SmartDec

Viola Smart Contracts Security Analysis

This report is public.

Published: March 9, 2018

Abstract

In this report we consider the security of the Viola project. Our task is to find and describe security issues in the Viola smart contracts.

Procedure

In our audit, we consider the following crucial features of the smart contract code:

1. Whether the code is secure.
2. Whether the code corresponds to the documentation (including whitepaper).
3. Whether the code meets best practices in efficient use of gas, code readability, etc.

We perform our audit according to the following procedure:

- automated analysis
    - we scan the smart contracts with our own Solidity static code analyzer SmartCheck
    - we scan the smart contracts with several publicly available automated Solidity analysis tools such as Remix, Solhint, Oyente and Securify (beta version since full version was unavailable at the moment this report was made)
    - we manually verify (reject or confirm) the issues found by tools
- manual audit
    - we manually analyze the smart contracts for security vulnerabilities
    - we check the smart contracts logic and compare it with the one described in the documentation
    - we check ERC20 compliance
- report
    - we reflect all the gathered information in the report

Disclaimer

The audit does not give any warranties on the security of the code. One audit cannot be considered enough. We always recommend proceeding to several independent audits and a public bug bounty program to ensure the security of the smart contracts. Besides, security audit is not an investment advice.

Checked vulnerabilities

We have scanned the Viola smart contracts for commonly known and more specific vulnerabilities. Here are some of the commonly known vulnerabilities that we considered (the full list includes them but is not limited to them):

- Reentrancy
- 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

Project Overview

In our analysis we consider Viola whitepaper (viola-tech.pdf, sha1sum 37b5ad21130f85e1e6611e37c723364a1bedc9c7) and the Viola smart contracts code (Viola-Ethereum-develop.zip, sha1sum daf0b4e41ed6af7583c7893028c879d3fcc987bd).

The Latest Version Of Code

The developer has fixed some issues and sent us the new version of code:

- git repository with the latest commit b5894ec
- viola-tech.pdf, sha1sum 290cf93901e71830b017ab1bc9710341ed5edcb5

Project Architecture

For the audit, we have been provided with the following set of files:

- TokenERC20.sol
  Was changed to VLTToken.sol in the latest version of code.
- ViolaCrowdSale.sol - inherits the Ownable contract from the OpenZeppelin library version ^1.4.0 (at the time of the audit exact version is 1.6.0)
- Migrations.sol
- app.js

The provided file set is a truffle project and an npm package.

Code Logic

TokenERC20.sol contract implements ERC20 compatible burnable token (ERC20 standard compliance was verified in the audit). The token has the following parameters:

- Name: “VIOLA”
  Was changed to “VIOLET” in the latest version of code.
- Symbol: “VIOLA”
  Was changed to “VAI” in the latest version of code.
- Decimals: 18
- TotalSupply: 250 * 10**6 * 10 **18 (250M tokens)

Additional functionality:

- `approveAndCall` function. It approves specified amount to the specified sender, and calls `receiveApproval` function of the sender. The sender must implement `TokenRecipient` interface.
  `approveAndCall` was removed in the latest version of code.

- Upgradability of the token. The owner may set upgrader of `TokenUpgraderInterface`, then a token owner may upgrade his/her tokens by calling upgrade function, which transfers the senders token to the upgrader.
  Token upgradability was removed in the latest version of code.
- Payable fallback function that invokes upgrade of the senders tokens if the upgrader is set.

ViolaCrowdSale.sol contract implements the crowdsale logic as described below. The crowdsale contract basically meets the whitepaper with discrepancies as described below.
The crowdsale process goes through 6 states, enumerated in State enum as follows:

1. `Deployed` - after deployment until crowdsale parameters are set by `initaliseCrowdsale`
2. `PendingStart` - after `initaliseCrowdsale` until the crowdsale is started
3. `Active` - during actual crowdsale
4. `Paused` - crowdsale paused by the owner
5. `Ended` - crowdsale ended
6. `Completed` - after call to `completeCrowdSale`

Only whitelisted addresses may participate in the tokensale. An address is added to the whitelist by the owner of the sale at any state of the sale. The owner also assigns buy cap for the address, storing it in `maxBuyCap` mapping. `maxBuyCap[_investor]` value is updated after each token sale (value is decreased by the sale amount). However, `setWhitelistAddress` may be called again for the same address setting another cap value.

