ODEM ICO Smart Contract Code Audit

Release

G. Baecker

Feb 12, 2018

CONTENTS:

CHAPTER

ONE

INTRODUCTION

The *ODEM* Crowdsale is an ICO based on Ethereum smart contracts written in the Solidity programming language, starting in the middle of February 2018.

Prior to deploying this project onto the Ethereum main network, extensive code tests and several code audits were performed on the involved smart contracts.

This document is a report of one of those audits and subdivided into three main parts.

1. The first part describes the overall structure of the *ODEM* contracts a gives a detailed description of the functionality, features and lifecycle of the single contracts.
2. The second part contains the code review. It contains the findings after intensive reading of the projects source codes, comments on the coding style and thoughts about the vulnerability to some known attacks.
3. The third part is about testing – which test cases were identified, how the tests were carried out.

Note: Disclaimer

This audit is not a legally binding document and it doesn't guarantee anything.

Nevertheless it is a result of careful work done for the best.

That it's just a discussion document.

Version note: The smart contract project code was fetched from *Github*:

URL	https://github.com/odemio/ico.git
Date	Feb 3, 2018, 16:42
Commit	`42b9404431be3da972bc9b671a3b8546cdc36f81`

CHAPTER

TWO

FUNCTIONALITY

This chapter describes the overall structure and the functionality in detail of the contracts that constitute the *ODEM ICO* smart contract project.

Preliminary Note

Here, the term “tokens” always refers to the smallest unit, where all token amounts are whole numbers.

2.1 Contract Structure

The project consists of four contracts which by themselves are based on *OpenZeppelin*'s *Solidity* framework.

- *Whitelist*
- *ODEMCrowdsale*
- *ODEMToken*
- *TeamAndAdvisorsAllocation*

All contracts in this project are assumed to be singletons, i.e. there will be created exactly one instance for every contract in the project.

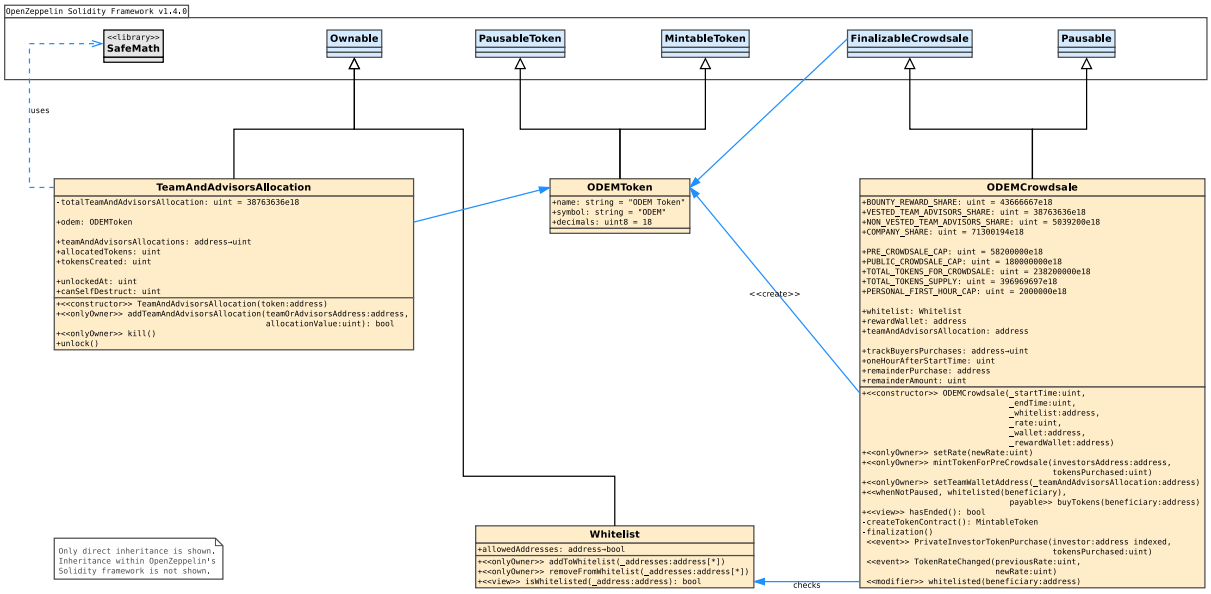


Diagram of ODEM ICO contracts

The contract *TeamAndAdvisorsAllocation* is loosely coupled to the other contracts, as it’s not essential for the crowdsale or token trading. Anytime before crowdsale finalization the crowdsale’s contract owner

can decide to replace it by another implementation or even to set a regular user account becoming the crowdsale instance's `teamAndAdvisorAllocation` address.

2.2 Deployment Order

The contracts must be deployed in the following order:

1. **Whitelist** has to be deployed manually at first. Its address is needed in the next step.
2. **ODEMCrowdsale** has to be deployed manually after `Whitelist`. Attention has to be payed to correct values of parameters `_startTime`, `_endTime`, `_whitelist`, `_wallet`, and `_rewardWallet` as these cannot be changed after deployment.
3. **ODEMToken** will automatically be deployed during initialization of `ODEMCrowdsale`. To determine the token's address call the public getter function `token()` of the crowdsale instance.
4. **TeamAndAdvisorsAllocation** has to be deployed manually before the crowdsale gets finalized. As this contract will start a 182 day retention period on initialization, it should be deployed at the last possible moment, that is after the crowdsale has ended, before calling `finalize()` of the crowdsale contract instance.

Warning: Before deploying `ODEMCrowdsale` it must be assured that the deployed `Whitelist` instance is fully functional, i.e. its owner can add and remove addresses and their `isWhitelisted` status is as expected.

Rationale: the constructor of `ODEMCrowdsale` will accept any address as `_whitelist` parameter, but the actual usage of `Whitelist`, that is to check for if an address was whitelisted, doesn't happen before the crowdsale period starts.

2.3 Whitelist

The whitelist contract constitutes a (quite generic) mutable set of addresses.

In context of *ODEMCrowdsale* it will contain the addresses of investors who are allowed to buy tokens during the crowdsale period.

When creating an *ODEMCrowdsale* instance, the address of an existing, i.e. deployed, *Whitelist* instance has to be given.

2.3.1 Features

Ownable

The owner if a *Whitelist* instance can transfer the ownership at any time to any other account.

Mutability

Addresses can be added to or removed from a *Whitelist* instance by its owner at any time.

2.3.2 Accounts/Roles

Owner

Only the *Whitelist* instance's owner is allowed to add/remove addresses to/from the whitelist.

Initially, this will be the account who deployed the *Whitelist* instance in the first place, but ownership can be transferred later by the current owner to any other account.

The owner doesn't need to be the same as of the *ODEMCrowdsale* instance.

Whitelisted Accounts

Their addresses are stored in the contract instance.

2.3.3 Lifecycle

Whitelist's behavior is invariant with respect to time.

2.3.4 Constraints

Adding/Removing

Only the owner is allowed to add or remove addresses.

There are no time related restrictions on adding or removing entries, but the *ODEMCrowdsale* instance will read this set only during the crowdsale period.

2.4 ODEMCrowdsale

The *ODEMCrowdsale* is the main contract of this project.

When deployed it creates a new *ODEMToken* instance, thus becoming its owner.

