Version 0.6 / 04.10.2017

etherisc Token Contract Audit

inacta AG

Eugen Lechner [eugen.lechner@inacta.ch]

inacta
• managing your information

Inhaltsverzeichnis

1. Introduction

Name	etherisc
Web Presence	http://etherisc.com
Name of Token	DIP (Decentralized Insurance Platform Token)
Whitepaper Reference	https://github.com/etherisc/tokensale/blob/develop/specification.md
GitHub Repository / Commit	https://github.com/etherisc/tokensale 7c243536be9bc825038ebc99a0529ebb2967cca7

2. Scope

This is an, 'eyes over' code review only of the contracts' source code. No static or dynamic testing has been done by the reviewing author. Only the Smart Contracts available under GitHub are reviewed. Frameworks like OpenZeppelin are excluded from the audit.

3. Executive Summary

We reviewed the Token Sales Contract and did not find any critical security problems. Our findings included one important and one moderate issue and we suggest four low severity improvements regarding code readability. In the latest version of the contract, all issues have been resolved.

4. Findings

#	Description	Severity	Priority	Resolved
1	Safe math operators are not used continuously (5.3.2, reported at Sep 18, 2017)	low	low	yes
2	Hight cyclomatic complexity of function calculateMaxContribution (5.3.1)	low	low	
3	Lock pragmas to specific compiler version (5.3.7)	low	low	
4	Constant modifier incorrect (5.3.9)	low	low	

5	Missing Input Validation (5.6.6)	medium	medium	
6	No transaction error on illegal sales phase or impossible sale of tokens (5.3.12)	medium	height	

5. Review Report of Token Sale Smart Contract

5.1 General

Use latest Solidity version?
The contract uses the latest stable Solidity version 0.4.11, supported by Truffle Framework.

Use an open bug bounty program?
We recommend to use it, this is good security practice to incentivize security researchers to further review the contract.

5.2 Compliance with ERC20 Standard

DIP Token is an ERC20 Standard Token, based on OpenZeppelin Solidity Framework.

5.3 Compliance with Smart Contract Best Practices

5.3.1 Keep it Small and Modular

The smart contracts are good structured

TokenStake.sol	Generic Token Staking Contract
DipToken.sol	DIP Token
DipWhitelistedCrowdsale.sol	DIP Token Generating Event
DipTge.sol	inherits from DipWhitelistedCrowdsale
TokenTimelock.sol	Generic Token Time Lock
VestedTokens.sol	inherits from TokenTimelock

By using OpenZeppelin libraries, the contract code is very slim and clear. Only the contract logic in DipWhitelistedCrowdsale has become very confusing due to the nesting of if-else. The cyclomatic complexity of function calculateMaxContribution() is 7. We recommend to simplify it, e.g. as follows:

```
function calculateMaxContribution(address _contributor) public constant returns
(uint256) {
    uint256 maxContrib = 0;
    if (crowdsaleState == state.priorityPass) {
      maxContrib = allowanceSum(_contributor).sub(
            contributorList[_contributor].contributionAmount);
      if (maxContrib > hardCap1.sub(weiRaised)) {
        maxContrib = hardCap1.sub(weiRaised);
      }
    } else if (crowdsaleState == state.openedPriorityPass) {
      if (allowanceSum(_contributor) > 0) {
        maxContrib = hardCap1.sub(weiRaised);
      }
    } else if (crowdsaleState == state.crowdsale) {
      maxContrib = hardCap2.sub(weiRaised);
    }

    return maxContrib;
  }

  function allowanceSum(address _contributor) internal constant returns (uint256) {
    return contributorList[_contributor].priorityPassAllowance.add(
          contributorList[_contributor].otherAllowance);
  }
```

Furthermore, there is no need to distinguish between field *priorityPassAllowance* and *otherAllowance* in the code - both are for the priority phase. The fields can be merged and further simplify the code.

5.3.2 Use safe math operators

DIP Token smart contracts use a safe math library from OpenZeppelin Solidity Framework, unfortunately this library is not used continuously. See DipWhitelistedCrowdsale function calculateMaxContribution(…), VestedTokens function grant(…). We recommend using the safe math operations throughout whole contracts.
Update: This comment has already been put back into the code. The SafeMath is now used continuously, with one exception in function calculateMaxContribution:

```
    if (maxContrib > hardCap1 - weiRaised) {
     maxContrib = hardCap1.sub(weiRaised);
   }
```

I recommend to modify it as follows:

```
        if (maxContrib > hardCap1.sub(weiRaised)) {
         maxContrib = hardCap1.sub(weiRaised);
       }
```

5.3.3 Use modifiers for recurring checks

No findings, the Etherisc smart contract has no recurring pre-condition checks.

