Skillchain (SKI)

Smart Contracts Audit

v1.0

Vittorio Minacori
BC1 - Blockchain Pioneers

Jan 28, 2019

Disclaimer

The audit makes no statements or warrantees about utility of the code, safety of the code, suitability of the business model, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bugfree status. The audit documentation is for discussion purposes only.

Overview

The audit was performed by **BC1 - Blockchain Pioneers** (https://www.bc1.tech) on all the Smart Contracts provided by the **Skillchain** (https://www.skillchain.io) team.

This report can be found here https://github.com/bc1tech/skillchain-audit/blob/master/audit/SkillChain-BC1-Audit-Report-V1.0.pdf.

Code is hosted on **Skillchain** official GitHub repo (https://github.com/Skillchain/smart-contracts) and was forked here https://github.com/bc1tech/skillchain-audit by **BC1 - Blockchain Pioneers** team at commit **f725ac8ec66798ef75a095b550aa3126e13504b0**.

Audited smart contracts are in the ***contracts*** folder and they have been deployed by Skillchain on Ethereum blockchain at the following addresses:

- **SkillChainToken** 0x996dc5dfc819408dd98cd92c9a76f64b0738dc3d
- **SkillChainContributions** 0x0a5e725fa3fd699a73202bc089201f49ad4e99a9
- **SkillChainPrivateSale** 0x0bcaa1c8da60b9f9711191ebf192c879870b322a
- **SkillChainPresale** 0x5f84a30e819fe1d62452899a28a8b3d0776f9ff0

Smart contracts was compiled using ***solc v0.4.19+commit.c4cbbb05.Emscripten.clang***.

Smart contracts use the ***OpenZeppelin*** library at version 1.6.0.

Note: the following audit report takes into account the compiler and the library used at the time of the deploy. We are also “inheriting” the audit done from OpenZeppelin on its own contract at the used release (v1.6.0) and located here https://github.com/OpenZeppelin/openzeppelin-solidity/blob/v1.6.0/audit/ZeppelinAudit.md.

Audit Details

Arithmetic Safety Audit ✅

The arithmetic security audit is divided into three parts: integer overflow audit, integer underflow audit and operation precision audit.

Integer Overflow Audit ✅

Solidity can handle 256 bits of data at most. When the maximum number increases, it will overflow. If the integer overflow occurs in the transfer logic, it will make the amount of transfer funds miscalculated, resulting in serious capital risk.

Integer Underflow Audit ✅
Solidity can handle 256 bits of data at most. When the minimum number decreases, it will underflow. If the integer underflow occurs in the transfer logic, it will make the amount of transfer funds miscalculated and lead to serious capital risk.

Operation Precision Audit ✅
Solidity performs type coercion in the process of multiplication and division. If the precision risk is included in the operation of capital variable, it will lead to user transfer logic error and capital loss.

Audit Result: Passed
Security Recommendation: No.

Competitive Competition Audit ✅

The competitive competition audit is divided into two parts: reentrancy audit and transaction ordering dependence audit. With competitive vulnerabilities, an attacker can modify the output of a program by adjusting the execution process of transactions with a certain probability.

Reentrancy Audit ✅
Reentrancy occurs when external contract calls are allowed to make new calls to the calling contract before the initial execution is complete. For a function, this means that the contract state may change in the middle of its execution as a result of a call to an untrusted contract or the use of a low level function with an external address.

Transaction Ordering Dependence Audit ✅
Since miners always get rewarded via gas fees for running code on behalf of externally owned addresses (EOA), users can specify higher fees to have their transactions mined more quickly. Since the Ethereum blockchain is public, everyone can see the contents of others' pending transactions. This means if a given user is revealing the solution to a puzzle or other valuable secret, a malicious user can steal the solution and copy their transaction with higher fees to preempt the original solution. If developers of smart contracts are not careful, this situation can lead to practical and devastating front-running attacks.

Audit Result: Passed
Security Recommendation: No.

Access Control Audit ✅

Access control audit is divided into two parts: privilege vulnerability audit and overprivileged audit.

Privilege Vulnerability Audit ✅
Smart contracts with privilege vulnerability, attackers can weigh their own accounts to gain higher execution privileges.

Overprivileged Audit ⚠️
Overprivileged auditing focuses on whether there are special user privileges in audit contracts, such as allowing a user to unlimitedly mine tokens.