ViolaCrowdSale.sol smart contract has no explicit constructor. Initially the smart contract is created with parameters as follows:

- `state == Deployed`
- `bonusVestingPeriod == 60 days`
- `bonusTokenRateLevelOne == 20`
- `bonusTokenRateLevelTwo == 15`
- `bonusTokenRateLevelThree == 10`
- `bonusTokenRateLevelFour == 0`

Initialization of crowdsale parameters is performed by `initaliseCrowdsale.` The function accepts parameters as follows:

```
_startTime
_endTime
_rate
_tokenAddress
_wallet
```

`_tokenAddress` parameter is intended to be the address of the ERC20Token (implementation of Viola Token), created separately. It is assumed in the code, that the owner of the ERC20Token and ViolaTokenSale is the same.
`_wallet` is the recipient of the collected ETH after the token sale is completed.

`_startTime, _endTime, _tokenAddress, _wallet` cannot be changed after initialization. Token to ETH exchange rate (`_rate`) can be changed later.

`initaliseCrowdsale` does not check `state == Deployed`, that allows the owner to re-initialize the token sale any time (the bug is also described in Missing check section). `initialiseCrowdsale` sets the state to `PendingStart`.
The issue has been fixed by the developer and is not present in the latest version of code.

The contract owner may change the following important crowdsale parameter at any time:

- `minWeiToPurchase`
- `Rate` - ETH to token exchange rate
- `bonusTokenRateLevelOne` - bonus rate (in per cent) for tier 1 buyers
- `bonusTokenRateLevelTwo` - bonus rate (in per cent) for tier 2 buyers
- `bonusTokenRateLevelThree` - bonus rate (in per cent) for tier 3 buyers
- `bonusTokenRateLevelFour` - bonus rate (in per cent) for tier 4 buyers
- `leftoverTokensBuffer` - reserved token amount

The crowdsale is started by the `startCrowdSale` function. It can be invoked by any user provided that the current time is in the crowdsale period.

The crowdsale is ended by `endCrowdSale` function. It can be invoked by the owner at any time, and by any user provided that tokens has sold out. Only the owner may end the crowdsale if there are tokens left even if `endTime` is reached.

The owner may pause and unpause the active crowdsale by calling `pauseCrowdSale` and `unpauseCrowdSale` functions.

The owner may complete the crowdsale by calling `completeCrowdSale` function. Crowdsale completion is possible after the end of crowdsale (call to `endCrowdSale` function) and after all tokens either distributed or burned (by calling `burnExtraTokens`).

The owner may burn unsold tokens by calling `burnExtraTokens` function.

The owner may add an address to the whitelist and remove an address from the whitelist. If the address removed has already purchased tokens, it is refunded.

The owner may add an address to the list of “know your customer”-approved addresses and remove the address from the list of KYC-approved addresses. The removed address is refunded.

The default payable function assigns tokens to the buyer.

Only whitelisted addresses are allowed to buy tokens in one or several transactions capped by `maxBuyCap[investor]` value. `maxBuyCap[investor]` value is decreased by the transaction value in each transaction.

The total sum of investment for an investor is accumulated in `investedSum[investor]` state variable.

The incoming amount of ETH converted to tokens by ratio as specified by `‘rate’` state variable. The buyer is rewarded by bonus tokens depending on the time of purchase. The first day buyers are rewarded with `bonusTokenRateLevelOne` per cent bonus (default value 20%), the next two days buyers are rewarded with `bonusTokenRateLevelTwo` per cent bonus (default value 15%), the next 7 days buyers are rewarded with `bonusTokenRateLevelTree` per cent bonus (default value 10%), and the 11th day onward buyers are rewarded with `bonusTokenRateLevelFour` per cent bonus (default value 0%).

Purchased tokens and bonus tokens are stored separately in `tokensAllocated[investor]` and `bonusTokensAllocated[investor]` state variables.

