of security vulnerabilities.

Mythril did not detect any high severity issue. All the considerable issues raised by Mythril are already covered in the code review section of this report.

Remix IDE

Remix is a powerful, open source tool that helps you write Solidity contracts straight from the browser. Written in JavaScript, Remix supports both usage in the browser and locally.

The OpenDAO Vesting smart contract was tested for various compiler versions. The smart contract compiled without any error on explicitly setting compiler version 0.5.17 to 0.6.7. The contract fails to compile when the compiler version is set to 0.6.8 or higher.

SOLIDITY COMPILER

COMPILER

0.6.8+commit.0bbfe453

Include nightly builds

LANGUAGE

Solidity

EVM VERSION

compiler default

COMPILER CONFIGURATION

Auto compile

Enable optimization

Hide warnings

Compile VestingContract.sol

CONTRACT

Comp (VestingContract.sol)

Publish on Swarm

Publish on Ipfs

Compilation Details

Home VestingContract.sol

```
1  pragma solidity 0.6.8;
2
3  library SafeMath {
4      /**
5       * @dev Returns the addition of two unsigned integers, reverting on
6       * overflow.
7       *
8       * Counterpart to Solidity's `+` operator.
9       *
10      * Requirements:
11      * - Addition cannot overflow.
12      */
13     function add(uint256 a, uint256 b) internal pure returns (uint256) {
14         uint256 c = a + b;
15         require(c >= a, "SafeMath: addition overflow");
16
17         return c;
18     }
19
20     /**
21      * @dev Returns the subtraction of two unsigned integers, reverting on
22      * overflow (when the result is negative).
23      *
24      * Counterpart to Solidity's `-` operator.
25      *
26      * Requirements:
27      * - Subtraction cannot overflow.
28      */
29     function sub(uint256 a, uint256 b) internal pure returns (uint256) {
30         return sub(a, b, "SafeMath: subtraction overflow");
31     }
32
33     /**
34      * @dev Returns the subtraction of two unsigned integers, reverting with custom message on
35      * overflow (when the result is negative).
36      *
37      * Counterpart to Solidity's `-` operator.
38      *
39      * Requirements:
40      * - Subtraction cannot overflow.
```

SOLIDITY COMPILER

COMPILER

0.5.17+commit.d19bba13

Include nightly builds

LANGUAGE

Solidity

EVM VERSION

compiler default

COMPILER CONFIGURATION

Auto compile

Enable optimization

Hide warnings

Compile VestingContract.sol

CONTRACT

Comp (VestingContract.sol)

