Rubicon Smart Contract Audit Report

Introduction

During the period of **October 8th, 2020 to October 26th, 2020** - Kavita Bhatia performed security audit for Rubicon smart contracts. The code for audit was taken from following the official smart contract link:

https://github.com/RubiconDeFi/rubicon_protocol/tree/master/contracts

The contracts audited –

i) RBCN.sol
ii) RubiconMarket.sol
iii) SenateAlpha.sol
iv) Timelock.sol.

Repository: commit 498c9ea8cb2bef77753f1243980cd1c445305363

Overview of RUBICON

In an attempt to advance, democratize, and unlock the future of global equity markets, the Rubicon Protocol is being launched. Importantly, this protocol will improve or eliminate many of the aforementioned barriers, costs, and difficulties associated with the equity markets of today. Additionally, this protocol will embrace the ethos and requisite characteristics of the new wave of decentralized finance by being open-sourced, community governed, permissionless, and free to all.

The contracts will contain a scalable open order book exchange for equities, governance token that enables participation on a permissionless basis, clear rules and guidelines for the on boarding and trading of existing equities, and visible data for all modern-era regulatory agencies and market participants to validate and monitor.

Main Features:

- RBCN will adopt the **ERC-20 token** standard while also enabling governance participation.
- Rubicon Protocol Token will be abbreviated to **RBCN**
- Token will be **non-fungible**
- Fixed supply of **1,000,000,000** tokens will be available for ownership
- 51% will be provided for free to market participants over time.
- Rubicon Protocol to function as a **Decentralized Autonomous Organization (DAO)**
- All market participants are required to adhere to local, national, and international regulations and compliance obligations themselves.
- Plays no role in the regulation or compliance activities
- It will be open to all token standards that enable compliant security token exchange.
- It will support the transaction of **ERC-1404 assets**
- It will act as a middle layer that will enable any digital asset or equity to freely trade
- It will use open order book model of exchange and will enable the usage of “limit” buy and sell orders while enabling market makers and equity holders to effectively manage risk and market exposure.

Scope of Audit

The scope of this audit was to analyze and document Rubicon contracts codebase for quality, security, and correctness.

Checked Vulnerabilities

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- 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

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 Smart contract is structured in a way that will not result in future problems.

Static Analysis

Static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step 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 ma nually verified.

Gas Consumption

In this step we have checked the behaviour of smart contract 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 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 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

TYPE	HIGH	MEDIUM	LOW	INFORMATIONAL
OPEN	0	3	5	6
CLOSED	0	0	0	0

Issues Found – Code Review / Manual Testing

High Severity Issues

None.

Medium Severity Issues

1. **Use of deprecated global variable now**
   The global variable now is deprecated from solidity version 0.7 onwards.
   Files: RBCN.sol [#173], RubiconMarket.sol [#324, #357, #402, #417, #473]

2. **Using pragma experimental ABIEncoderV2**
   The new ABI encoder is able to encode and decode arbitrarily nested arrays and structs. It produces less optimal code (the optimizer for this part of the code is still under development) and has not received as much testing as the old encoder. Ethereum through bug bounty program received a report about a flaw within the new experimental ABI encoder (referred to as ABIEncoderV2). Upon investigation, they found that the component suffers from a few different variations of the same type. The bug, when triggered, will under certain circumstances send corrupt parameters on method invocations to other contracts. More details can be found.
   Files: RBCN.sol, SenateAlpha.sol
   Details:
   https://blog.ethereum.org/2019/03/26/solidity-optimizer-and-abiencoderv2-bug/

3. **Re-entrancy**
   One of the major dangers of calling external contracts is that they can take over the control flow, and make changes to your data that the calling function wasn't expecting. This class of bug can take many forms, and both of the major bugs that led to the DAO's collapse were bugs of this sort. It's recommended that the calls to external functions/events should happen after any changes to state variables in your contract so your contract is not vulnerable to a re-entrancy exploit.
   https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/ReentrancyGuard.sol