2.4.1 Constants

The following token related constants are defined:

- **BOUNTY_REWARD_SHARE**

  = `43666667e18` (~ 43.7M $\times 10^{18}$ tokens)

  Amount of tokens minted in favor of the reward wallet account on crowdsale finalization.

- **VESTED_TEAM_ADVISORS_SHARE**

  = `38763636e18` (~ 38.8M $\times 10^{18}$ tokens)

  Amount of tokens minted in favor of *TeamAndAdvisorAllocation* instance on crowdsale finalization.

- **NON_VESTED_TEAM_ADVISORS_SHARE**

  = `5039200e18` (~ 5M $\times 10^{18}$ tokens)

  Amount of tokens minted in favor of the wallet account on crowdsale finalization.

- **COMPANY_SHARE**

  = `71300194e18` (~ 71.3M $\times 10^{18}$ tokens)

  Amount of tokens minted in favor of the wallet account on crowdsale finalization.

- **PRE_CROWDSALE_CAP**

  = `58200000e18` (~ 58.2M $\times 10^{18}$ tokens)

  Maximum total amount of tokens minted in favor of the private investors during pre-crowdsale.

- **PUBLIC_CROWDSALE_CAP**

  = `180000000e18` (~ 180M $\times 10^{18}$ tokens)

  Maximum total amount of tokens bought by investors during crowdsale if the pre-crowdsale cap was reached.

If pre-crowdsale cap was not reached, the remaining tokens can be bought additionally during crowdsale.

- **TOTAL_TOKENS_FOR_CROWDSALE**

  = `PRE_CROWDSALE_CAP + PUBLIC_CROWDSALE_CAP` (~ 238.2M ×10 ^{18} tokens)

  Maximum total amount of tokens minted during pre-crowdsale and crowdsale.

- **TOTAL_TOKENS_SUPPLY**

  = `396969697e18` (~ 397M ×10 ^{18} tokens)

  Total supply of tokens after finalization.

  Should be greater than or equal to:

```
  BOUNTY_REWARD_SHARE
+ VESTED_TEAM_ADVISORS_SHARE
+ NON_VESTED_TEAM_ADVISORS_SHARE
+ COMPANY_SHARE
+ TOTAL_TOKENS_FOR_CROWDSALE
```

  The remaining tokens will be minted in favor of the wallet account.

- **PERSONAL_FIRST_HOUR_CAP**

  = `2000000e18` (~ 2M ×10 ^{18} tokens)

  Maximum amount of tokens an investor can buy during the first hour after crowdsale start.

2.4.2 Features

Ownable

The owner of an *ODEMCrowdsale* instance can transfer the ownership at any time to any other account.

Pausable

During the crowdsale (i.e. from start till end, see *Lifecycle*) the sale of tokens to investors can be halted and continued by the *ODEMCrowdsale* instance's owner.

Pausing in other periods is possible but without any effects.

Finalizable

After the end of crowdsale, the *ODEMCrowdsale* instance has to be finalized manually (or by any off-chain automatism) to enable the free trade/transfer of tokens.

This can be done:

- solely by the owner
- only after the crowdsale has ended
- only once

2.4.3 Accounts/Roles

Owner

The owner of an *ODEMCrowdsale* instance is the account who created/deployed it in the first place and can

- transfer ownership at any time to any other account
- adjust the (tokens per wei) rate at any time to any non-zero value

- mint tokens for the benefit of any private investor (in fact any account) as long as the total token supply will not exceed the pre-crowdsale cap (see *Constants*: `PRE_CROWDSALE_CAP`) and the crowdsale has not started yet
- set the address of a deployed `TeamAndAdvisorsAllocation` instance
- finalize the contract instance after the crowdsale has ended (and only if it wasn't finalized already) and an address for team wallet was set

Wallet

This account (a multisig wallet) will hold the crowdsale funds.

The wallet address must be given when creating a *ODEMCrowdsale* instance and cannot be changed afterwards.

When the crowdsale is finalized, a predefined amount of tokens (see *Constants*: `NON_VESTED_TEAM_ADVISORS_SHARE`, `COMPANY_SHARE`) plus all remaining tokens, i.e. the difference of total tokens supply to some constant (see *Constants*: `TOTAL_TOKENS_SUPPLY`) will be minted for the benefit of the wallet account.

Reward Wallet

When the crowdsale is finalized, a fixed amount of tokens from the bounty and rewards campaign (see *Constants*: sol:*BOUNTY_REWARD_SHARE*) will be minted for the benefit of this account.

Private Investors

Before the crowdsale starts, the *ODEMCrowdsale*'s instance owner can mint tokens for the benefit of private investor accounts. This won't be possible from the crowdsale's start time on.

The total amount of pre-crowdsale minted tokens is capped (see *Constants*: `PRE_CROWDSALE_CAP`).

Regular Investors

Regular investors are able to purchase tokens during the crowdsale period, but only if the crowdsale wasn't paused by the *ODEMCrowdsale* instance's owner.

Every investor is only allowed to buy tokens for the benefit of himself/herself, and prior to this his/her account address has to be whitelisted by the *Whitelist*'s instance owner.

During the first hour after crowdsale start, the amount of tokens a single investor can buy is capped (see *Constants*: `PERSONAL_FIRST_HOUR_CAP`), which is independent of if the investor already received tokens due to pre-crowdsale minting.

The total amount of tokens that can be bought during crowdsale is capped (see *Constants:* `TOTAL_TOKENS_FOR_CROWDSALE`).

If the last investor tries to buy more tokens than are available, he/she will get the remaining ones (with respect to the cap) and his/her address along with the overpaid amount of ether will be stored for later refund. These refunds will be paid out manually.

Token

An *ODEMCrowdsale* instance will create a new *ODEMToken* instance, become its owner, and store its address during initialization (i.e. deployment).

Transferring the ownership of the *ODEMCrowdsale* instance doesn't affect the ownership of its assigned *ODEMToken* instance (it will remain the crowdsale contract instance).

2.4.4 Lifecycle

Pre-Crowdsale

During initialization, i.e. deployment, of an *ODEMCrowdsale* instance, a paused *ODEMToken* instance will be created, and the following state variables will be stored:

- start and end time of crowdsale period

- wallet and rewardWallet addresses
- address of prior to this created *Whitelist* instance
- address of newly created *ODEMToken* instance
- (tokens per wei) rate

The rate can be changed by the owner at any later point in time, the other state variables not.

Until the start of crowdsale the owner may mint tokens (with respect to the given cap) for the benefit of private investors.

First Hour of Crowdsale

Within the first hour after the start of crowdsale regular investors can buy a per investor limited amount of tokens for themselves.

Investors must be whitelisted prior to be able to purchase tokens. The whitelisting of an account can be done and undone at any point in time.

Remaining Duration of Crowdsale

Investors can buy tokens for themselves as long as the cap of total available tokens is not reached.

Investors must be whitelisted prior to be able to purchase tokens. The whitelisting of an account can be done and undone at any point in time.

End of Crowdsale

The crowdsale ends if either the crowdsale period elapsed or all available tokens were sold to investors.

In the latter case the crowdsale will end before its predefined end time.

Finalization

After end of crowdsale it has to be finalized manually by the owner. As prerequisite, the address of a `TeamAndAdvisorsAllocation` instance has to be set prior to finalization.

A fixed amount of tokens gets minted for benefit of the *TeamAndAdvisorAllocation* instance, the wallet and reward wallet.

