BFDToken Contract Audit

Author: Xia Yang

The VaaS Platform

Lindong Technology Corporation of Chengdu

2018/02/28

Audited Material Summary

In order to develop the BFDChain protocol and strengthen the network effect, there will be a number of digital tokens called BFDT issued. These BFDT can be used to use core components to pay for copyright fees. Holding BFDT can become a member of the Befund platform. BFDT will be created and distributed once after the issuance of tokens. The total amount is fixed, and will not be increased or reduced.

The audit aims at the BFDToken contract, which implements the issuance of tokens according to the EIP20 Interface.

BFDToken.sol

The BFDToken contract implements the issuance of tokens and is what participants will directly interact with. It inherits all the features of EIP20Interface and *SafeMath* contract. The EIP20Interface is a standard API for tokens within smart contracts and the *SafeMath* contract provides basic safe mathematics calculation function.

```
contract BFDToken is EIP20Interface, SafeMath
```

The *SafeMath* contract is well constructed and has no security issues.

Constructor

```
function BFDToken() public {
totalSupply = 20*10**26;
balances[msg.sender] = totalSupply;
owner = msg.sender;
}
```

It sets the contract's token amount to 2 billion. And The entire supply is allocated to the owner and has no security issues.

BFDToken Public Functions

allocateToken

```
function allocateToken(address _to, uint256 _eth, uint256
_type) isOwner notFinalised public {
```

```
        require(_to != address(0x0) && _eth != 0);
        require(addressType[_to] == 0 || addressType[_to] ==
_type);
        addressType[_to] = _type;
        uint256 temp;
        if (_type == 3) {
            temp = safeMul(_eth, 60000 * 10**18);
            balances[_to] = safeAdd(balances[_to], temp);
            balances[msg.sender]                                =
safeSub(balances[msg.sender], temp);
            releaseForSeed[_to][0]                              =
safeDiv(safeMul(balances[_to], 60), 100);
            releaseForSeed[_to][1]                              =
safeDiv(safeMul(balances[_to], 30), 100);
            releaseForSeed[_to][2] = 0;

            AllocateToken(_to, temp, 3);
        } else if (_type == 4) {
            temp = safeMul(_eth, 20000 * 10**18);
            balances[_to] = safeAdd(balances[_to], temp);
            balances[msg.sender]                                =
safeSub(balances[msg.sender], temp);
            AllocateToken(_to, temp, 4);
        } else if (_type == 5) {
            temp = safeMul(_eth, 12000 * 10**18);
            balances[_to] = safeAdd(balances[_to], temp);
            balances[msg.sender]                                =
safeSub(balances[msg.sender], temp);
            AllocateToken(_to, temp, 5);
```

```
        } else {
            revert();
        }
    }
```

The alloeateToken function implements allocate token for seed investors (3), angel investors (4) and regular investors (5). This function, through modifiers, only allows owner to allocate tokens, and allocating before the contract's token ends distribution. If both checks pass, the function assigns token in corresponding proportional, according to the different types.

Arithmetic Security

The function only sub msg.sender balances using *SafeMath* contract to protect the calculation safe, the total allocated tokens is at most 2 billion.

Suggestion: When this function is called, if the owner inputs a wrong type, there is no way to remedy it. A *ChangeType* function can be added to change the wrong type, and only the owner has the permission to call this function.

allocateTokenForTeam

```
function allocateTokenForTeam(address _to, uint256 _value)
isOwner notFinalised public {
        require(addressType[_to] == 0 || addressType[_to] ==
1);
        addressType[_to] = 1;
        balances[_to] = safeAdd(balances[_to], safeMul(_value,
10**18));
        balances[msg.sender] = safeSub(balances[msg.sender],
safeMul(_value, 10**18));

        for (uint256 i = 0; i <= 4; ++i) {
            releaseForTeamAndAdvisor[_to][i]                =
safeDiv(safeMul(balances[_to], (4 - i) * 25), 100);
```

```
        }
        AllocateToken(_to, safeMul(_value, 10**18), 1);
    }
```

The *allocateTokenForTeam* function is used to allot tokens for team. It can only be called by the owner and the tokens can only be distributed before the allocation ends.

_value is the number of tokens allocated, in units of 10**18. For example, when the value is 100, the actual number of tokens is 100*10**18. This address type is 1.

Arithmetic Security

Similar to *allocateToken*, *SafeMath* contract endures the calculation would not out of limits. The time of locked position is 2 years. 25% will be released each period.

allocateTokenForAdvisor

```
function allocateTokenForAdvisor(address _to, uint256 _value)
isOwner public {
        require(addressType[_to] == 0 || addressType[_to] ==
2);
        addressType[_to] = 2;
        balances[_to] = safeAdd(balances[_to], safeMul(_value,
10**18));
        balances[msg.sender] = safeSub(balances[msg.sender],
safeMul(_value, 10**18));

        for (uint256 i = 0; i <= 4; ++i) {
            releaseForTeamAndAdvisor[_to][i]                    =
safeDiv(safeMul(balances[_to], (4 - i) * 25), 100);
        }
        AllocateToken(_to, safeMul(_value, 10**18), 2);
    }
```

The *allocateTokenForAdvisor* function implements distribution of tokens for

advisor. Only the owner can call this function and the token can only be distributed when the allocation is not finalized.