   Files: RubiconMarket.sol

   a. **_buys(uint256,uint256) [RubiconMarket.sol#878-898]:**
   - State variables written after the call(s):
     i. dustId = id (RubiconMarket.sol#894)
     ii. cancel(id) (RubiconMarket.sol#895)
   - Event emitted after the call(s):
     i. LogItemUpdate(id) (RubiconMarket.sol#348)
     ii. LogKill(bytes32(id),keccak256(bytes)(abi.encodePacked(offer.pay_gem,offer.buy_gem)),offer.owner,offer.pay_gem,offer.buy_gem,uint128(offer.pay_amt),uint128(offer.buy_amt),uint64(now)) (RubiconMarket.sol#349-358)

   b. _matcho(uint256,ERC20,uint256,ERC20,uint256,bool) [RubiconMarket.sol#978-1029]:
   - State variables written after the call(s):

i. _sort(id,pos) (RubiconMarket.sol#1027)
ii. _best[address(pay_gem)][address(buy_gem)] = id (RubiconMarket.sol#1078)

- Event emitted after the call(s):
  i. LogSortedOffer(id) (RubiconMarket.sol#1089)
  ii. _sort(id,pos) (RubiconMarket.sol#1027)

c. SimpleMarket.buy(uint256,uint256) [RubiconMarket.sol#290-333]:

- State variables written after the call(s):
  i. delete offers[id] (RubiconMarket.sol#329)
- Event emitted after the call(s):
  i. LogItemUpdate(id) (RubiconMarket.sol#314)
  ii. LogTake(bytes32(id),keccak256(bytes)(abi.encodePacked(offer.pay_gem,offer.buy_gem)),offer.owner,offer.pay_gem,offer.buy_gem,msg.sender,uint128(quantity),uint128(spend),uint64(now)) (RubiconMarket.sol#315-325)
  iii. LogTrade(quantity,address(offer.pay_gem),spend,address(offer.buy_gem)) (RubiconMarket.sol#326)

d. _offeru(uint256,ERC20,uint256,ERC20) [RubiconMarket.sol#1035-1049]:

- State variables written after the call(s):
  i. _head = id (RubiconMarket.sol#1047)
  ii. _near[id] = _head (RubiconMarket.sol#1046)
- Event emitted after the call(s):
  i. LogUnsortedOffer(id) (RubiconMarket.sol#1048)

e. SimpleMarket.cancel(uint256) [RubiconMarket.sol#336-361]:

- Event emitted after the call(s):
  i. LogItemUpdate(id) (RubiconMarket.sol#348)
  ii. LogKill(bytes32(id),keccak256(bytes)(abi.encodePacked(offer.pay_gem,offer.buy_gem)),offer.owner,offer.pay_gem,offer.buy_gem,uint128(offer.pay_amt),uint128(offer.buy_amt),uint64(now)) (RubiconMarket.sol#349-358)

f. SimpleMarket.offer(uint256,ERC20,uint256,ERC20)[RubiconMarket.sol#382-419]:

- Event emitted after the call(s):
  i. LogItemUpdate(id) (RubiconMarket.sol#408)
  ii. LogMake(bytes32(id),keccak256(bytes)(abi.encodePacked(pay_gem,buy_gem)),msg.sender,pay_gem,buy_gem,uint128(pay_amt),uint128(buy_amt),uint64(now)) (RubiconMarket.sol#409-418)

Low Severity Issues

1. **Pure functions should not read or modify the state**
   Using an inline assembly is considered reading from the state. Functions that read or modify the state should not be declared as pure functions.
   Files: SenateAlpha.sol [#317-321], RBCN.sol [#300-304]

2. **Use external instead of public visibility level for functions**
   The public functions that are never called by contract should be declared external to save gas.
   Files: List of functions that can be declared external:

- delegate(address) [RBCN.sol#153-155]
- delegateBySig(address,uint256,uint256,uint8,bytes32,bytes32)[RBCN.sol#166-175]
- getPriorVotes(address,uint256)[RBCN.sol#194-226]
- propose(address[],uint256[],string[],bytes[],string)[SenateAlpha.sol#142-180]
- queue(uint256)[SenateAlpha.sol#182-191]
- execute(uint256)[SenateAlpha.sol#198-206]
- cancel(uint256)[SenateAlpha.sol#208-221]
- getActions(uint256)[SenateAlpha.sol#223-226]
- getReceipt(uint256,address)[SenateAlpha.sol#228-230]
- castVote(uint256,bool)[SenateAlpha.sol#254-256]
- castVoteBySig(uint256,bool,uint8,bytes32,bytes32) [SenateAlpha.sol#258-265]
- __acceptAdmin() [SenateAlpha.sol#287-290]
- __abdicate() [SenateAlpha.sol#292-295]
- __queueSetTimelockPendingAdmin(address,uint256) [SenateAlpha.sol#297-300]
- __executeSetTimelockPendingAdmin(address,uint256) [SenateAlpha.sol#302-305]
- canCall(address,address,bytes4) [RubiconMarket.sol#24-26]
- totalSupply() [RubiconMarket.sol#154]
- balanceOf(address)[RubiconMarket.sol#155]
- allowance(address,address) [RubiconMarket.sol#156]
- approve(address,uint256) [RubiconMarket.sol#158]
- transfer(address,uint256) [RubiconMarket.sol#159]

transferFrom(address,address,uint256) [RubiconMarket.sol#160-162]

3. **Boolean equality check**
   Boolean constants can be used directly. No need to be compare to true or false.
   Files: SenateAlpha.sol [#271]

4. **Gas Consumption**
   State variables like .balance, or .length of non-memory array used for or while loop consume extra gas for the every iteration of loop. It's is recommended to store the value of state variables like .length or balance in a local variable to make the code more gas efficient.
   Files: RubiconMarket.sol[#995-1021], SenateAlpha.sol[#185-187, #201-203, #215-217]

5. **Costly loops**
   Loops that do not have a fixed number of iterations, for example, loops that depend on storage values, have to be used carefully: Due to the block gas limit, transactions can only consume a certain amount of gas. Either explicitly or just due to normal operation, the number of iterations in a loop can grow beyond the block gas limit which can cause the complete contract to be stalled at a certain point. Avoid loops with big or unknown number of steps.
   Files: RubiconMarket.sol[#914-917, #1140, #954-956, #930-932, #865, #852, #946-949], RBCN.sol[#213]

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. **Gas Optimization tips for all contracts**
   Link below contains some tips that can be followed to optimize gas utilized.
   https://blog.polymath.network/solidity-tips-and-tricks-to-save-gas-and-reduce-bytecode-size-c44580b218e6

2. **Implicit Visibility**
   It's a good practice to explicitly define visibility of state variables, functions, interface functions and fallback functions. The default visibility of state variables – internal; function – public; interface function - external.
   Files: RubiconMarket.sol [#104, #105, #222, #532]

3. **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 latest compiler version.

4. **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 **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.

5. **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:
   a. Pragma statements
   b. Import statements
   c. Interfaces
   d. Libraries
   e. Contracts
   Inside each contract, library or interface, use the following order:
   1. Type declarations
   2. State variables
   3. Events
   4. 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

6. **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.

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 analyses
- Correctly parses 99.9% of all public Solidity code
- Average execution time of less than 1 second per contract

Slither failed to compile RUBICON Protocol contracts with pragma experimental ABIEncoderV2.