If the total supplied tokens is below a predefined cap (see *Constants*: `TOTAL_TOKENS_SUPPLY`) , the remaining tokens (i.e. the difference) will be minted for the benefit of the wallet account.

The *ODEMToken* instance will be unpaused, so that tokens become free tradable/transferable.

2.4.5 Constraints

Amount of tokens

The amount of tokens minted during different periods must not exceed the given limits (see *Constants*).

Adjustment of rate

The (tokens per wei) rate must be set by the owner only. This can be done at any time.

Private Investments

Private investors may get tokens only before the crowdsale starts. The pre-crowdsale token minting can be triggered by the owner only.

Regular Investments

Investors must be whitelisted prior to be able to buy tokens. The beneficiary of a tokens purchase is always the purchaser (= investor) himself/herself.

If the crowdsale was paused, no token purchase is possible.

2.5 ODEMToken

ODEMToken is an ERC20 compliant token contract.

It isn't meant to be deployed by itself, but an instance of it to be automatically created when a new *ODEMCrowdsale* instance is initialized. The latter will become the *ODEMToken* instance's owner.

So the following sections refer to an *ODEMToken* instance created and owned by an existing *ODEMCrowdsale* instance.

2.5.1 Features

Ownable

The *ODEMToken* contract is *Ownable*, thus exposing a method to its owner for transferring the ownership to a new address. But since *ODEMCrowdsale* doesn't use this feature, it will stay its *ODEMToken*'s owner forever.

Pausable

The trade of tokens, i.e. transfer from one account to another, of tokens can be halted and continued by its owner (see *Lifecycle*).

Mintable

The *ODEMToken* instance's owner is able to mint some tokens, i.e. create new tokens and increase any account's token balance.

2.5.2 Accounts/Roles

Owner

When an *ODEMCrowdsale* instance is initialized, i.e. deployed, it will create a new *ODEMToken* instance. Hence the *ODEMCrowdsale* instance will own the new *ODEMToken* instance.

Token holders

The *ODEMToken* contract by itself doesn't impose any restrictions to which accounts can hold tokens.

But as the contract instance is meant to be created and owned by an *ODEMCrowdsale* instance, there are some limitations on how to get tokens:

1. Being one of the initial investors, thus getting some tokens minted by the *ODEMCrowdsale* instance owner before the crowdsale period starts (pre-crowdsale).
2. Becoming whitelisted by the *ODEMWhitelist* instance's owner, thus being allowed to purchase tokens during the crowdsale period.
3. The predefined wallet and rewardWallet accounts will get some tokens at the end of crowdsale period.
4. As a team member or an advisor one can get tokens allocated by the *TeamAndAdvisorsAllocation*'s instance owner, after its 182 retention period has ended.
5. Being the receiver of an ERC20 compliant token transfer after the crowdsale has ended.

So while the circle of token holders is limited until the crowdsale period ends, the *ODEMToken* is freely tradable afterwards.

2.5.3 Lifecycle

Paused

When an *ODEMToken* instance gets created by an *ODEMCrowdsale* instance, its state will be set paused.

Token minting is possible but trade/transfer is not.

Unpaused

After the crowdsale period has ended, the *ODEMCrowdsale* instance has to be finalized manually (or by any off-chain automatism).

This will make the *ODEMCrowdsale* instance to unpause its *ODEMToken* instance, thus making tokens transferable from token holders to any *Ethereum* accounts.

ODEMCrowdsale ensures that minting of tokens is not possible anymore.

2.5.4 Constraints

The *ODEMToken* by itself doesn't impose any restrictions on

- when it is paused/unpaused
- beneficiaries of minted or transferred tokens

as these are controlled by the owning *ODEMCrowdsale* instance.

Pause/Unpause

The pause/unpause state can be changed by the owning *ODEMCrowdsale* instance only.

Minting

The amount and receivers of minted tokens is controlled by the owning *ODEMCrowdsale* instance only.

Total Supply

The maximum total supply of tokens is controlled by the owning *ODEMCrowdsale* instance's minting restrictions and won't exceed `TOTAL_TOKENS_FOR_CROWDSALE` (see *ODEMCrowdsale Constants*) before crowdsale finalization.

After crowdsale finalization the total amount of tokens is fixed to `TOTAL_TOKENS_SUPPLY` (see *ODEMCrowdsale Constants*).

2.6 TeamAndAdvisorAllocation

A *TeamAndAdvisorAllocation* instance has to be deployed prior to finalization of crowdsale. It receives a fixed share of *ODEMToken* tokens (see *ODEMCrowdsale Constants*: `VESTED_TEAM_ADVISORS_SHARE`), thus becoming a token holder.

It allows the distribution of its tokens to team members and advisors, which they can transfer to their own accounts as soon as the retention period has expired.

2.6.1 Constants

- **totalTeamAndAdvisorsAllocation** = `38763636e18` (~ 38.8M ×10 :sup: 18 tokens)

  Total maximum amount of tokens that can get assigned to team members and advisors.

2.6.2 Features

Ownable

The *TeamAndAdvisorsAllocation* contract is *Ownable*, thus exposing a method to its owner for transferring the ownership to a new address.

Retention period

The withdrawal of tokens is blocked for 182 days after the finalization of the crowdsale.

Destruction

At least 365 days after finalization of the crowdsale, this contract instance can be destroyed by the contract's owner.

2.6.3 Accounts/Roles

Owner

The owner can assign token shares to team members and advisors.

One year after the creation of this contract instance the owner can destroy it.

Team Member or Advisor

To these accounts a share in tokens will be assigned.

After the expire of the initial retention period, they can unlock (i.e. withdraw) their share in tokens, which will be transferred to their accounts.

2.6.4 Lifecycle

Retention Period

During the first 182 days after contract instance creation the token share of team members and advisors can be set, but noone will be able to transfer them to their own account.

Unlock Period

After the retention period has ended, team members and advisors are allowed to unlock their token share, thus triggering the transfer to their own accounts.

Destruction

When 365 days after the contract instance creation have passed, the owner is allowed to destroy this contract instance.

All remaining tokens of this contract instance will be transferred to the owner's account. Team members and advisors who have not unlocked their tokens share will lose it.

2.6.5 Constraints

Allocation

The amount of allocated tokens can be set for every team member or advisor account only once.

The total amount of allocated tokens must not exceed the predefined cap (see *Constants*).

Total Supply

The predefined cap of allocated tokens `totalTeamAndAdvisorsAllocation` must not be greater than the amount of initially minted tokens `VESTED_TEAM_ADVISORS_SHARE`, otherwise it would be possible to allocate more tokens than available, i.e. some team members won't be able to unlock their share.

Time Periods

Constraints regarding time are already described in *Lifecycle*.

CHAPTER

THREE

CODE REVIEW

An essential part of an audit is the code review.

The aim is to identify possible issues that may constitute a risk to parts of or the whole project by investigating carefully the code from its overall global dependency structure down to single lines of code.

Note: There was no code review of *OpenZeppelin*'s *Solidity* framework itself as it is supposed to have been thoroughly audited already.

3.1 Methodology

Issues were categorized in one of three classes:

Major issues are real show stoppers and must be fixed before production deployment. They jeopardize the project due to risk of loosing funds or driving the contracts unusable.

Minor issues are rather annoyances, but don't compromise the project. They may make a non-critical function useless or decrease the user experience.

Notes are no issues. Superfluous operations, missing events, or wrong values in otherwise unused variables fall into this category.