An investor may claim his/her tokens by calling `claimTokens` function. Claiming is available after the end of token sale and if the investor is passed the KYC process. All normal tokens plus externally purchased tokens are transferred to this investor.

An investor may claim bonus tokens by calling `claimBonusTokens`. Claiming is available after the end of token sale, the end of lockdown period (60 days after the start of token sale), and if the investor is passed the KYC process. All bonus tokens plus externally assigned bonus tokens are transferred to the investor.

The crowdsale owner may distribute normal tokens to an investor by calling `distributeICOTokens` function. Distribution is available after the end of token sale. KYC check is not performed in this case. All normal tokens plus externally purchased tokens are transferred to this investor.

The crowdsale owner may distribute bonus tokens to an investor by calling `distributeICOTokens` function. Distribution is available after the end of token sale, the end of lockdown period (60 days after the start of token sale). KYC check is not performed in this case. All bonus tokens plus externally assigned bonus tokens are transferred to the investor.

The owner may register externally purchased tokens and bonus tokens by calling `externalPurchaseTokens` function. The investor, the amount of normal tokens and the amount of bonus tokens is specified in parameters of the function.
The owner may refund externally purchased tokens and bonus tokens to the specified investor.

The owner may add specified amount of normal and bonus tokens to any investor by calling `allocateTopupToken`. The purpose of this function is probably to compensate rate change at the end of token sale as specified in the whitepaper.

Function `emergencyERC20Drain` may be used by the owner to refund mistakenly sent tokens.

Automated Analysis

We used several publicly available automated Solidity analysis tools.
Securify does not support 0.4.18 compiler version; the specified version was changed in the code to 0.4.16 for this tool.
Here are the combined results of SmartCheck, Solhint, Securify, and Remix. Oyente has found no issues.
All the issues found by tools were manually checked (rejected or confirmed).

Tool	Vulnerability	False positives	True positives
Remix	Fallback function of contract requires too much gas	3	
	Gas requirement of function high: infinite	79	1
	Is constant but potentially should not be	2	
	Potential Violation of Checks-Effects-Interaction pattern	5	
	Use assert(x) / require(x)	1	
	Use of "now"		14
	Variables have very similar names	6	
Total Remix		96	15
SmartCheck	Balance Equality	1	
	Dos With Revert	12	
	Erc20 Approve		1
	Gas Limit And Loops		1
	Locked Money	1	
	Pragmas Version		2

	Redundant Fallback Reject		1
	Reentrancy External Call	19	
	Unchecked Math	18	6
	Underflow Overflow		1
	Visibility		1
Total SmartCheck		51	13
Solhint	Compiler version must be fixed		6
	Event and function names must be different	3	
	Explicitly mark visibility of state		1
	Fallback function must be simple	1	
Total Solhint		4	7
Overall Total		151	35

Securify* — beta version, full version is unavailable.

Cases where these issues lead to actual bugs or vulnerabilities are described in the next section.

Manual Analysis

Contracts were completely manually analyzed, their logic was checked and compared with the one described in the documentation. Besides, the results of automated analysis were manually verified. All confirmed issues are described below.

Critical issues

Critical issues seriously endanger smart contracts security. We highly recommend fixing them.

Missing state check

`initaliseCrowdsale` does not check `state == Deployed`, that allows the owner to re-initialize the token sale any time. We highly recommend adding the corresponding check.
<u>The issue has been fixed by the developer and is not present in the latest version of code.</u>

Medium severity issues

Medium issues can influence smart contracts operation in the current implementation. We highly recommend addressing them.

Discrepancies with the whitepaper

Crowdsale end date. The whitepaper specifies (pg. 40) that the crowdsale lasts for 30 days. However, the crowdsale end date is specified explicitly in `initialiseCrowdsale` function. The crowdsale period may differ from the specified in the whitepaper.
<u>The issue has been fixed by the developer and is not present in the latest version of code.</u>

Tokens for Crowdsale. The whitepaper specifies (pg. 40) that 37% of total supply (of 250M tokens) are assigned for the crowdsale. However, the smart contract contains no such provision. As the crowdsale is performed between whitelisted and KYC'ed participants, the crowdsale owners should assign the buy cap for the participants accordingly.
<u>The developer is aware about the issue and will set the 37% share manually.</u>