_value is the number of tokens allocated, in units of 10**18.

Arithmetic Security

Similar to *allocateToken*. The time of locked position is 2 years. 25% will be released each period.

changeOwner

```
function changeOwner(address _owner) isOwner public {
        owner = _owner;
    }
```

The *changeOwner* function change the owner of this contract, which ensures that only the owner of the contract can call this function.

setFinaliseTime

```
function setFinaliseTime() isOwner public {
        require(finaliseTime == 0);
        finaliseTime = now;
    }
```

The setFinaliseTime function records the time when crowdfunding completed.

Due to the isOwner modifier, it can only be called by the owner of the contract.

transfer

```
function transfer(address _to, uint256 _value) public returns
(bool success) {
        require(canTransfer(msg.sender, _value));
        require(balances[msg.sender] >= _value);
        balances[msg.sender] -= _value;
        balances[_to] += _value;
        Transfer(msg.sender, _to, _value);
        return true;
```

```
    }
```

The transfer function acts like the usual EIP20 transfer, except that the account of msg.sender must be in a tradable state , otherwise the function will throw.

Arithmetic Security

As above, there will just not be enough tokens in existence to cause an overflow. The second *require* ensures that the sender owns at least *_value* so underflow is not possible either.

canTransfer

```
function canTransfer(address _from, uint256 _value) internal
view returns (bool success) {
        require(finaliseTime != 0);
        uint256 index;
        if (addressType[_from] == 0 || addressType[_from] == 4
|| addressType[_from] == 5) {
            return true;
        }
          if (addressType[_from] == 3) {
            index = safeSub(now, finaliseTime) / 60 days;
            if ( index >= 2) {
                index = 2;
            }
            require(safeSub(balances[_from],     _value)     >=
releaseForSeed[_from][index]);
        }    else    if    (addressType[_from]    ==    1    ||
addressType[_from] == 2) {
            index = safeSub(now, finaliseTime) / 180 days;
            if (index >= 4) {
                index = 4;
            }
```

```
            require(safeSub(balances[_from],     _value)     >=
releaseForTeamAndAdvisor[_from][index]);
        }
        return true;
    }
```

The *canTransfer* is an internal function that used to check whether the *_from* can transfer *_values*, based on the type of the *_from* and the locking rules described in the "*issuance plan*" respectively . The function only can be executed after the allocation is finalised.

Consultants, partners and teams unlock 25% every 6 months, seed investors unlock 30% every 2 month and other types have no lock-up restrictions. The function will return true only when the *_from* account still satisfy the locking rule after trading *_value* amount of tokens.

transferFrom

```
    function transferFrom(address _from, address _to, uint256
_value) public returns (bool success) {
        require(canTransfer(_from, _value));
        uint256 allowance = allowed[_from][msg.sender];
        require(balances[_from]  >=  _value  &&  allowance  >=
_value);
        balances[_to] += _value;
        balances[_from] -= _value;
        if (allowance < MAX_UINT256) {
            allowed[_from][msg.sender] -= _value;
        }
        Transfer(_from, _to, _value);
        return true;
    }
```

The *transferFrom* function allows *_spender* to transfer tokens from address *_from* to address *_to with _value* amount, and will fire the *Transfer* event. This function only allows the transaction if the participant is:

1. *_from* account is in a tradable state.

2. The number of tokens that msg.sender can withdraw from *_from* account is greater than or equal to the *_value*.

3. The number of tokens in *_from* account is greater than or equal to the *_value*.

If all these checks pass, the function will execute the transaction, add the tokens to the *_to* account and reduce the tokens of *_from* account .

Arithmetic Security

As above, checking the balance can ensure that the calculation of the transaction tokens does not cause an error when minus is less than subtraction. However, we still recommend using *SafeMath* to completely eliminate the possibility of calculation errors.

balanceOf

```
function balanceOf(address _owner) public view returns
(uint256 balance) {
        return balances[_owner];
    }
```

Simply returns the balance of account associated to *_owner*. Standard EIP20 behaviour.

approve

```
function approve(address _spender, uint256 _value) public
returns (bool success) {
        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
        return true;
    }
```

The *approve* function allows *_spender* to withdraw from the account of msg.sender multiple times, up to the *_value* amount. If this function is called again it overwrites the current allowance with *_value* .

allowance

```
function allowance(address _owner, address _spender) public
view returns (uint256 remaining) {
        return allowed[_owner][_spender];
    }
```

Simply returns the how many tokens are allowed to withdraw from *_owner* to *_spender*. Standard EIP20 behaviour.

Security issue

We have audited the all source codes of the BFD Token contract line by line, based the requirement described in the website (http://www.befund.io/assets/upload/Befund_Presentation_EN.pdf?v=1.0.0.60). The design and implementation of all contracts conform to the functional requirement, and there aren't common code level issues in these all contracts.

Disclaimer

This is an audit of the smart contracts and their security and correctness only, and not of the platform or anything else.