To categorize found issues the following risk matrix is used where the columns represent probability, and rows represent severity of an issue.

Table 3.1: Risk Matrix

	No Impact	Negligible	Marginal	Critical	Catastrophic
Certain	Note	Minor	Major	Major	Major
Likely	Note	Minor	Minor	Major	Major
Possibly	Note	Note	Minor	Major	Major
Unlikely	Note	Note	Minor	Minor	Major
Rare	Note	Note	Note	Minor	Major

3.2 Results

No major or minor issues were found.

The source code is easy to read, its structure is easy to follow, and its intention is easy to grasp.

Nearly all written code conforms *Solidity*'s style guide. All sources – with the exception of *Whitelist*, which is a very simple and self-explaining contract – are well documented.

All arithmetic operations are carried out by the well-known *SafeMath* library. There are no deep nested code dependency trees and no intertwined code paths.

The extensive usage of *OpenZeppelin*'s *Solidity* framework is a good practice to increase the overall code robustness by reusing a well-established smart contract code base without reinventing the wheel.

3.3 Known Attacks

3.3.1 Reentrancy attack

This attack consists on recursively calling the `call.value()` method in an *ERC20* token to extract the ether stored on the contract if the user is not updating the balance of the sender before sending the ether.

See: more information on reentry

Not affected as there are no external calls to unknown addresses.

3.3.2 Overflows attacks

An overflow happens when the result of an arithmetic operation on a variable exceeds the variable's type limit.

See: more information on overflow

Not affected, as all arithmetic operations are carried by the SafeMath library.

3.3.3 Short address attack

This attack is based on providing a too short (less than 20 bytes) address when calling a contract function and thus shifting subsequent parameters to the left.

See: more information on short address

Not affected, as all third-party callable functions either don't have address parameters or no other parameters beside a single address.

3.4 Source Whitelist.sol

3.4.1 Major issues

None

3.4.2 Minor issues

None

3.4.3 Notes

Functions *addToWhitelist(...)* and *removeFromWhitelist(...)*

Propability	Certain
Impact	No Impact

The functions `addToWhitelist` and `removeFromWhitelist` don't pay attention to if a given account address was already whitelisted or not.

This has no impact on the desired functionality – providing a mutable set of whitelisted addresses – but adds log entries for the event `WhitelistUpdated` even when an update didn't happen actually.

Proposal: Add an additional test and ignore addresses that were/weren't already whitelisted.

```
10 function addToWhitelist(address[] _addresses) public onlyOwner {
11     for (uint256 i = 0; i < _addresses.length; i++) {
12         if (!allowedAddresses[_addresses[i]]) {
13             allowedAddresses[_addresses[i]] = true;
14             WhitelistUpdated(now, "Added", _addresses[i]);
15         }
16     }
17 }
18
19 function removeFromWhitelist(address[] _addresses) public onlyOwner {
20     for (uint256 i = 0; i < _addresses.length; i++) {
21         if (allowedAddresses[_addresses[i]]) {
22             allowedAddresses[_addresses[i]] = false;
23             WhitelistUpdated(now, "Removed", _addresses[i]);
24         }
25     }
26 }
```

The proposed code may slightly increase the gas usage when executed due to the additional state variable access.

Final thoughts: These functions are called by the *Whitelist* administrator only who may track off-chain which investors are already whitelisted, or do a prior check by calling `isWhitelisted`.

3.4.4 Coding style

Doc comments

There are no comments at all.

3.5 Source ODEMCrowdsale.sol

3.5.1 Major issues

None

3.5.2 Minor issues

None

3.5.3 Notes

Function *buyToken(...)*

Probability	Possible
Severity	Negligible

The mapping `trackBuyersPurchases` will contain a wrong value for the remainder investor (whose purchase will make the total token supply exceed the cap), as he purchased less tokens than noted there.

Proposal: Put the update of `trackBuyersPurchases` after remainder logic.

```
138     //remainder logic
139     if (token.totalSupply().add(tokens) > TOTAL_TOKENS_FOR_CROWDSALE) {
140         tokens = TOTAL_TOKENS_FOR_CROWDSALE.sub(token.totalSupply());
141         weiAmount = tokens.div(rate);
142
143         // save info so as to refund purchaser after crowdsale's end
144         remainderPurchaser = msg.sender;
145         remainderAmount = msg.value.sub(weiAmount);
146     }
147
148     trackBuyersPurchases[beneficiary] = trackBuyersPurchases[beneficiary].add(tokens);
```

Function *buyToken(...)*

Probability	Possible
Severity	Negligible

If the amount of remaining tokens (`TOTAL_TOKENS_FOR_CROWDSALE - token.totalSupply`) is not a multiple of `rate`, the remainder investor (whose purchase will make the total token supply exceed the cap) will receive a tiny amount of tokens more than paid for.

It must be pointed out, that the worth of these additional tokens is less than 1 wei (= 10:sup:*-18* ether).

Simplified Example:

Assumptions

- rate is 10 tokens per wei
- remaining amount is 26 tokens
- investor wants to buy for 5 wei

1. Usually, the investor would receive 5 wei × 10 = 50 tokens.
2. As this would exceed the total cap, he merely receives the remaining 26 tokens.
3. Their worth in wei is calculated by integer division: int(26/10) = 2 wei.
4. After crowdsale end, the investor will be refunded: 5 wei - 2 wei = 3 wei.

The investor actually paid 2 wei and received 26 tokens, that is 6 more than he paid for.

Proposal: Calculate the correct amount of bought tokens by

```
141     //remainder logic
142     if (token.totalSupply().add(tokens) > TOTAL_TOKENS_FOR_CROWDSALE) {
143         weiAmount = TOTAL_TOKENS_FOR_CROWDSALE.sub(token.totalSupply()).div(rate);
144         tokens = weiAmount.mul(rate);
145
146         // save info so as to refund purchaser after crowdsale's end
147         remainderPurchaser = msg.sender;
148         remainderAmount = msg.value.sub(weiAmount);
149     }
```

and account for remaining token amount being below rate in

```
161 function hasEnded() public view returns (bool) {
162     if (token.totalSupply() > TOTAL_TOKENS_FOR_CROWDSALE - rate) {
163         return true;
164     }
```

Final thoughts: Leave the code as it is, as the additional gas cost of an exact solution outreaches the wasted token's worth by several orders of magnitude.

Function *hasEnded()*

Probability	Certain
Severity	No Impact

The test for if the crowdsale has ended, i.e. is ready to be finalized, is implemented in a way that it first performs an access to an external state variable `token.totalSupply` and then accesses internal state via `super.hasEnded()`.

Provided that fulfillment of both test cases are equally probable (in a lazy evaluation environment) one could gain a tiny reduction in gas usage by first doing the cheaper test.

Proposal: Simplify and reorder the tests. Lazy evaluation of binary *OR* operator won't test the second condition if the first is met already.

```
161 function hasEnded() public view returns (bool) {
162     return super.hasEnded() || token.totalSupply() == TOTAL_TOKENS_FOR_CROWDSALE;
163 }
```

Final thoughts: As this function is likely to be called only once when `ODEMCrowdsale`'s owner attempts to finalize the crowdsale, the gas reduction is negligible, and the code readability is high already.

Function *finalization()*

Probability	Certain
Severity	No Impact

The external state variable `token.totalSupply()` is accessed twice although its value doesn't change between those calls.