Token Sale Schedule. According to the whitepaper (pg. 41) the crowdsale is three-tiered: the first tier (first 2 days) buyers are rewarded with first tier bonuses, the second tier (next 8 days) buyers are rewarded with second tier bonuses, and the third tier (next 20 days) buyers are rewarded with third tier bonuses. But according to the smart contract the crowdsale is four-tiered: the first day, the next two days, the next 7 days, and the 11th day onward.
<u>The issue has been fixed by the developer: the whitepaper has been updated.</u>

Token Sale Rewards. The whitepaper does not specify time-based bonus rewards (see "Token Sale Schedule" above). The smart contract specifies the rewards as 20% of bought tokens for the first tier, 15% for the second, 10% for the third, and 0% for the fourth. Note, that the owner may change the rewards anytime before and during the crowdsale.
<u>The issue has been fixed by the developer: the whitepaper has been updated.</u>

Bonus Token Lockdown. The whitepaper specifies (pg. 42) 60-day lockdown for the bonus token, but does not specify when 60 days start counting. According to the smart contract 60 days are counted from the crowdsale start time.
<u>The issue has been fixed by the developer: the whitepaper has been updated.</u>

Bonus Token Rate. The whitepaper specifies (pg. 42) that the bonus tokens are calculated based on the exchange rate at the point of purchase, or the end of token sale, whichever is higher. According to the smart contract the bonus tokens are calculated only at the point of purchase.
<u>The developer is aware about the issue and will set the top-up of bonus tokens manually.</u>

Token Exchange Rate. The whitepaper specifies (pg. 41) that amount of VIOLA to be issued is pegged against the exchange rates at the point of purchase and at the end of the token sale, whichever is highest. According to the smart contract no end-of-sale token adjustment is performed. However, the smart contract do implement 'allocateTopupToken' function, which allows assigning arbitrary amount of tokens and bonus tokens to any used by the contract owner. It is possible that main and bonus token amount adjustment is intended to be performed by the contract owner on an ad-hoc basis.
<u>The developer is aware about the issue and will set the top-up of tokens manually.</u>

Token Upgradability. The VIOLA token implementation (TokenERC20.sol) supports token upgrade. This feature is not documented in the whitepaper.
<u>The feature has been removed in the latest version of code.</u>

It should be noted, that the smart contract code allows a great degree of control over the crowdsale process from the contract owner. It is possible that the contract owner will control the crowdsale to meet the whitepaper provisions as specified.

ERC20 standard violation

Viola Token implementation (TokenERC20) does not fully conform to the ERC20 standard.

- Token does not support sending 0 tokens (see https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md:
  Transfers of 0 values MUST be treated as normal transfers and fire the Transfer event).
  <u>The issue has been fixed by the developer and is not present in the latest version of code.</u>
- Approval event is not triggered after successful operation with allowances (see https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md:
  MUST trigger on any successful call to approve(address _spender, uint256 _value)).

The issue has been fixed by the developer and is not present in the latest version of code.

Gas limit and loops

The ViolaCrowdSale contract functions contain loops with unknown number of iterations.

- ViolaCrowdSale.sol, line 201

```
    for (uint counter = 0; counter < registeredAddress.length;
counter ++)
```

If the arrays are large enough, the functions will not fit in the block gas limit and the transactions calling it will thus never be confirmed. If the array can be influenced by an attacker (e.g., if an attacker can register arbitrary number of investor accounts), this can lead to an attack.
We highly recommend avoiding loops with big or unknown number of steps.
The issue has been fixed by the developer and is not present in the latest version of code.

Potential tokens loss

There is no way to retrieve tokens mistakenly sent to TokenERC20 contract address. We recommend adding such functionality to the contract.
The issue has been fixed by the developer and is not present in the latest version of code.

Potential money loss

If the TokenERC20 contract receives money in the `upgradable` state, tokens assigned to the sender are transferred to another contract, but the received amount of money remains on the balance of this contract.

- TokenERC20.sol, line 232

  `assert(upgrade());`

The issue has been fixed by the developer and is not present in the latest version of code.