Audit Result: Passed
Security Recommendation: Check the Audit Results / Low severity section for details.

Security Design Audit ✅

Security design audit is divided into five parts: security module usage, compiler version security, hard-coded address security, sensitive function usage security and function return value security.

Security Module Usage ✅

The security module usage audit whether the smart contract uses the SafeMath library function provided by OpenZeppelin to avoid overflow vulnerabilities; if it does not, whether the transfer amount is strictly checked during the execution.

Compiler Version Security Audit ⚠️

Compiler version security focuses on whether the smart contract explicitly indicates the compiler version and whether the compiler version used is too low to throw an exception.

Hard Coded Address Security Audit ✅

Hard-coded address security audit static addressed in the smart contract to check whether there is an exception to the external contract, thus affecting the execution of this contract.

Sensitive Functions Audit ✅

Sensitive functions audit checks whether the smart contract uses the not recommended functions such as fallback, call and tx.origin.

Function Return Value Audit ⚠️

Function return value audit mainly analyzes whether the function correctly throws an exception, correctly returns to the state of the transaction.

Audit Result: Passed
Security Recommendation: Check the Audit Results / Low severity section for details.

Denial of Service Audit ✅

Denial of service attack sometimes can put the smart contract offline forever by maliciously behaving when being the recipient of a transaction, artificially increasing the gas necessary to compute a function, abusing access controls to access private components of smart contracts, taking advantage of mixups and negligence and so on.

Audit Result: Passed
Security Recommendation: No.

Gas Optimization Audit ✅

If the computation of a function in a smart contract is too complex, such as the batch transfer to a variable-length array through a loop, it is very easy to cause the gas fee beyond the block's gas Limit resulting in transaction execution failure.

Audit Result: Passed
Security Recommendation: No.

Design Logic Audit ✅

In addition to vulnerabilities, there are logic problems in the process of code implementation, resulting in abnormal execution results.

Audit Result: Passed
Security Recommendation: No.

Testing Process

The BC1 - Blockchain Pioneers team forked the source code repository and added a standard environment to compile, test and analyze contracts.

Fixed package version

First we used package versions fixed to the time of Skillchain token deploy transaction https://etherscan.io/tx/0xf539799fc7f62e0a68caefda5d83290a285800d5ff16e97394438686b2a399be.

Code source here https://github.com/bc1tech/skillchain-audit/tree/master.

Packages:

- truffle: 4.0.6
- solc: 0.4.19
- solium: 1.1.7

Code build can be found here https://travis-ci.org/bc1tech/skillchain-audit/builds/485351444.

Lint passed ✅

[https://travis-ci.org/bc1tech/skillchain-audit/jobs/485351445]
Only a warning on **Avoid using 'now' (alias to 'block.timestamp')**.

Test passed ✅

[https://travis-ci.org/bc1tech/skillchain-audit/jobs/485351446]
Test completed with 116 passing and 0 failed.

Test Coverage 100% ✅

[https://coveralls.io/builds/21308961]
26 of 26 branches covered (100.0%).
64 of 64 relevant lines covered (100.0%).
23.56 hits per line.

Latest package version

Then we used latest package versions based on code compatibility.

Code source here https://github.com/bc1tech/skillchain-audit/tree/latest.

Packages:

- truffle: 4.1.15
- solc: 0.4.25
- solhint: 1.5.0

Code build can be found here https://travis-ci.org/bc1tech/skillchain-audit/builds/485956500.

Lint failed ❌

[https://travis-ci.org/bc1tech/skillchain-audit/jobs/485956502]
Task failed due the “Visibility modifier must be first in list of modifiers” requested in Solidity > 0.4.21.

Test passed ✅

[https://travis-ci.org/bc1tech/skillchain-audit/jobs/485956501]
Test completed with 116 passing and 0 failed.

Test Coverage 100% ✅

[https://coveralls.io/builds/21336182]
26 of 26 branches covered (100.0%).
64 of 64 relevant lines covered (100.0%).
23.56 hits per line.