To reduce gas cost consider buffering the value in a local variable first.

Proposal: Retrieve the total tokens supply only once.

```
179 function finalization() internal {
180     // This must have been set manually prior to finalize().
181     require(teamAndAdvisorsAllocation != address(0x0));
182
183     // final minting
184     token.mint(teamAndAdvisorsAllocation, VESTED_TEAM_ADVISORS_SHARE);
185     token.mint(wallet, NON_VESTED_TEAM_ADVISORS_SHARE);
186     token.mint(wallet, COMPANY_SHARE);
187     token.mint(rewardWallet, BOUNTY_REWARD_SHARE);
188
189     uint totalSupply = token.totalSupply();
190
191     if (TOTAL_TOKENS_SUPPLY > totalSupply) {
192         uint256 remainingTokens = TOTAL_TOKENS_SUPPLY.sub(totalSupply);
193
194         token.mint(wallet, remainingTokens);
```

```
195     }
196
197     token.finishMinting();
198     ODEMToken(token).unpause();
199     super.finalization();
200 }
```

Final thoughts: This function is called only once by `ODEMCrowdsale`'s owner. The gas reduction is negligible, and the code readability is high already,

3.5.4 Coding style

Constructor

For long function declarations, it is recommended to drop each argument onto its own line at the same indentation level as the function body. The closing parenthesis and opening bracket should be placed on their own line as well at the same indentation level as the function declaration.

```
52 function ODEMCrowdsale(
53     uint256 _startTime,
54     uint256 _endTime,
55     address _whitelist,
56     uint256 _rate,
57     address _wallet,
58     address _rewardWallet
59 )
60     public
61     FinalizableCrowdsale()
62     Crowdsale(_startTime, _endTime, _rate, _wallet)
63 {
```

Order of functions

Functions should be grouped according to their visibility and ordered

- constructor
- fallback function (if exists)
- external
- public
- internal
- private

Proposal: The public function `setTeamWalletAddress(. . . )` should appear before the internal function `createTokenContract()`.

Function *buyToken(. . .)*

The `buyTokens` function ensures that token purchaser and beneficiary are the same:

```
126 require(msg.sender == beneficiary);
```

In function body code both variables are used interchangeably.

Proposal: To increase code readability one should consider using only one of

- `msg.sender`

- `beneficiary`

within function code.

Function *setTeamWalletAddress(...)*

Different namings for the same thing should be avoided:

- `setTeamWalletAddress`
- `teamAndAdvisorsAllocation`

A single terminus, either “team wallet” or “team and advisors allocation” would increase code readability

Proposal: Rename this function to `setTeamAndAdvisorsAllocationAddress(...)`.

Doc comments

The heading comment of function `hasEnded()` is not a doc comment.

3.6 Source ODEMToken.sol

3.6.1 No Need for Code Review

The token contract implementation is completely done by *OpenZeppelin*’s base contracts

- `PausableToken`
- `MintableToken`

There are no custom code extensions besides definition of public constants

```
12 string public constant name = "ODEM Token";
13 string public constant symbol = "ODEM";
14 uint8 public constant decimals = 18;
```

As the token contract instance is meant to be created and owned by an `ODEMCrowdsale` instance during its whole lifetime, and there’s no mechanism implemented in the latter to transfer the ownership, the `ODEMToken`’s behavior regarding minting and pausing is fully controlled by the `ODEMCrowdsale` instance.

3.7 Source TeamAndAdvisorsAllocation.sol

3.7.1 Major Issues

None

3.7.2 Minor Issues

None

3.7.3 Notes

Function *addTeamAndAdvisorsAllocation(...)*

Probability	Rare
Severity	Marginal

The token allocation for a single team member can be set only once. But as long as the total amount of allocated tokens `allocatedTokens` is below the cap `totalTeamAndAdvisorsAllocation`, a team members allocation can be set again after he/she unlocked his/her share – again only once until next unlock, etc.

```
45 assert(teamAndAdvisorsAllocations[teamOrAdvisorsAddress] == 0); // can only add once.
```

The only purpose imaginable for this constraints is to increase trust of team members, i.e. that the contract owner is not able to change one's allocation once set.

Proposal: If trust is an issue here, an event should be added, too:

```
event TeamAndAdvisorAllocationAdded(address indexed beneficiary,
                                    uint256 amount);
```

Final thoughts: As team members and advisors have to trust the contracts's owner anyway, the mentioned constraint can be removed.

Function *unlock()*

Probability	Unlikely
Severity	Negligible

The only way for the contract owner to know if all team members have unlocked their shares is either:

- to check if

  ```
    ODEMCrowdsale.VESTED_TEAM_ADVISORS_SHARE
  - ODEMToken.balanceOf(TeamAndAdvisorsAllocation)
  = TeamAndAdvisorsAllocation.allocatedTokens
  ```

  which won't work if some tokens where transfered to the `TeamAndAdvisorsAllocation` instance by a third party,
- to keep an off-chain list of all team members or look into the transaction history and iterate over `teamAndAdvisorsAllocations`.

Prior to killing the contract it may be desirable to know if there are still some shares left.

Proposal: Introduce a public state variable tracking the total amount of allocated tokens which weren't unlocked by team members.

3.7.4 Coding style

Assert vs require

Use `require` to validate state conditions prior to executing state changing operations, and `assert` to validate contract state after making changes.

Proposal Use `require` instead `assert` at the beginning of these functions:

```
40 function addTeamAndAdvisorsAllocation(address teamOrAdvisorsAddress, uint256 allocationValue)
41     external
42     onlyOwner
43     returns(bool)
44 {
45     require(teamAndAdvisorsAllocations[teamOrAdvisorsAddress] == 0); // can only add once.
```

```
58 function unlock() external {
59     require(now >= unlockedAt);
```

```
76 function kill() public onlyOwner {
77     require(now >= canSelfDestruct);
```

Contract Naming

Naming a function `addTeamAndAdvisorsAllocation` which adds an entry to the global mapping `teamAndAdvisorsAllocations` is fine by itself as it implies a "team and advisor allocation" to refer to a single entry.

But thus the contract name is misleading.

Proposal: Rename the contract to plural form by adding an "s":

```
11 contract TeamAndAdvisorsAllocations is Ownable {
```

Constant *totalTeamAndAdvisorsAllocation*

The state variable `totalTeamAndAdvisorsAllocation` looks like a constant.

Proposal: Consider renaming it and placing it above state variables.

```
14 uint256 private constant TOTAL_TEAM_AND_ADVISORS_ALLOCATION = 38763636e18; // 38 mm
15
16 uint256 public unlockedAt;
17 uint256 public canSelfDestruct;
18 uint256 public tokensCreated;
19 uint256 public allocatedTokens;
```

Variable *canSelfDestruct*

The state variable name `canSelfDestruct` implies some boolean term. Actually its a timestamp.

Proposal: Consider renaming it to something else, e.g. `destructibleAt` or `destructibleAfter`.

Function *addTeamAndAdvisorsAllocation(...)*

Unnecessary return value. A call to this function will always return `true` if not rejected.

CHAPTER

FOUR

TESTING

In order to reveal issues in the contracts code which would have been overseen otherwise, some tests on the Ethereum Virtual Machine (EVM) were performed.

Note: There were no tests performed on the *OpenZeppelin Solidity* framework itself as it is supposed to have been thoroughly tested already.