Assertion Failure

Assertion is used incorrectly.

- TokenERC20.sol, `upgrade` function, line 213

  `assert(upgrader.upgradeFor(msg.sender, value));`
- TokenERC20.sol, `upgradeFor` function, line 226

  `assert(upgrader.upgradeFrom(msg.sender, _for, _value));`
- TokenERC20.sol, fallback function, line 232

  `assert(upgrade());`

There is a difference between `assert` and `require`. `require` is supposed to be used to check the pre-conditions, e.g., that the function arguments fulfill certain criteria. `assert`, on the other hand, is meant to check for internal invariants of the contract. `assert` is never supposed to evaluate to true; if is happens, that means that the contract state is logically broken. Proper usage of `assert` vs `require` also helps to reduce false positive rates of

some verification tools. We recommend replacing `assert` with `require`.
<u>The issue has been fixed by the developer and is not present in the latest version of code.</u>

Operations order

Solidity division is integer division, so the fractional part of the result is lost.

- ViolaCrowdSale.sol, line 413

```
    uint bonusTokens =
tokens.div(100).mul(getTimeBasedBonusRate());
```

Therefore, to calculate the given percent of the value, the division operation should be performed after the multiplication, that is, RESULT = VALUE * PERCENT / 100.
Note, that the multiply operation should be checked for overflow.
<u>The issue has been fixed by the developer and is not present in the latest version of code.</u>

Payable fallback

The fallback function should not be payable.
TokenERC20.sol, line 230

```
function () payable {
    ...
    }
```

We do not recommend using the payable fallback.
<u>The issue has been fixed by the developer and is not present in the latest version of code.</u>

Low severity issues

Low severity issues can influence smart contracts operation in future versions of code. We recommend to take them into account.

OpenZeppelin version

The package.json does not specify the exact version of the OpenZeppelin library:

- package.json, line 25

```
    "zeppelin-solidity": "^1.4.0"
```

Thus, with the release of version 1.6.0 (it came out at the time of the audit) a new version will be used. But incompatible changes can occur.
For example, the location of the ERC20 token related files has changed in version 1.6.0 (compared to version 1.4.0), so the project ceased building successfully.
We recommend specifying the exact versions of the OpenZeppelin library in package.json.

Pragmas version

Solidity source files indicate the versions of the compiler they can be compiled with.
Example:

```
    pragma solidity ^0.4.18; // bad: compiles w 0.4.18 and above
```

```
pragma solidity 0.4.18; // good: compiles w 0.4.18 only
```

We recommend following the latter example, as future compiler versions may handle certain language constructions in a way the developer did not foresee. Besides, we recommend using the latest compiler version (0.4.20 at the moment).
The issue has been fixed by the developer and is not present in the latest version of code.

Implicit visibility level

There is a variable with implicit visibility level.

- TokenERC20.sol, line 49

```
mapping (address => mapping (address => uint256)) allowed;
```

We recommend specifying visibility level explicitly and correctly.
The issue has been fixed by the developer and is not present in the latest version of code.

ERC20 approve issue

There is ERC20 approve issue (TokenERC20.sol, line 128). We recommend implementing `increaseApproval`/`decreaseApproval` functions (they can be found in OpenZeppelin contracts). Also we recommend instructing users not to use approve directly and to use `increaseApproval/decreaseApproval` functions instead.
Another option is to change the approved amount to 0, wait for the transaction to be mined, and then to change the approved amount to the desired value — link.
Changing the approved amount from a nonzero value to another nonzero value allows a double spending with a front-running attack.
The developer is aware about the issue and will take corresponding measures.

Unchecked math

Solidity is prone to an integer over- and underflow. Overflow leads to unexpected effects and can lead to loss of funds if exploited by malicious account. We recommend using the SafeMath library for all arithmetic operation. It is not used in ViolaCrowdSale.sol, lines 420, 421, 579, 580, 637, 638.
This may lead to overflow.
The issue has been fixed by the developer and is not present in the latest version of code.

Redundant fallback

The fallback function is redundant.

