VERSION 0.5

Etherisc Smart Contracts Engagement Report

Auditors
Christoph Jentzsch (christoph@slock.it)
Jonas Bentke (jonas@slock.it)

Address
Slock.it UG (haftungsbeschraenkt)
Markt 16, 09648 Mittweida,
Sachsen, Germany
HRB 30026

1. Introduction

Etherisc aims to be a decentralized insurance platform. This review seeks to enumerate implementation choices that can facilitate or expose six selected smart contracts from this project to attack or exploitations commonly found in the wild.

Note, this review solely covers the six smart contracts used for the token sale with the exception of everything else (view scope).

This review does *not* represent an endorsement for the upcoming Etherisc token sale and does not constitute support or analysis of the Etherisc project, business plan, capability and profitability, either for its team or the project. This document is a purely technical analysis of the selected six (6) smart contracts.

Please consult the risk and disclaimer sections of this document for more information.

2. Scope

2.1 Services In-scope

This review covers the following smart contracts that represent the Etherisc token sale:

- `TokenStake.sol`
- `DipToken.sol`
- `DipTge.sol`
- `DipWhitelistedCrowdsale.sol`
- `TokenTimelock.sol`
- `VestedTokens.sol`

We will solely examine the aforementioned six smart contracts at, and only at, the hash of 173cc61d936c8f2c0859e9f5cdbdbb6b29b251d6

This review seeks to enumerate implementation choices that can facilitate or expose these six smart contracts to attacks or exploitations listed in the following two documents:

http://solidity.readthedocs.io/en/develop/security-considerations.html
(this specific document as it was written on the 20th September 2017)

https://github.com/ConsenSys/smart-contract-best-practices
(this specific document as it was written on the 20th September 2017)

2.2 Areas Out of Scope

Any file not included in the agreed six smart contracts listed above or within the Github Hash version agreed above.

Assessment of any solution, site or device for physical security.

Any issues not contained within code owned by the customer unless otherwise specified.

Developing exploits or proof of concepts for vulnerabilities found.

No development or remediation work will be conducted as part of this engagement, including development of security assessment tools, techniques, scenarios, standards or metrics.

Implementation for any recommendations developed in this phase of the project, including documentation of any existing or recommended processes, standards, policies or guidelines.

Any security assessment for other applications besides those specified above, this includes the hosting environment of the application.

Any security assessment of the EVM byte code, Solidity compiler, Ethereum protocol and implementations, user interfaces and any documentation, regardless of release date.

3. Executive Summary

The 'crowdsale' is split into different timeslots where certain individuals may be allowed to buy tokens. This is enforced by the `DipTge` contract. Buying tokens through the `DipTge` contract results in a minting process through the `DipToken` contract. This contract is a token contract implementing `ERC20` standards and adds additional functionality including pausing and minting. The final contract is a vesting contract that enables said individuals to vest any `ERC20` token (in this case the `DipToken`) and redeem it after the end of vesting period.

4. Risks

The suggestions delivered as part of this engagement are made based on Slock.it UG and/or industry accepted practices, nothing contained herein (or in any deliverables provided hereunder) should be relied upon as or otherwise considered to be a certification, warranty, guarantee, or other validation that your computing environment is and will remain secure from attack.

Ethereum smart contracts are a fairly new technology which lack many of the analysis tools from other, more mature programming language. For this reason, Slock.it UG makes no warranties, express or implied, in this document. Furthermore, the conduct of this review does not warrant that the smart contracts reviewed will be free from bugs, errors, exploits or generally impervious to attack, even after our suggestions have been implemented.

Finally, all suggestions provided, if any, should be implemented only after thorough testing to ensure that no performance anomalies are introduced. In some cases, additional work will be required based upon actual business needs and applications that may be deployed by Etherisc.

5. Code Analysis

DipTge		
Ref.	Issue	Description
5.1	Medium Severity Pause functionality has limited effect.	The pause function from the token contract does not influence the minting process, because the buyToken function from DipTge is using mint in order to create tokens for the user, even if the token contract is paused, users can still buy tokens. This might be a risk in case the token contract is compromised. The same is true for the salvageTokens function. Suggested Solution Add whenNotPaused modifier to the affected functions.
5.2	Critical Severity Ownership of DipToken	The DipTge contract is the owner of the DipToken contract. Only onlyOwner functions that are implemented into DipTge can be executed successfully. It is the pause system as a whole that is affected by this issue as well as the salvageTokens function in DipToken. The only way to pause the token contract is through the constructor, which is discarded after creation. After the initial pause, there is no function to pause it again at a later stage, in case of emergency. Suggested Solution Implement a forward function in DipTge which can only be called by the owner.
5.3	Low Severity Fallback function	The fallback function as inherited by the Crowdsale.sol contract uses more than 2300 gas and will generally fail if the call is not explicitly sending more gas. This means any <address>.send or <address>.transfer to this function will fail.