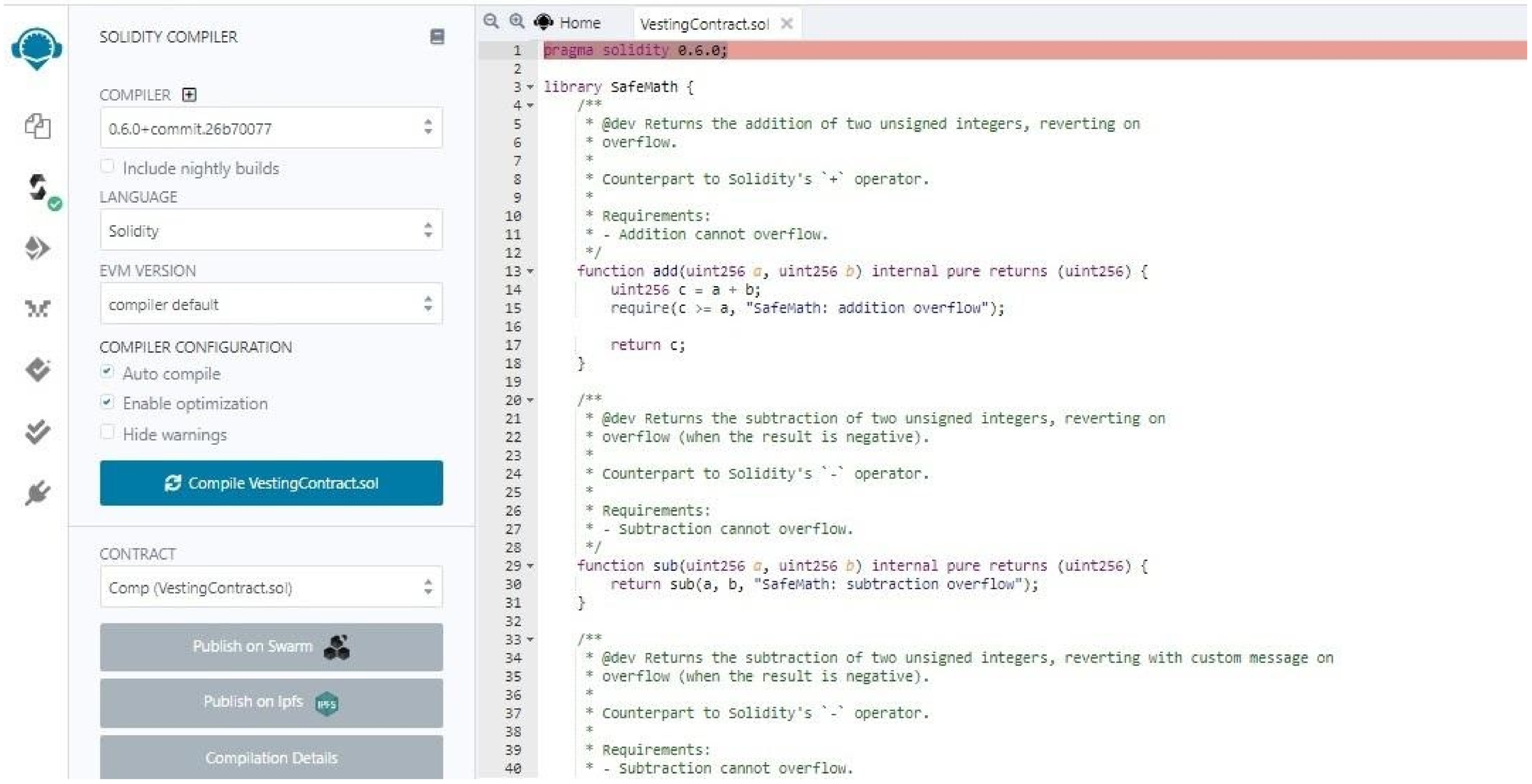
Publish on Swarm

Publish on Ipfs

Compilation Details

Home VestingContract.sol

```
1  pragma solidity ^0.5.16;
2
3  library SafeMath {
4      /**
5       * @dev Returns the addition of two unsigned integers, reverting on
6       * overflow.
7       *
8       * Counterpart to Solidity's `+` operator.
9       *
10      * Requirements:
11      * - Addition cannot overflow.
12      */
13     function add(uint256 a, uint256 b) internal pure returns (uint256) {
14         uint256 c = a + b;
15         require(c >= a, "SafeMath: addition overflow");
16
17         return c;
18     }
19
20     /**
21      * @dev Returns the subtraction of two unsigned integers, reverting on
22      * overflow (when the result is negative).
23      *
24      * Counterpart to Solidity's `-` operator.
25      *
26      * Requirements:
27      * - Subtraction cannot overflow.
28      */
29     function sub(uint256 a, uint256 b) internal pure returns (uint256) {
30         return sub(a, b, "SafeMath: subtraction overflow");
31     }
32
33     /**
34      * @dev Returns the subtraction of two unsigned integers, reverting with custom message on
35      * overflow (when the result is negative).
36      *
37      * Counterpart to Solidity's `-` operator.
38      *
39      * Requirements:
40      * - Subtraction cannot overflow.
```



It is recommended to use any fixed compiler versions from 0.5.17 to 0.6.7.
It threw some warnings and these have been reported in the code review or manual testing section of this report. There was no critical or major issue reported by Remix.

Closing Summary

Overall, the smart contracts are very well written and adhere to ERC-20 guidelines. Several issues of low severity were found during the audit. There were no critical or major issues found that can break the intended behaviour.

Disclaimer

Quillhash audit is not a security warranty, investment advice, or an endorsement of the OpenDAO platform. This audit does not provide a security or correctness guarantee of the audited smart contracts. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them. Securing smart contracts is a multistep process. One audit cannot be considered enough. We recommend that the OpenDAO Team put in place a bug bounty program to encourage further analysis of the smart contract by other third parties.

QuillAudits

Canada, India, Singapore and United Kingdom

audits.quillhash.com

hello@quillhash.com