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 RUBICON contracts:

https://tool.smartdec.net/scan/17a71882def8449eadfdedc3a3a1a8e1

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

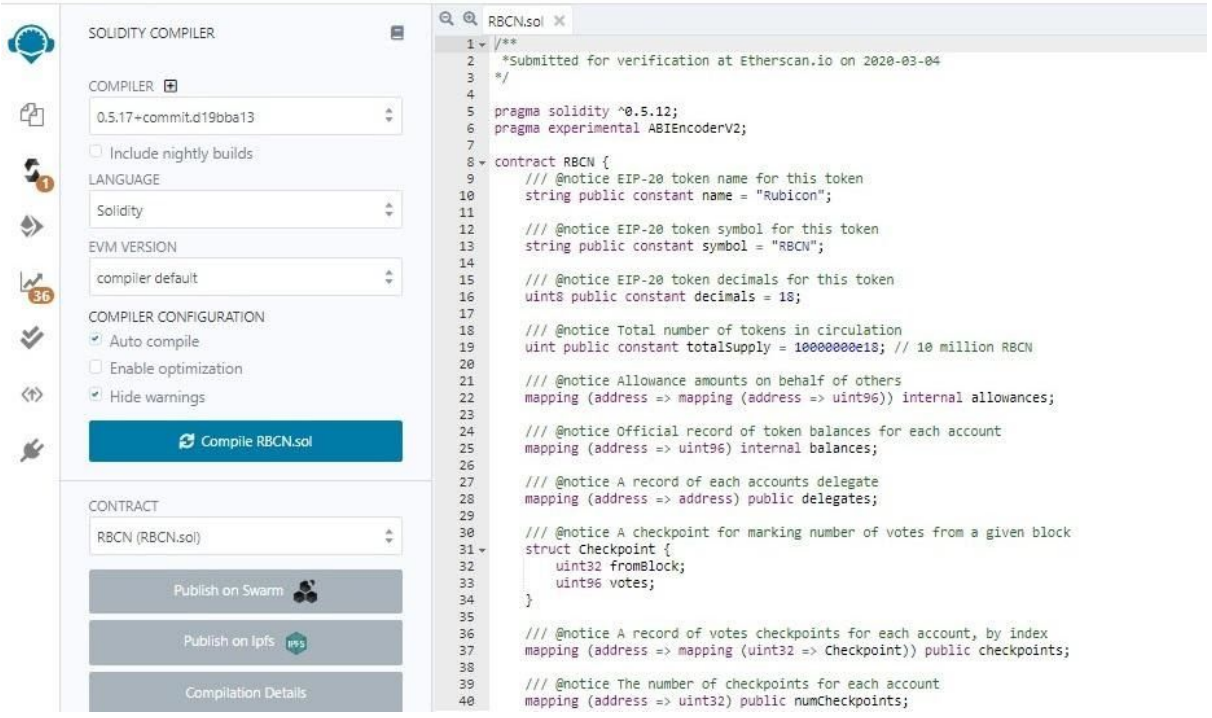
Mythril is a security analysis tool for EVM bytecode. It detects security vulnerabilities in smart contracts built for Ethereum. It uses symbolic execution, SMT solving and taint analysis to detect a variety of security vulnerabilities.

