# Etherisc Token and Crowdsale Audit Report

Validity Labs AG

1 October 2017

## 1 Introduction

We reviewed the Etherisc token and crowdsale smart contracts that were published on github with the commit hash 173cc61d936c8f2c0859e9f5cdbdbb6b29b251d6. Etherisc provided an ERC20-compatible standard token and a corresponding crowdsale contract that are based on the OpenZeppelin solidity library. The crowdsale has a few special features extending Zeppelin's `Crowdsale` and `FinalizableCrowdsale`:

- three-round crowdsale
- owner-controlled whitelisting of accounts and contribution amounts per round for the first two rounds of the crowdsale
- function to retrieve tokens that have accidentally been sent to the crowdsale and token contract

Additionally, the unsold tokens should be made available to a multisig timelock contract that releases tokens according to a pre-defined vesting schedule.

Etherisc provided extensive truffle unit and integration tests as well as a code coverage assessment detailing a 100% code coverage. Generally, the code follows best practices and established smart contract development guidelines. The code is structured in a readable fashion and extends the Zeppelin contracts only where needed. We have found several non-critical issues outlined below and we recommend to address all of them before deploying the contracts.

## 2 Issues

### 2.1 `salvageTokens` of token contract not callable

The `salvageTokens` of the `DipToken` contract is only callable by its owner. This token contract gets deployed in the constructor of the `DipTge` contract. Therefore, the owner of the `DipToken` contract is the `DipTge` contract which does not implement a call to the token's `salvageTokens` function. Thus `DipToken.salvageTokens` can never be called successfully by anyone.

It turns out that this function is tested in a truffle test that works against a separately deployed token that therefore can be accessed by the externally owned account that created it. We suggest to also implement an integration test that tests against a token that has been deployed by the crowdsale contract.

### 2.2 Reaching `minCap` is not enforced and not reaching it has no implications

The `minCap` that is set in the `DipWhitelistedCrowdsale` contract has no meaning. Reaching it is not enforced and not reaching it has no implications. If that was the intended behaviour then we suggest to remove it and reduce gas usage for all investors.

### 2.3 `MinCapReached` will fire continuously after reaching `minCap`

It might be more efficient for secondary tools to consume the `MinCapReached` event if it was just fired once. Currently, it is also fired for all following investmets upon reaching the `minCap`.

### 2.4 Handling of refunds

The publicly accessible alternative investment entrypoint `buyTokens` might look to an investor as if they could buy in from a common wallet (e.g. exchange) and issue tokens onto a separate account (`_beneficiary`). However, in case the investor can only place a fraction of their submitted value, the refund will always be issued to the `msg.sender`. We therefore suggest to send the reimbursed amounts to `_beneficiary` instead of `msg.sender`. For the majority of investors we expect this to be the same address anyway, as most investors will invest via the fallback function.

### 2.5 `isActive` mentioned in specifications is not implemented

The `isActive` field of the `ContributorData` struct is not implemented but apparently also not required.

### 2.6 Increase readability of token name by adding space

The non-functional token name is used by a variety of tools. We suggest to increase readability by adding a space in the `name` property `DecentralizedInsurance`.

### 2.7 Turn contract variables into constants

We suggest to implement the following variables that are anyway unchanged as constants:

- `name`
- `symbol`
- `decimals` (this is commonly defined as `uint8` instead of `uint256`
- `MAXIMUM_SUPPLY`

### 2.8 Extend specifications on available investment amounts

The amount that an individual investor can contribute is currently not well-defined in the specifications. We note the behaviour as outlined below but do not know if this is the anticipated behaviour as the specifications do not contain these maximal contributions:

- pendingStart: 0
- priorityPass: $pa + oa - ca$ (and not exceeding a total contribution of hardCap1)
- openedPriorityPass:

$$\begin{cases} 0, & \text{if } pa = 0 \text{ and } oa = 0 \\ weiRaised + msg.value < hardCap1, & \text{for all other investors} \end{cases}$$

- crowdsale: any amount that is not exceeding a total contribution of `hardCap2`
- any other state: 0

Here, $pa$, $oa$ and $ca$ indicate the `priorityPassAllowance`, `otherAllowance` and `contributionAmount` of a specific investor.

### 2.9 Use explicit `uint256` instead of alias `uint`

Using explicit types instead of aliases increases readability and is generally considered best practice. Thus we recommend using the explicit `uint256` type instead of the alias `uint` which is found in `TokenStake.sol` and `DipWhitelistedCrowdsale.sol`.

### 2.10 Require more recent compiler version

The code requires a minimal Solidity compiler version 0.4.11. We suggest to require a later Solidity version to prevent the code being compiled with an outdated compiler version. Latest compiler versions are e.g. providing optimizer improvements that reduce gas costs for all users.

### 2.11 Clarify specifications on token fraction

The current specification of the token fraction reads `All DIP token values are stored in uint256 as 10E-18 fractions of tokens.` In most environments (e.g. JavaScript) `10E-18` evaluates to $10^{-17}$ which is probably not what the authors intended nor what is implemented in the smart contract.

### 2.12 Distribution of unsold tokens via vesting contract is not enforcable in code

The specifications state `The finalization function mints the remaining tokens and transfers it to a special multiSig address, from which vested tokens are distributed.` However, during deployment, the target `wallet` address could be even an externally owned account as the multisig with vested distribution is not deployed as part of the crowdsale contract - in contrast to, e.g. the token contract.

### 2.13 Simplify `if-revert` logic to the more readable `require`

The `if (!token.mint(_beneficiary, tokens)) revert();` section in `DipWhitelistedCrowdsale` should be refactored to the more readable but identical `require(token.mint(_beneficiary, tokens));` syntax.