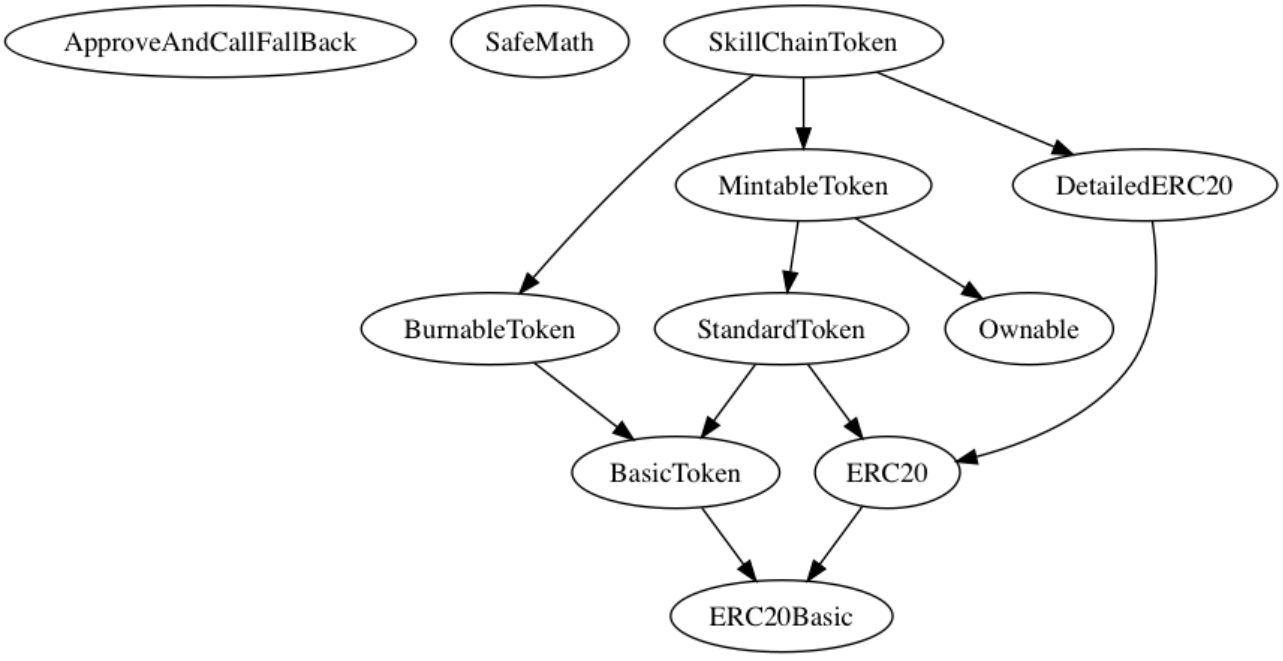
Structure Analysis

Inheritance Tree Graph

The BC1 - Blockchain Pioneers team build a set of inheritance tree graph for each smart contract provided.

Graphs can be found at
https://github.com/bc1tech/skillchain-audit/tree/master/audit/inheritance.

As sample we included in this document the **SkillChainToken** inheritance tree graph below.



Control Flow Graph

The BC1 - Blockchain Pioneers team build a set of control flow graph for each smart contract provided.

Graphs can be found at
https://github.com/bc1tech/skillchain-audit/tree/master/audit/graph.

File Description Table

The BC1 - Blockchain Pioneers team build a set of file description table for each smart contract provided.

Tables can be found at
https://github.com/bc1tech/skillchain-audit/tree/master/audit/table.

Audit Results

Critical severity ✅

No critical severity issues were found.

High severity ✅

No high severity issues were found.

Medium severity ✅

No medium severity issues were found.

Low severity ⚠️

- *transfer()* and *transferFrom()* use **SafeMath** functions of the extended OpenZeppelin library, which will cause them to throw instead of returning false. Not a security issue but departs from standard. It is inherited from OpenZeppelin library and it is a very low severity issue.
- **SkillChainToken** has not a way to define the maximum number of tokens mintable. If contracts weren't been already deployed, we would have suggested to set a max cap of mintable tokens and we would have flagged this as medium severity. Since there is no *finishMinting* call into the Crowdsale contract, by transferring ownership to a wallet, the owner could generate more tokens than declared. Since the owner called the *finishMinting* function manually at https://etherscan.io/tx/0x6ac1a96e3f862ea57e800c4c5caac56b2c64121052ea40305291db3a17ec0a45 there is no way to generate other tokens so we are marking it as very low severity.
- Compiler version is not fixed. Since contracts were already deployed with solc 0.4.19 we mark it as very low severity.