4.1 Methodology

The identification and developing of test scenarios involved several steps:

1. All involved accounts and their roles had to be identified.
2. The capabilities of these roles within different periods along the project's life cycle were identified.
3. Invariants within the different life cycle periods had to be identified.
4. Based on the steps above test cases were drafted. Additional attention was paid to the actual implementation and meaningful constant values.
5. Automated tests had to be written (where not already done).
6. Manual tests and automated tests were run on different test environments.

4.2 Results

All performed tests were successful, i.e. their results were as expected.

4.3 Involved Roles/Accounts

The following roles where identified:

Whitelist:

- contract account of `Whitelist` instance

Whitelist owner:

- a regular account
- initially the deployer of `Whitelist` but this can change due to `transferOwnership`

Crowdsale:

- contract account of `ODEMCrowdsale` instance

Crowdsale owner:

- a regular account
- initially the deployer of `ODEMCrowdsale` but this can change due to `transferOwnership`

Private investor:

- multiple accounts which receive tokens during pre-crowdsale period
- regular accounts while testing

Regular investor / Whitelisted Investor:

- multiple accounts which were added to `Whitelist`
- buy tokens during crowdsale
- regular accounts while testing

Company Wallet:

- contract account of a multisig wallet
- a reguar account while testing

Reward Wallet:

- contract account of a wallet receiving bounty tokens
- a reguar account while testing

Token:

- contract account of `ODEMToken` instance

Token Owner:

- same as *Crowdsale* while testing crowdsale (and when in production)
- a regular account for simple isolated tests

Token Holder:

- multiple accounts while testing
- have a non-zero token balance
- **includes**
  - *Private investor* (due to pre-crowdsale minting)
  - *Regular investor* (due to token purchase during crowdsale)
  - *Company wallet* (due to received share on crowdsale finalization)
  - *Reward wallet* (due to received share on crowdsale finalization)
  - *TeamAndAdvisorsAllocation* (due to received share on crowdsale finalization)
  - *TeamAndAdvisorsAllocation owner* (due to destruction of `TeamAndAdvisorsAllocation`)
  - *Team member* (due to token share allocation after 182 days retention period)
  - other accounts (due to token trades/transfers after crowdsale finalzation)

TeamAndAdvisorsAllocation:

- contract account of `TeamAndAdvisorsAllocation`

TeamAndAdvisorsAllocation owner:

- a regular account
- initially the deployer of `TeamAndAdvisorsAllocation` but this can change due to `transferOwnership`

Team member:

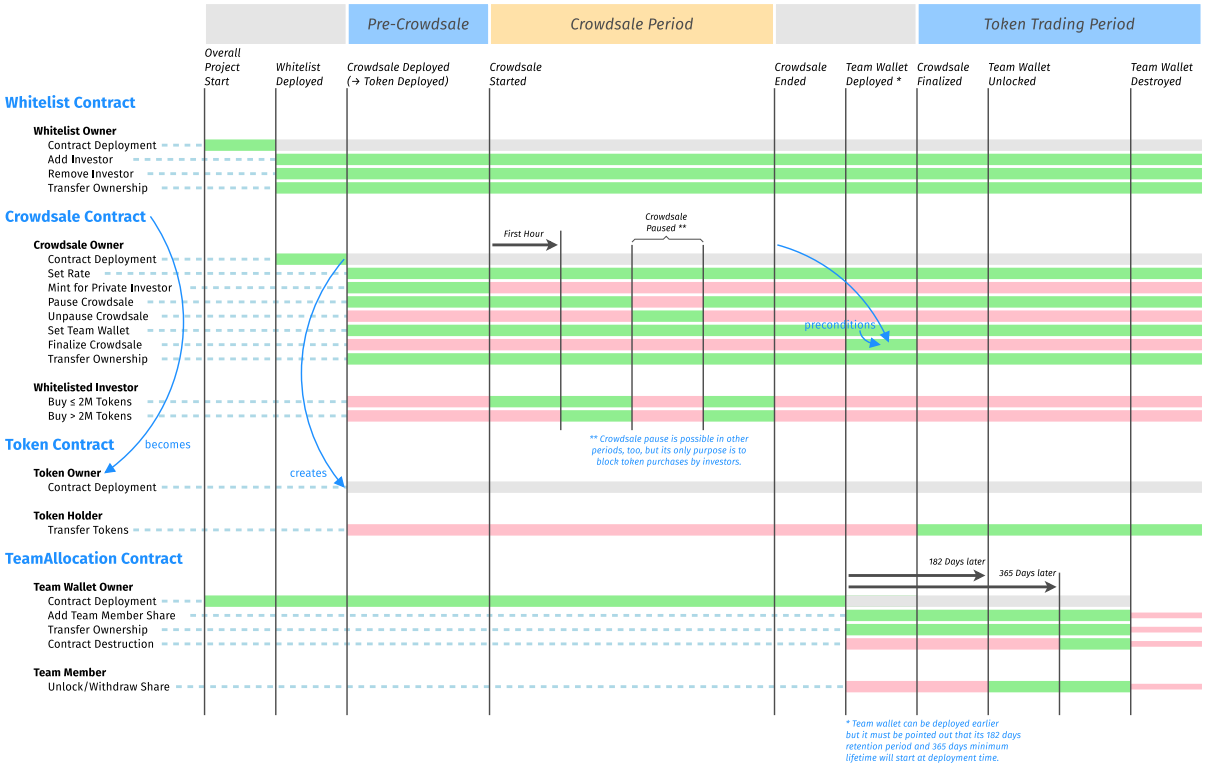
- multiple regular accounts while testing
- were added to `TeamAndAdvisorsAllocation` with a non-zero share

Third party:

- any unprivileged account
- context depends on specific test

4.4 Lifecycle

The following diagram gives a brief overview of the allowed/forbidden actions the participating roles may (green bars) or may not (red bars) perform over the time.



Capabilities of participants over time

4.5 Test Cases Whitelist

4.5.1 Deployment

Test: Deployment of Whitelist

Rationale: A deployed `Whitelist` is needed prior to the deployment of `ODEMCrowdsale`.

Expected behavior:

- Success.
- Deployer account becomes whitelist owner.

4.5.2 Adding Addresses

Test: Whitelist owner adds addresses

This test was carried as multiple sub-tests with:

1. a single address that wasn't whitelisted before
2. a single address that was whitelisted already
3. ten addresses at once, where the address list contains double entries and invalid addresses (`address(0)`)

Rationale: The *Whitelist* implementation is agnostic regarding submitted addresses.

Expected behavior:

- Success.
- For every submitted item in address list an event `WhitelistUpdated` with `operation = "Added"` is logged.
- For every submitted address `addr` the function call `isWhitelisted(addr)` returns `true`.

Test: A third party (non-owner) adds an address

Rationale: Only the whitelist owner should be able to add addresses.

Expected behavior:

- Failure / Transaction reversal.

4.5.3 Removing Addresses

Test: Whitelist owner removes addresses

This test was carried out as multiple sub-tests with:

1. a single address that is whitelisted
2. a single address that wasn't whitelisted before
3. ten addresses at once, where the address list contains double entries and invalid addresses (`address(0)`)

Rationale: The *Whitelist* implementation is agnostic regarding submitted addresses.