- TokenERC20.sol, line 230

```
function () payable {
        if (upgradable) {
            assert(upgrade());
            return;
        }
      revert()
      }
```

If state is `upgradable` it is possible to call the `upgrade` function directly.

Contracts should reject unexpected payments. Before Solidity 0.4.0, it was done manually. Starting from Solidity 0.4.0, contracts without a fallback function automatically revert payments, making the code above redundant. We recommend removing the fallback function to save on the gas.
The issue has been fixed by the developer and is not present in the latest version of code.

Redundant code

The check is redundant: numbers are always more or equal 0.

- ViolaCrowdSale.sol, line 583
  `assert(externalBonusTokensAllocated[_investor] >= 0);`

We recommend avoiding redundant checks to save on the gas.
The issue has been fixed by the developer and is not present in the latest version of code.

Variable duplication

The `allowance` and `allowed` state variables are declared in the contract.

- TokenERC20.sol, lines 48, 49

`allowed` is never set in the code and is not `public`. It is only used in `upgradeFor` method. It seems, that `allowed` variable should be removed in favour of `allowance`.
The issue has been fixed by the developer and is not present in the latest version of code.

Explicit initialization

Some critical crowdsale parameters, such as `startDate`, `endDate`, `rate`, `wallet`, `token` are assigned by call to `initaliseCrowdSale` function. We recommend assigning all critical parameters in a constructor. I.e. functionality of `initaliseCrowdSale` should be moved to `ViolaCrowdSale` function (constructor).
The issue is known to the developer and is a part of design.

Spelling issues

There are some spelling issues in the code.

- TokenERC20.sol, line 27
  Interface identifier `tokenRecipient` should start with the capital T letter, i.e. `TokenRecipient`
- TokenERC20.sol, line 107
  `function initaliseCrowdsale (...)`
  There is a spelling error in `initaliseCrowdsale`. It should be `initialiseCrowdsale` or `initialiseCrowdSale`
- `‘crowdsale’` word is used inconsistently in the code: sometimes `‘Crowdsale’`, sometimes `‘CrowdSale’`. We highly recommend using only one of the options.

The issue has been fixed by the developer and is not present in the latest version of code.

Javascript codestyle

Codestyle issues influence code conciseness and readability and in some cases may lead to bugs in future. We recommend to take them into account. During the analysis of codestyle problems we've found JS style guide violations and errors: missed semicolons, unexpected compression type coercion, wrong JSDOC.
We highly recommend using any linter, e.g. Eslint with the default configuration or some custom configuration - https://www.npmjs.com/package/eslint-config-defaults.

Conclusion

In this report we have considered the security of the Viola smart contracts. We performed our audit according to the procedure described above.
The audit showed several vulnerabilities of different severity level. Most of the issues (and all of the important ones) either have been fixed by the developer in the latest version of code or are known to the developer and will be managed manually.

This analysis was performed by SmartDec.

Alexandr Chernov, Chief Research Officer
Katerina Troshina, Chief Executive Officer
Evgeniy Marchenko, Lead Developer
Pavel Yuschenko, Chief Technical Officer
Elizaveta Kharlamova, Analyst
Ivan Ivanitskiy, Chief Analytics Officer
Alexander Seleznev, Chief Business Development Officer

Sergey Pavlin, Chief Operating Officer [illegible]

March 9, 2018

Appendix

Code coverage

```
---------------------|----------|----------|----------|-------
---|----------------|
File                 |  % Stmts | % Branch |  % Funcs |  %
Lines |Uncovered Lines |
---------------------|----------|----------|----------|-------
---|----------------|
 contracts/          |    82.69 |    52.45 |    83.33
|    82.01 |                |
  TokenERC20.sol     |    45.07 |    18.52 |       50
|    39.06 |... 232,233,235 |
  ViolaCrowdsale.sol |    95.28 |    64.67 |    95.83
|    94.86 |... 640,645,646 |
---------------------|----------|----------|----------|-------
---|----------------|
All files            |    82.69 |    52.45 |    83.33
|    82.01 |                |
---------------------|----------|----------|----------|-------
---|----------------|
```