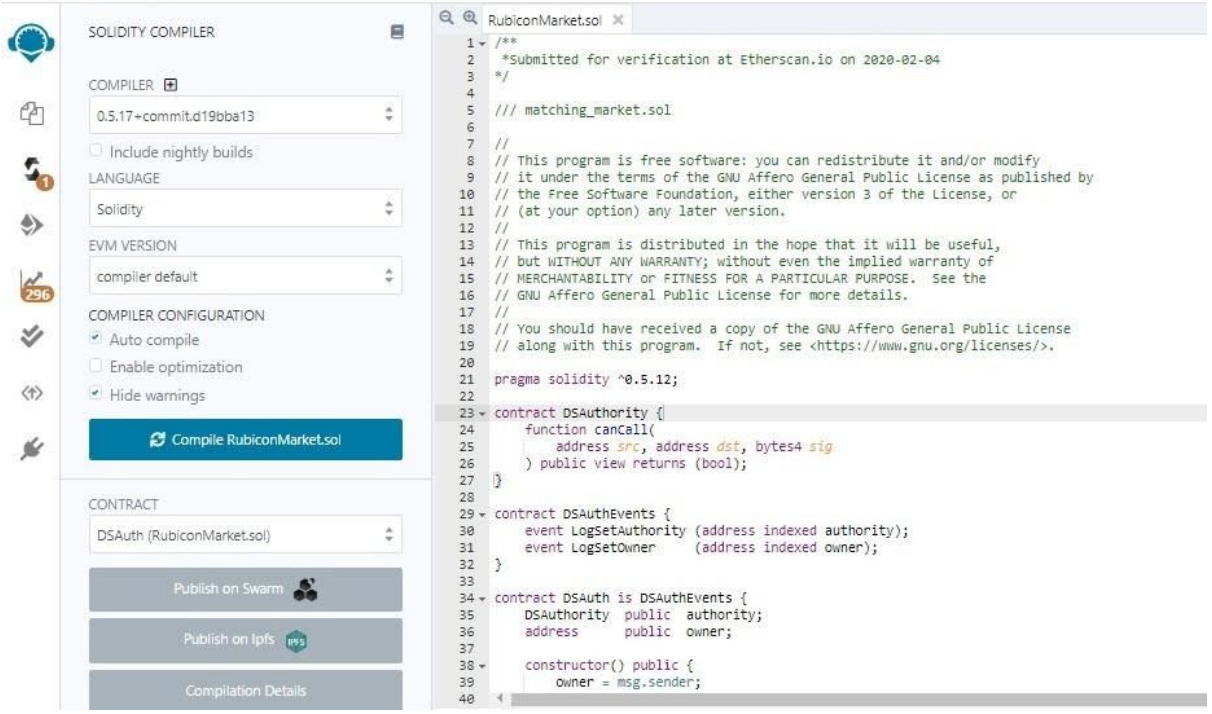
Mythril did not detect any high severity issue. All the considerable issues raised by Mythril are already covered in the code review section of this report.