Suggestions

Some minor changes in code style can be made to be compliant to solidity 0.5 requirements.

Code Analysis

SkillChainToken

Source code:
https://github.com/Skillchain/smart-contracts/blob/master/contracts/SkillChainToken.sol

This contract is the main token contract.
SkillChainToken extends **DetailedERC20**, **MintableToken** and **BurnableToken** from the OpenZeppelin library and uses **SafeMath** for uint256.
SkillChainToken is a standard ERC20 token with the ability to be minted by token owner and burned by token holder. It has also a name, a symbol and a decimals amount.

```
contract SkillChainToken is DetailedERC20, MintableToken, BurnableToken {
```

Modifiers

canTransfer

```
modifier canTransfer() {
    require(mintingFinished);
    _;
}
```

This modifier ensures that minting has been finished before allowing token transfer.

Constructor

```
function SkillChainToken() DetailedERC20("Skillchain", "SKI", 18) public {}
```

The contract constructor initiates the **DetailedERC20** with token name, symbol and decimals.

Suggestion: Defining constructor as function with the same name as the contract is deprecated. Use “constructor(...) { ... }” instead.

Public methods

transfer

```
function transfer(address _to, uint _value) canTransfer public returns (bool) {
    return super.transfer(_to, _value);
}
```

This function adds the *canTransfer* modifier, and then calls the *transfer* function of the parent contract.

Suggestion: Visibility modifier "public" should come before other modifiers.

transferFrom

```
    function transferFrom(address _from, address _to, uint _value) canTransfer public
returns (bool) {
        return super.transferFrom(_from, _to, _value);
    }
```

This function adds the *canTransfer* modifier, and then calls the *transferFrom* function of the parent contract.

Suggestion: Visibility modifier "public" should come before other modifiers.

approveAndCall

```
    function approveAndCall(address _spender, uint256 _tokens, bytes _data) public
returns (bool success) {
        approve(_spender, _tokens);
        ApproveAndCallFallBack(_spender).receiveApproval(
            msg.sender,
            _tokens,
            this,
            _data
        );
        return true;
    }
```

This function calls the ERC20 *approve* function and then executes a function on *_spender* with the *_data* parameter, if the function ends successfully allows *_spender* to withdraw from caller account multiple times, up to the *_tokens* amount. If this function is called again it overwrites the current allowance with *_tokens*. Receiver should implement the **ApproveAndCallFallBack** interface as described below.

Notes:

- Definition of *approveAndCall* is not part of the ERC20 standard but it is useful for DApp and users to have and implement because of it facilitates workflow and user experience when people want to make an action after an approval in a single transaction.
- Receiver should make sure to create user interfaces in such a way that they set the allowance first to 0 before setting it to another value for the same spender due the *approve/transferFrom* ERC20 native issue (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729). Note that this is not fixable while adhering to the current full ERC20 standard, though it would be possible to add a "secureApprove" function. The impact isn't extreme since at least you can only be attacked by addresses you approved. Also, users could mitigate this by always setting spending limits to zero and checking for spends, before setting the new limit.

transferAnyERC20Token

```
    function transferAnyERC20Token(address _tokenAddress, uint256 _tokens) onlyOwner
public returns (bool success) {
        return ERC20Basic(_tokenAddress).transfer(owner, _tokens);
    }
```

This function makes the contract owner able to recover any ERC20 token sent into the contract for error.

Suggestion: Visibility modifier "public" should come before other modifiers.

ApproveAndCallFallBack

Source code:
https://github.com/Skillchain/smart-contracts/blob/master/contracts/ApproveAndCallFallBack.sol

This is a simple interface that developers who want to accept the *approveAndCall* function should implement.

```
    contract ApproveAndCallFallBack {
```

receiveApproval

```
    function receiveApproval(
       address from,
       uint256 tokens,
       address token,
       bytes data) public;
```

This function is intended to be called after an *approve* event. Implementer should check that *msg.sender* MUST be the same of the passed *token* parameter to be sure that call come from the right token contract and (likely) also check that token address is accepted.