5.4	Medium Severity Missing invariants	Invariants are a good way to ensure that everything in the contract works as expected. They should be defined as states that are never reachable and run an `assert` on them. This will warn the user against irregular and unexpected behaviour and shows security leaks in advance, or even prevent them from happening at all. There are Invariants in the spec. Suggested Solution We recommend checking the spec Invariants after each state changing function call.
5.5	Medium Severity Validation of input parameter in `DipWhiteListedCrowdsale`	In the constructor from `DipWhiteListedCrowdsale` a validation of the input parameters is missing. Suggested Solution Validation of that input data would help ensure that the passed timestamps are consistent with each other.
5.6	Medium Severity Unsafe Math operation	Use of subtraction without safe math functionality at `DipWhitelistedCrowdsale.sol L111`. Suggested Solution Use `safeMath` library for this operation
5.7	Low Severity Min cap has no purpose	There is no minimum cap in the specification and it is not enforced in the contracts. Suggested Solution It should be either removed from the `buyToken` function or enforced by allowing a refund in the case it is not met.
5.8	Low Severity No speed bumps	In case an attacker takes control of the keys of the wallet contract/address, there is no delay in value transfers to respond. Suggested Solutions Delay every value transfer by a certain amount of time.

DipToken		
5.9	No Severity Unused state changing code	In `Crowdsale.sol L89` the function `forwardFunds` is never called. Suggested Solution Remove unused code since it removes complexity and therefore security risks
5.10	Low Severity Calling a non-constant function from a constant function	`DipWhitelistedCrowdsale.sol L223` `setCrowdsaleState` is called, which is a non-constant function. `validPurchase`, the function from which it is called, is a constant. Therefore no state change will be made. Suggested Solution Remove function call
5.11	No Severity Inconsistent usage of type alias	In `DipWhitelistedCrowdsale.sol L89` `uint` is used instead of `uint256` which is used in the rest of the contract. Suggested Solution Use `uint256` (preferred) or `uint` consistently
VestedToken		
5.12	Critical Severity `stakeFor` lock	The `stakeFor` function stakes token without a timelock. Releasing tokens only works with the `releaseTimeLock` function and the internal accounting. Therefore, `stakeFor` should be an internal function like `stake`. Otherwise an actor could call `stakeFor` directly and would never be able to release his token. Suggested Solution Make `stakeFor` internal

5.13	No Severity Missing Natspec documentation	The documentation of the code is limited and is not complete (examples: `VestedTokens.sol` and `TokenStake.sol` are missing NatSpec documentation) Suggested Solution A complete documentation of every function following the NatSpec standard.

5.14	Low Severity Owner not used	`VestedToken` inherits from `TokenTimeLock` which inherits from Ownable. No functionality of Ownable is used in the contract. Suggested Solution Remove inheritance from `Ownable`
`TokenStake`		
5.15	No Severity Add indexed to events involving addresses	It is recommended to index addresses for better light client integration. **Suggested Solution** Add indexed in `TokenStake.sol L27-28` and `VestedToken.sol L19`
5.16	No Severity Add public getter	It is more convenient for UI developer and for interoperability with other smart contract to add public getters for state variables Suggested Solution add missing “public” to `TokenStake.sol L24` and `DipTge.sol L37`

6. Design Analysis

This section analyzes whether the smart contracts may be vulnerable to common security risks and whether certain industry-standard security recommendations were followed.

6.1 External calls

The most important security flaws arise from calls to unknown external sources. Except of one call to `msg.sender`, all three contracts do not have any calls to unknown contracts. The only calls are to the wallet, to the `DipToken` contract, to an `ERC20` token contract supplied by the owner and the one call to `msg.sender`. This call is executed using the `transfer` functionality.

Only 2300 gas are sent with the call, therefore no state changes can occur in the case of any code execution of code which belongs to msg.sender. The calls to the `wallet` and the `DipToken` are also executed either through `send` or `transfer`, both only giving 2300 gas to the process. Return parameters of *send* are properly caught and checked.

Running the function `salvageTokens` from `DipTge` and `DipToken` will send all tokens owned by the contract on any given `ERC20` token to a given address. This call is only secured through the `onlyOwner` modifier.

6.2 Modifier

'Callable' functions for the user (non-owner) are restricted to the necessary calls for the token, the token sale and the vesting processes through modifiers.

6.3 Math operations

In order to remove the risk of overflow and underflow most mathematical operations are done with the `SafeMath` library provided by `OpenZeppelin`, except of one (see issue 5.6 for this exception). The function `calculateMaxContribution` is using an unsafe subtraction which should be replaced with the safe `sub` function of the `safeMath` library.

There is one division operation in the `TokenStake` contract. The function `grant` divides the `vestingPeriod` in parts. Below it we find a check that will throw if there was a integer rounding. This requires the user that calls *grant* to choose a `vestingPeriod` and a `cliff` that does not create an uneven split.

6.4 Griefing through inactivity

The contracts do not contain any logic where one user has to wait for input of another user. Approving token transfers for another user might fall under that category, but as it is a standard `ERC20` token, dapps from third party institutions should be aware of that.

6.5 Race conditions

Due to the fact that there are no unsafe external calls, there is no possibility of re-entrancy or cross function race conditions, therefore there are no mutex implementations or similar constructs.