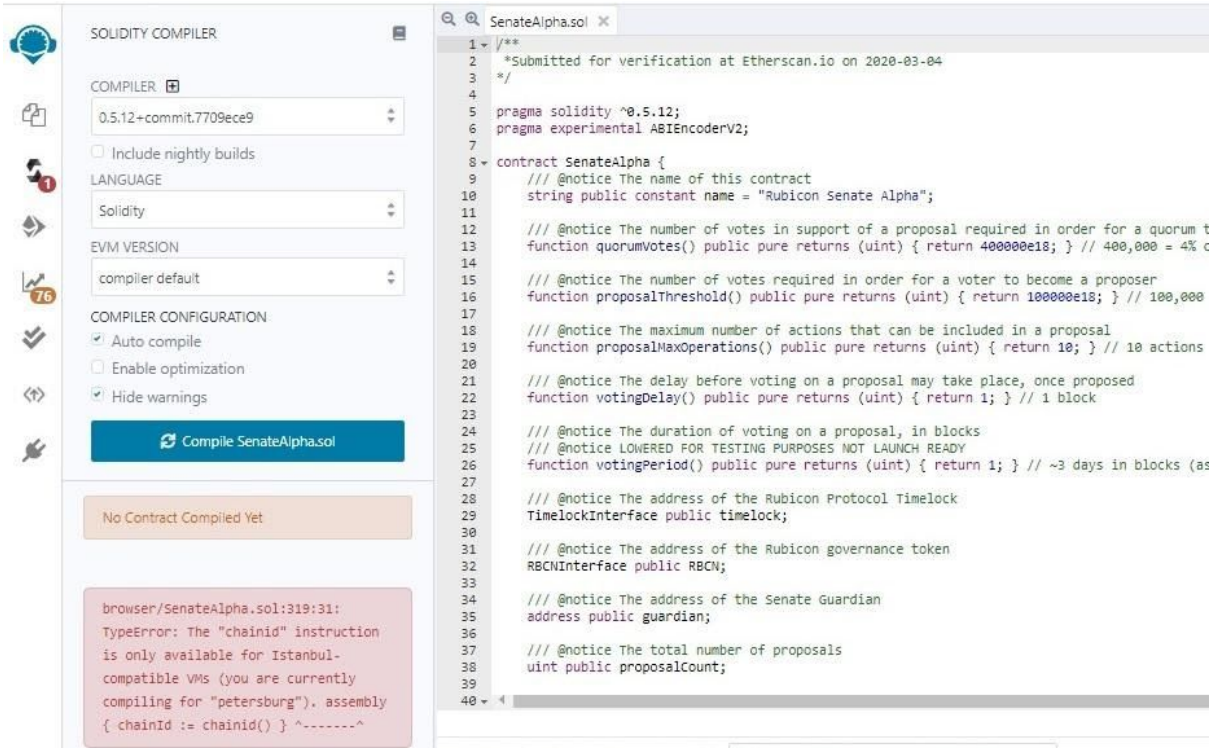
Remix IDE

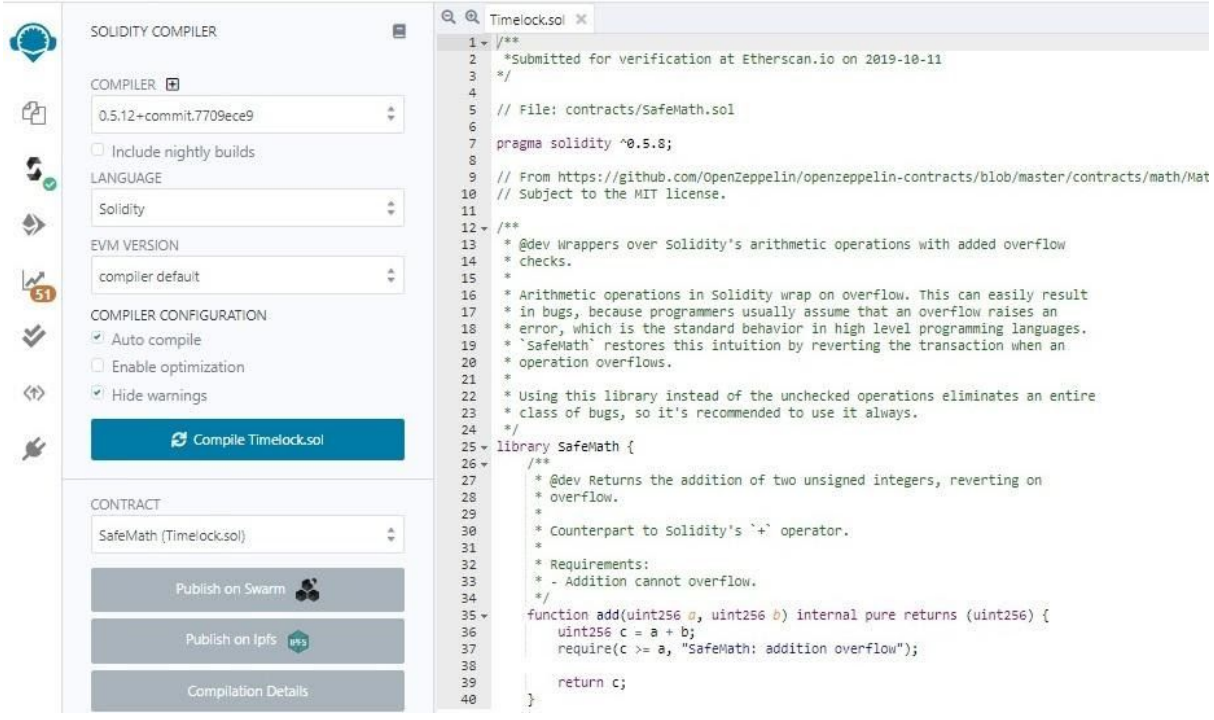
Remix is a powerful, open source tool that helps you write Solidity contracts straight from the browser. Written in JavaScript, Remix supports both usage in the browser and locally.