Expected behavior:

- Success.
- For every submitted item in address list an event `WhitelistUpdated` with `operation = "Removed"` is logged.
- For every submitted address `addr` the function call `isWhitelisted(addr)` returns `false`.

Test: A third party (non-owner) removes an address

Rationale: Only the whitelist owner should be able to remove addresses.

Expected behavior:

- Failure / Transaction reversal.

4.6 Test Cases ODEMCrowdsale

4.6.1 Deployment

Test: Deployment of ODEMCrowdsale with invalid parameters

This test was carried out as multiple sub-tests where:

1. `_startTime < now`
2. `_startTime > _endTime`
3. `_whitelist = address(0)`
4. `_rate = 0`
5. `_wallet = address(0)`
6. `_rewardWallet = address(0)`

Rationale: These are base constraints on the given values for a proper functionality of the crowdsale.

Expected behavior:

- Failure / Transaction reversal.

Test: Deployment of ODEMCrowdsale with valid parameters

Rationale: Contract `ODEMCrowdsale` must be deployable.

Expected behavior:

- Success.
- Deployer account becomes crowdsale owner.
- The following constructor parameters set state variables to given values
  - `_startTime` → `startTime`
  - `_endTime` → `endTime`
  - `_whitelist` → `whitelist`
  - `_rate` → `rate`
  - `_wallet` → `wallet`
  - `_rewardWallet` → `rewardWallet`
- The state variable `oneHourAfterStartTime` is correctly calculated as `startTime + 60 * 60`
- An `ODEMToken` instance was created and assigned to state variable `token`:
  - the attribute `token.name` should be `"ODEM Token"`
  - the token owner is the crowdsale contract account
  - the token is paused, i.e. `token.paused` is set to `true`

Test: Constants sanity check

Rationale: The defined constants in `ODEMCrowdsale` impose restrictions on the amount of mintable/available tokens. The following conditions are not explicitly checked within the code, but rely on meaningful hard-coded values:

- `PRE_CROWDSALE_CAP <= TOTAL_TOKENS_FOR_CROWDSALE`
- `TOTAL_TOKENS_FOR_CROWDSALE <= TOTAL_TOKENS_SUPPLY`

- ```
  TOTAL_TOKENS_SUPPLY >= BOUNTY_REWARD_SHARE
                       + VESTED_TEAM_ADVISORS_SHARE
                       + NON_VESTED_TEAM_ADVISORS_SHARE
                       + COMPANY_SHARE
                       + TOTAL_TOKENS_FOR_CROWDSALE
  ```

Expected behavior:

- The constraints are fulfilled.

4.6.2 Explicit state value changes

Test: Crowdsale owner sets a valid rate

Rationale: The rate can be changed at any time as long at is greater than 0.

Expected behavior:

- Success.
- The crowdsale state variable `rate` is properly adjusted.
- An event `TokenRateChanged` with correct values for old and new rate is logged.

Test: Crowdsale owner sets rate to zero

Rationale: An invariant is sol:*rate > 0*.

Expected behavior:

- Failure / transaction reversal.

Test: A third party (non-owner) changes rate

Rationale: Only the crowdsale owner should be able to change the rate.

Expected behavior:

- Failure / transaction reversal.

Test: Crowdsale owner sets a valid teamAndAdvisorsAllocation address

Rationale: The teamAndAdvisorsAllocation address can be changed at any time.

Expected behavior:

- Success.
- The crowdsale state variable `teamAndAdvisorsAllocation` is properly adjusted.

Test: Crowdsale owner sets teamAndAdvisorsAllocation to zero address

Rationale: The supplied address must be non-zero.

Expected behavior:

- Failure / transaction reversal.
  ```

Test: A third party (non-owner) sets a valid teamAndAdvisorsAllocation address

Rationale: Only the crowdsale owner should be able to set the team wallet address.

Expected behavior:

- Failure / transaction reversal.

4.6.3 During Pre-Crowdsale Period

Test: Invariants

Rationale: During pre-crowdsale the following condition must hold:

- *token.totalSupply <= PRE_CROWDSALE_CAP*

Test: Crowdsale owner mints a valid amount of tokens for a private investor

Rationale: Before crowdsale starts private investors get their token share minted by owner.

Expected behavior:

- Success.
- The investors token balance gets increased by the correct amount.
- The total token supply is increased accordingly.
- An event `PrivateInvestorTokenPurchase` with correct token amount is logged.

Test: Crowdsale owner mints an invalid amount of tokens for a private investor

Rationale: The total amount of mintable tokens during pre-crowdsale is capped by constant `PRE_CROWDSALE_CAP`.

Expected behavior:

- Failure / transaction reversal.

Test: A third party (non-owner) mints tokens for a private investor

Rationale: Only crowdsale owner should be able to mint tokens during pre-crowdsale period.

Expected behavior:

- Failure / transaction reversal.

4.6.4 During Crowdsale Period

Test: Invariants

Rationale: During pre-crowdsale the following condition must hold:

- *token.totalSupply <= TOTAL_TOKENS_FOR_CROWDSALE*

Test: Crowdsale owner mints a valid amount of tokens for a private investor

Rationale: After the crowdsale has started the owner should not be able anymore to mint tokens for private investors.

Expected behavior:

- Failure / transaction reversal.

Test: A whitelisted investor buys a valid amount of tokens

Rationale: Any account that was whitelisted should be able to purchase tokens as long as the individual first hour cap or the total token cap won't be exceeded.

Expected behavior:

- Success.
- The token amount is `rate * msg.value`.
- The investor's entry in `trackBuyersPurchase` is increased by the correct token amount.
- The state variable `weiRaised` is increased by sent wei value.
- The sent wei are forwarded to the company wallet.
- The investor's token balance gets increased by the correct token amount.
- The total token supply is increased accordingly.
- An event `TokenPurchase` with correct token amount and value is logged.

Test: A whitelisted investor buys too many tokens within the first hour

Rationale: During the first hour after crowdsale start the individual total amount of tokens an investor can buy is limited by constant `PERSONAL_FIRST_HOUR_CAP`. This cap doesn't take tokens into account which were received during pre-crowdsale.

Expected behavior:

- Failure / transaction reversal.

Test: A whitelisted investor buys tokens while crowdsale is paused

Rationale: The crowdsale owner can pause and unpause the crowdsale. During pause nobody should be able to purchase tokens, even if the purchaser was whitelisted and no cap would be exceeded.

Expected behavior:

- Failure / transaction reversal.

Test: A whitelisted investor buys tokens for the benefit of another account

Rationale: The crowdsale definition in *OpenZeppelin*'s *Solidity* framework allows a token purchaser to be different from the beneficiary. In `ODEMCrowdsale` purchaser and beneficiary must be the same account.

Expected behavior:

- Failure / transaction reversal.

Test: A whitelisted investor buys tokens so that the total cap is exceeded

Rationale: The total amount of tokens minted during pre-crowdsale and crowdsale is capped by the constant `TOTAL_TOKENS_FOR_CROWDSALE`. If a purchase exceeds this limit the purchaser receives the remaining tokens. His/Her account and overspent wei value will be safed for later refund.

Expected behavior:

- Success.
- The received amount token are the remaining, i.e. difference of `TOTAL_TOKENS_FOR_CROWDSALE - token.totalSupply`, their worth in wei is remaing tokens divided by `rate`.
- The state variable `weiRaised` is increased by wei worth of remaining tokens.
- The sent wei are forwarded to the company wallet.
- The investor's token balance gets increased by the received token amount.
- The total token supply is increased accordingly and should be equal to `TOTAL_TOKENS_FOR_CROWDSALE`.
- An event `TokenPurchase` with received token amount and worth in wei logged.
- The crowdsale ends early, i.e. `hasEnded()` becomes true.

Test: A third party (non-whitelisted-investor) buys tokens

Rationale: Prior to be able to purchase tokens investors have to be whitelisted.

Expected behavior:

- Failure / transaction reversal.

4.6.5 After Crowdsale Period

Test: Invariants

Rationale: After crowdsale the following condition must hold:

- *token.totalSupply == TOTAL_TOKENS_SUPPLY*

Test: A whitelisted investor buys tokens

Rationale: After the crowdsale has ended nobody should be able to buy tokens anymore, even if the investor is whitelisted or the total token cap was not reached, i.e. the crowdsale ended due to `endTime`.

Expected behavior:

- Failure / transaction reversal.

Test: Crowdsale owner finalizes without having set a team wallet address

Rationale: Prior to finalization the address of a `TeamAndAdvisorAllocation` contract instance has to be saved.

Expected behavior:

- Failure / transaction reversal.

Test: Crowdsale owner finalizes after having set a team wallet address

Rationale: The finalization of crowdsale is a crucial step to make the token tradeable/transferrable.

Expected behavior:

- Success.
- Token balance of teamAndAdvisorsAllocation is increased by `VESTED_TEAM_ADVISORS_SHARE`.
- Token balance of company wallet is increased by `NON_VESTED_TEAM_ADVISORS_SHARE + COMPANY_SHARE` plus the remaining total amount of tokens (`TOTAL_TOKENS_SUPPLY - token.totalSupply`).
- Token balance of reward wallet is increased by `BOUNTY_REWARD_SHARE`.
- The token total supply is `TOTAL_TOKENS_SUPPLY`.
- The token minting is finished, i.e. not possible anymore.
- The token is unpaused, i.e. is tradeable.
- An event `Finalized` (for the crowdsale) and an event `Unpause` (for the token) are logged.

Test: Crowdsale owner finalizes a again

Rationale: A finalized crowdsale should not be finalizable anymore.

Expected behavior:

- Failure / transaction reversal.

4.7 Test Cases ODEMToken

4.7.1 Deployment

Test: Deployment of ODEMToken

Rationale: The token contract will be deployed automatically by `ODEMCrowdsale` on initialization.

Expected behavior:

- Success.
- Deployer account (crowdsale contract) becomes token owner.
- The global state variables `name`, `symbol`, `decimals` are correctly set.

4.8 Test Cases TeamAndAdvisorsAllocation

4.8.1 Deployment

Test: Deployment of TeamAndAvisorsAllocation

Rationale: Contract `TeamAndAvisorsAllocation` must be deployable.

Expected behavior:

- Success.
- Deployer account becomes contract instance owner.
- The state variable `odem` is set to the constructor parameters value `token`.

- The state variable `unlockedAt` is correctly calculated as `now + 182 days` (`= now + 182 * 24 * 60 * 60`).
- The state variable `canSelfDestruct` is correctly calculated as `now + 365 days` (`= now + 365 * 24 * 60 * 60`).

Test: Constants sanity check

Rationale: The constants `totalTeamAndAdvisorsAllocation` constitutes a cap on the amount of tokens that can be assigned by team members. It should not be greater than the amount of tokens that get minted for this contract on crowdsale finalization (`ODEMCrowdsale.VESTED_TEAM_ADVISORS_SHARE`), otherwise some team members won't be able to unlock their share.

Expected behavior:

- The constraint is fulfilled.

4.8.2 Assigning Allocations

Test: Owner adds a team member's share

Rationale: The team member's share gets saved.

Expected behavior:

- Success.
- The total number of allocated tokens is increased by the share amount.
- The team member's share is correctly added to the state variable `teamAndAdvisorsAllocations`.

Test: Owner changes a team member's share before it was unlocked

Rationale: The share of a team member (an account) can be set only once until it gets unlocked and thus withdrawn by the team member.

Expected behavior:

- Failure / transaction reversal.

Test: Owner adds a team member's share so that the total cap is exceeded

Rationale: The total amount of allocated tokens must not be greater than `totalTeamAndAdvisorsAllocation`.

Expected behavior:

- Failure / transaction reversal.

Test: A third party (non-owner) adds an allocation

Rationale: Only the owner should be able to add allocations.

Expected behavior:

- Failure / transaction reversal.

4.8.3 Unlocking token share

Test: Team member unlocks his/her share within retention period

Rationale: Nobody should be able to unlock his/her share during the first 182 days after contract deployment.

Expected behavior:

- Failure / transaction reversal.

Test: Team member unlocks his/her share after retention period

Rationale: Team members get their token share by unlocking it, so that it gets credited to their token balance.

Expected behavior:

- Success.
- The total amount of tokens possessed by the contract is saved to `tokensCreated` if this was the first call to `unlock()` by anyone.
- Token balance of *TeamAndAdvisorsAllocation* contract is reduced by the team member's share.
- Token balance of team member is increased by his/her share.
- The allocation of the team member is set to zero.

Test: Team member unlocks his/her share after contract was killed

Rationale: After killing the contract no unlocking of token share should be possible.

Expected behavior:

- Failure / transaction reversal.

Test: A third party (non-team-member) unlocks

Rationale: Non-team-members have no token share, i.e. their allocation is zero. If they attempt to unlock they'll waste gas.

Test:

- Success.
- The total amount of tokens possessed by the contract is saved to `tokensCreated` if this was the first call to `unlock()` by anyone.
- An amount of zero tokens will be transferred / added to the caller's balance.

4.8.4 Contract Destruction

Test: Owner kills contract within the first 365 days after deployment

Rationale: The `TeamAndAdvisorsAllocation` contract instance has to be accessible for at least one year.

Expected behavior:

- Failure / transaction reversal.

Test: Owner kills contract 365 days or more after deployment

Rationale: Destroying the `TeamAndAdvisorsAllocation` contract will make all team members lose their not-yet unlocked token share.

Expected behavior:

- All remaining tokens of the contract (including those which weren't unlocked by team members) get transferred to the owner, thus increasing his/her token balance.
- Token balance of `TeamAndAdvisorsAllocation` instance is zero.
- The contract gets destroyed (code data of account is set to `0x0`).

4.9 Test Environments

The *ODEM ICO* contracts were tested on different networks.

1. Deployment on the Rinkeby test network.

   The testing was carried out by manually calling the contract functions via MyEtherWallet and MetaMask from different accounts.

   Analysis of transaction history and contract state were done via MyEtherWallet and Etherscan.

2. Deployment on a local machine.

   Automated tests for the Truffle (v4.0.5) test framework were written and executed on a local Ganache-CLI (v6.0.3) instance.

3. Deployment on a local test network.

   The above mentioned automated tests and some additional manual tests were carried out on a Parity (v1.8.6) development network.

   Analysis of transaction history and contract state was done via Parity's frontend.

CHAPTER

FIVE

CONCLUSION

After in-depth reading through the *ODEM* smart contract project code, intensive conversations with the customers and developers about the goals and the actual implementation of this project, extensive testing and attendance of source code corrections, no issues were found that would hinder the *ODEM* Crowdsale from being deployed on the Ethereum main network and thus going productive.