6.6 Transaction ordering

The contracts do not contain transactions that depend on one another, each action from the user is one closed use case. There are stages in the contract but having several transactions to the contract in one block does not affect its security.

6.7 Timestamp dependencies

Timestamps play an important role in these contracts. However, the use cases do not require a totally accurate timestamp to function properly. In the contracts, `now` is used to make sure that a set timeframe is enforced upon making a purchase or changing the state of the 'crowdsale'. The small time difference between the miners can practically be ignored looking at the uncertainty that arises from the blocktime itself.

One possible attack vector lies in the fact that the miner can manipulate the timestamp to a certain degree. Such an attack would allow the miner to either accept or reject transactions that call the `buyToken` function, by adding or subtracting from the timestamp accordingly. In some edge cases, this could change the outcome of the crowdsale as a whole, by denying the lower funding limit to be reached.

6.8 Unbound loops

There are two `for` loops, both are capped by user input. The first is `DipTge` and is used to add multiple addresses to an array. The second is a loop to set a number of timelocks dependent on the `vestingPeriod` found in `VestingToken`. The user can send large arrays as input parameter which will lead to an out-of-gas exception. But this does not affect the security of the contracts as a whole, or other users.

6.9 DoS

At the `buyingToken` function potential code from the user is called. This call, however, is used to pay back the user and does not influence the workflow of the crowdsale. To throw an exception at this point in the code would cost the user even more then accepting the money back. Every call is handling one value transfer at a time or in the case of the `grant` function only its own assets, there is no attack surface for DoS attacks against the contract system. Nevertheless, a DoS attack against the Ethereum network is always possible. Someone could spam the network by sending transactions with high gas prices, leading to a situation where the contract system can not be used with average gas prices.

6.10 Circuit breakers

It is important to implement methods that stop the workflow in case of emergencies. Such methods could include pause functions. The `DipToken` includes a pause mechanism that enables the owner to stop all `transfer` and `approval` functions. This implementation is not practically usable due to the issues described in 5.1 and 5.2.

6.11 Speed bumps

In conjunction with a working pause system, speed bumps delay any transfers of value so there is time to pause the system and respond. It is only at the moment of buying tokens that Ether is used and sent. This Ether goes straight into a pre-assigned wallet leaving no incentive for an attacker to attempt to withdraw any Ether from the contract itself. In case the wallet contract is taken over by an attacker, there would be no time to respond.

6.12 Type deduction

We found no possibility of type deduction in the contracts.

6.13 Call data

There is no direct use of `msg.data` which excludes errors through dirty higher bits.

6.14 Inheritance overwrite

The inheritance graph is extensive, nevertheless, there is no unexpected use of `super` functions along the C3 linearization graph. All `super` calls are directed towards function signatures that are unique to their parent contracts and thus avoid calling unintended functions.

6.15 Upgradability

None of the contracts are updatable. It would be a security improvement for the contracts in case of an unexpected error or newly discovered EVM vulnerability. Making the tokens upgradable will allow the owner to change broken code and unlock functionality. It is also important to consider updatable contracts in the context of future compatibility and protocol changes. Updateable contracts would mean the introduction of secure governance and a set of procedures on how to upgrade the contracts. Upgradability means the contracts can be changed at a loss of trust from users (as they can no longer rely on the functionality they initially trusted to exist in the future). All being considered, we still recommend including upgradability in the beginning and to remove that option in the future (by setting the owner key to zero) once the contracts have been battle tested.

6.16 Fallback functions

Except for the fallback function in the `DipTge` contract there are no others defined. The one in `DipTge` will not work when used with `<address>.send` or `<address>.transfer` because only 2300 gas is initially supplied. Both the other contracts will throw by default.

6.17 Invariants

Invariants should be checked at the end of a function to ensure that the reached state is safe. Some invariants are checked in the code (such as `validPurchase`) but they are not checked at the end of a transaction (in particular in `buyTokens`). The specification defines a number of invariants which are not enforced in invariant checks. This reduces their overall security. Invariants are preventive measures that ensure the correct operation of the contract. `(See issue 5.4)`

7. Disclaimers

Except as expressly set forth in this agreement or any applicable attachment, Slock.it UG disclaims all other warranties and representations with respect to the services and deliverables, express or implied, either in factor by operation of law, statutory or otherwise, including warranties of merchantability, fitness for a particular purpose, non-infringement, or title and warranties as to the quality, suitability, adequacy, accuracy, or completeness of the services and deliverables.

The warranties, obligations, and liabilities of slock.it ug and the rights, claims, and remedies of Etherisc specifically set forth in this review are exclusive. Etherisc hereby releases Slock.it UG from all other warranties, obligations, and liabilities, than otherwise mentioned in review and hereby waives all other rights, claims, and remedies against Slock.it UG and its affiliates, express or implied, arising by law or otherwise, with respect to any equipment, information, or other tangible or intangible items or services provided under this agreement, and releases Slock.it UG from all liability for loss or damage sustained relating thereto.