The Rubicon Protocol smart contracts were tested for various compiler versions. Except Timelock.sol all the contracts failed to compile in REMIX.






SOLIDITY COMPILER
COMPILER
0.5.12+commit.7709ece9
Include nightly builds
LANGUAGE
Solidity
EVM VERSION
compiler default
COMPILER CONFIGURATION
Auto compile
Enable optimization
Hide warnings
Compile SenateAlpha.sol
No Contract Compiled Yet
browser/SenateAlpha.sol:319:31: TypeError: The "chainid" instruction is only available for Istanbul-compatible VMs (you are currently compiling for "petersburg"). assembly { chainId := chainid() } ^-------^
SenateAlpha.sol
/**
*Submitted for verification at Etherscan.io on 2020-03-04
*/
pragma solidity ^0.5.12;
pragma experimental ABIEncoderV2;
contract SenateAlpha {
/// @notice The name of this contract
string public constant name = "Rubicon Senate Alpha";
/// @notice The number of votes in support of a proposal required in order for a quorum t
function quorumVotes() public pure returns (uint) { return 400000e18; } // 400,000 = 4% o
/// @notice The number of votes required in order for a voter to become a proposer
function proposalThreshold() public pure returns (uint) { return 100000e18; } // 100,000
/// @notice The maximum number of actions that can be included in a proposal
function proposalMaxOperations() public pure returns (uint) { return 10; } // 10 actions
/// @notice The delay before voting on a proposal may take place, once proposed
function votingDelay() public pure returns (uint) { return 1; } // 1 block
/// @notice The duration of voting on a proposal, in blocks
/// @notice LOWERED FOR TESTING PURPOSES NOT LAUNCH READY
function votingPeriod() public pure returns (uint) { return 1; } // ~3 days in blocks (as
/// @notice The address of the Rubicon Protocol Timelock
TimelockInterface public timelock;
/// @notice The address of the Rubicon governance token
RBCNInterface public RBCN;
/// @notice The address of the Senate Guardian
address public guardian;
/// @notice The total number of proposals
uint public proposalCount;


SOLIDITY COMPILER
COMPILER
0.5.12+commit.7709ece9
Include nightly builds
LANGUAGE
Solidity
EVM VERSION
compiler default
COMPILER CONFIGURATION
Auto compile
Enable optimization
Hide warnings
Compile Timelock.sol
CONTRACT
SafeMath (Timelock.sol)
Publish on Swarm
Publish on Ipfs
Compilation Details
Timelock.sol
/**
*Submitted for verification at Etherscan.io on 2019-10-11
*/
// File: contracts/SafeMath.sol
pragma solidity ^0.5.8;
// From https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/math/Mat
// Subject to the MIT license.
/**
* @dev Wrappers over Solidity's arithmetic operations with added overflow
* checks.
*
* Arithmetic operations in Solidity wrap on overflow. This can easily result
* in bugs, because programmers usually assume that an overflow raises an
* error, which is the standard behavior in high level programming languages.
* `SafeMath` restores this intuition by reverting the transaction when an
* operation overflows.
*
* Using this library instead of the unchecked operations eliminates an entire
* class of bugs, so it's recommended to use it always.
*/
library SafeMath {
/**
* @dev Returns the addition of two unsigned integers, reverting on
* overflow.
*
* Counterpart to Solidity's `+` operator.
*
* Requirements:
* - Addition cannot overflow.
*/
function add(uint256 a, uint256 b) internal pure returns (uint256) {
uint256 c = a + b;
require(c >= a, "SafeMath: addition overflow");
return c;
}

Closing Summary

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

Disclaimer

An audit is not a security warranty, investment advice, or an endorsement of the Rubicon Protocol. 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 Rubicon team put in place a bug bounty program to encourage further analysis of the smart contract by other third parties.