5.3.4 Use modifier to authenticate owner

Etheris uses the onlyOwner modifier from OpenZeppelin to check whether the sender is the contract owner.

5.3.5 Avoid negated conditions

Be aware that the negated conditions must be avoided because they can cause errors if the condition is complex.

No findings in Etherisc smart contract code.

5.3.6 Avoid external calls when possible

No findings, Etherisc smart contract has no external calls.

5.3.7 Lock pragmas to specific compiler version

Contracts should be deployed with the same compiler version and flags that they have been tested the most with. Locking the pragma helps ensure that contracts do not accidentally get deployed using, for example, the latest compiler which may have higher risks of undiscovered bugs.

Etherisc does’t do it, we strongly recommend it.

5.3.8 Prefer newer Solidity constructs

No findings in Etherisc smart contract code.

5.3.9 Use constant modifier if possible

Use constant modifier if possible, otherwise consumes these variables or methods unintentionally and unnecessarily state slots and increases gas cost whenever are called.

Function validPurchase() of DipWhitelistedCrowdsale.sol is declared as constant, although it cannot be, since it calls a function that is not constant.

5.3.10 Implements default payable function

This fallback function is implements by OpenZeppelin Crowdsale Smart Contract.

5.3.11 Lifecycle

Etherisc uses a default Lifecycle from OpenZeppelin Crowdsale (start, stop, fi-nalize).

5.3.12 Other findings

The Token buying function *buyTokens()* in DipWhitelistedCrowdsale.sol update the crowdsale state, however does not use this state to check whether the buy-ing possible. So it is still possible to transfer the ether, even if the hard cap has already been reached. In this case, a transaction error must be throw instead of returning the money and the transaction fees will be saved.
The same goes for token sales in the priority phase. When any no PriorityPass member and no selected individual want buy some tokens in the priority phase, it must be checked by require() and a transaction error must be throw for revert any changes.

5.4 Compliance with Token Sale Terms and Conditions

The conditions controlling the Token Sale reflect the Token Sale Specification.

5.5 Compliance with Code Style

Etherisc smart contracts code corresponds to the recommended Code Style. The overall contract complexity is low and make it easy to read and understand the contract. It does not contain any complex duplicate code that can lead to di-verging pro-gram logic.

5.6 Check known security attacks

The patterns of attacks that have already been successfully executed are checked.

5.6.1 Transaction Data Length Validation

Also known as ERC20 short address attack. It is recommended to remove all checks for this attack, it is not effective and cause new problems. Etherisc fol-lows this recommendation and not use that mitigation.

5.6.2 ERC20 API: An Attack Vector on Approve/TransferFrom Methods

This attack is fixed it in OpenZeppelin StandardToken, which is used by Ether-isc.

5.6.3 Re-Entrancy (Checks-Effects-Interactions Pattern)

No findings in etherisc smart contract code.

5.6.4 Transactions May Affect Ether Receiver

A contract is exposed to this vulnerability if a miner (who executes and validates transactions) can reorder the transactions within a block in a way that affects the receiver of ether.

No findings in etherisc smart contract code.

5.6.5 Unhandled Exception

A call/send instruction returns a non-zero value if an exception occurs during the execution of the instruction (e.g., out-of-gas). A contract must check the return value of these instructions and throw an exception.

No findings in etherisc smart contract code.

5.6.6 Missing Input Validation

Unexpected method arguments may result in insecure contract behaviors. To avoid this, contracts must check whether all transaction arguments meet their desired preconditions.

The initial creation parameters for DipTge smart contract (as a consequence also for DipWhitelistedCrowdsale) smart contract are not checked meet their desired preconditions. For example, it is possible to set the hard cap less than the min cap or start public time before start open priority time by mistake.

6. Limitations

We just reviewed and audited the Solidity source code of the smart contract. We did not review the low-level assembly code that is created by the Solidity compiler. We only audited the Etherisc smart contract and did not evaluate the Ethereum project in any other matters.