Tests

```
Contract: ViolaCrowdsale
    initializing contract
      ✓ should initialize with PendingStart status
    starting crowdsale
      ✓ should start crowdsale from PendingStart status (80ms)
      ✓ should not start crowdsale in Active status (106ms)
    burning token
      ✓ should decrease contract allowance (168ms)
      ✓ should decrease contract allowance (200ms)
    ending crowdsale
      ✓ should end crowdsale from Active status (108ms)
      ✓ should not end crowdsale from Paused status (128ms)
      ✓ should not end crowdsale from Ended status (123ms)
      ✓ should not end crowdsale from Completed status (203ms)
      ✓ should allow owner to transfer eth partially (316ms)
      ✓ should not allow owner to transfer eth more than non
kyc refund funds (345ms)
      ✓ should not allow owner to transfer eth more than
available fund (338ms)
    pausing crowdsale
      ✓ should pause crowdsale from Active status (88ms)
      ✓ should unpause crowdsale from Paused status (115ms)
    completing crowdsale
      ✓ should not end when didnt hit buffer (424ms)
      ✓ should auto end when hit buffer (429ms)
      ✓ should auto end when sold out (427ms)
      ✓ should complete crowdsale from Ended status (191ms)
      ✓ should transfer funds when crowdsale ended (558ms)
    setting whitelist address
      ✓ should accept whitelist address (46ms)
      ✓ should not accept 0 cap
      ✓ should not accept 0x0 address
    removing whitelist address
      ✓ should remove whitelist address (73ms)
      ✓ should refund after removal (502ms)
    setting bonus token rates
      ✓ should update bonus rate for level one (38ms)
      ✓ should update bonus rate for level two (56ms)
      ✓ should update bonus rate for level three
      ✓ should update bonus rate for level four (38ms)
    bonus rate
      ✓ at the beginning of Day 1 should be 30
      ✓ at the end of Day 1 should be 30 (45ms)
      ✓ at the beginning of Day 2 should be 15 (40ms)
      ✓ at the end of Day 3 should be 15
      ✓ at the beginning of Day 4 should be 8
      ✓ at the end of Day 10 should be 8
      ✓ at the beginning of Day 11 should be 4
      ✓ at the end should be 4
      ✓ after ending of ICO should be 0
    setting rate
      ✓ should accept rate
      ✓ should not accept 0 rate
    tokens
      ✓ should get tokens left
    buying token
      ✓ should transfer funds to contract (404ms)
      ✓ using fiat and eth should tally total tokens (201ms)
      ✓ using fiat and eth should tally total bonus tokens
(204ms)
      ✓ investor should get tokens (152ms)
      ✓ non-whitelisted investor should not be able to buy
tokens (122ms)
      ✓ should not buy when contract has ended (173ms)
      ✓ should not buy when contract is paused (129ms)
      ✓ should not buy when contract is completed (213ms)
      ✓ should not buy when insufficient token (156ms)
      ✓ should not buy when cap is reached (334ms)
      ✓ should not buy using fiat when cap reached (207ms)
      ✓ should update total allocated tokens when purchased
externally (105ms)
      ✓ should buy minWei (174ms)
      ✓ should not buy below minWei (139ms)
      ✓ should buy above minWei (181ms)
    allocate Tokens
      ✓ buyer should receive 30% bonus tokens within first
days (181ms)
      ✓ buyer should receive 15% bonus tokens from Day 2
(159ms)
      ✓ buyer should receive 8% bonus tokens from Day 4
(154ms)
      ✓ buyer should receive 4% bonus tokens from Day 11
(185ms)
    distributing tokens
      ✓ should distribute ICO tokens (78ms)
      ✓ should distribute bonus tokens (95ms)
      ✓ should not distrubte bonus tokens before vesting
period
    claiming tokens
      ✓ investor should claim ICO tokens (108ms)
      ✓ investor should claim bonus tokens (130ms)
      ✓ investor should not claim bonus tokens before vesting
period
    refunding partially
```

```
      ✓ should not have refund amount more than invested
amount
      ✓ should not have refund tokens more than allocated
tokens
      ✓ should not have refund bonus tokens more than
allocated bonus tokens
      ✓ should reduce the invested sum by the
```