SkillChainContributions

Source code:
https://github.com/Skillchain/smart-contracts/blob/master/contracts/SkillChainContributions.sol

SkillChainContributions extends **Ownable** from the OpenZeppelin library and uses **SafeMath** for uint256. **SkillChainContributions** is an utility contracts where to store amounts of tokens bought by each user for future purposes (like an airdrop or similar).

```
    contract SkillChainContributions is Ownable {
```

Properties

```
mapping(address => uint256) public tokenBalances;
address[] public addresses;
```

These two properties are helpful to store addresses of people who participate in token sale and what is the amount of tokens bought by each of them.

Constructor

```
function SkillChainContributions() public {}
```

The contract constructor does nothing.

Suggestion: Defining constructor as function with the same name as the contract is deprecated. Use “constructor(...) { ... }” instead.

Public methods

addBalance

```
function addBalance(address _address, uint256 _tokenAmount) onlyOwner public {
    require(_tokenAmount > 0);
    if (tokenBalances[_address] == 0) {
        addresses.push(_address);
    }
    tokenBalances[_address] = tokenBalances[_address].add(_tokenAmount);
}
```

This function adds a *_tokenAmount* to an *_address* balance. It requires that the amount is greater than zero. If *_address* has no balance it adds the address to the *addresses* list in order to be taken by index.

Suggestion: Visibility modifier "public" should come before other modifiers.

getContributorsLength

```
function getContributorsLength() view public returns (uint) {
   return addresses.length;
}
```

This function returns the number of unique contributors.

Suggestion: Visibility modifier "public" should come before other modifiers.

SkillChainPrivateSale

Source code:
https://github.com/Skillchain/smart-contracts/blob/master/contracts/SkillChainPrivateSale.sol

This contract is responsible for the main crowdsale logic. It is for a private sale phase.

SkillChainPrivateSale extends **CappedCrowdsale** and **Ownable** from the OpenZeppelin library and uses **SafeMath** for uint256.
Basically it is a classic **Crowdsale** with a defined max cap in wei.

```
contract SkillChainPrivateSale is CappedCrowdsale, Ownable {
```

Properties

```
SkillChainContributions public contributions;
uint256 public minimumContribution = 2 ether;
```

The first property *contributions* is a reference to a **SkillChainContributions** contract, that will be deployed in constructor and that will be shared within all the sale phases.
The *minimumContribution* value, instead, indicates the minimum value accepted to enter the token sale.

Constructor

```
function SkillChainPrivateSale(
    uint256 _startTime,
    uint256 _endTime,
    uint256 _rate,
    address _wallet,
    uint256 _cap
)
CappedCrowdsale(_cap)
Crowdsale(_startTime, _endTime, _rate, _wallet)
public
{
    contributions = new SkillChainContributions();
}
```

The contract constructor initiates the **CappedCrowdsale** by setting the max value of wei acceptable. It also initiates the **Crowdsale** contract by setting *_startTime*, *_endTime*, *_rate* and *_wallet*.
The constructor deploys the **SkillChainContributions** contract.

Suggestion: Defining constructor as function with the same name as the contract is deprecated. Use “constructor(...) { ... }” instead.

Internal methods

createTokenContract

```
    function createTokenContract() internal returns (MintableToken) {
        return new SkillChainToken();
    }
```

The *createTokenContract* is an internal function that creates a **SkillChainToken** contract, returning its address. It is only ever used inside of OpenZeppelin's **Crowdsale**. This function overrides the default implementation.

validPurchase

```
    function validPurchase() internal view returns (bool) {
        bool validContribution = msg.value >= minimumContribution;

        uint256 remainingBalance = cap.sub(weiRaised);
        if (remainingBalance <= minimumContribution) {
            validContribution = true;
        }

        return validContribution && super.validPurchase();
    }
```

This function checks if user is sending more than minimum contribution. It also accepts less than the minimum contribution only if crowdsale is closing and a little amount of wei is required to cover the cap. It returns this check value AND the parent *validPurchase* check value.

Public methods

buyTokens (payable)

```
    function buyTokens(address beneficiary) public payable {
        require(beneficiary != address(0));
        require(validPurchase());

        uint256 weiAmount = msg.value;

        // calculate token amount to be created
        uint256 tokens = getTokenAmount(weiAmount);

        // update state
        weiRaised = weiRaised.add(weiAmount);

        token.mint(beneficiary, tokens);
        TokenPurchase(
```

```
            msg.sender,
            beneficiary,
            weiAmount,
            tokens
        );

        forwardFunds();

        // log contribution
        contributions.addBalance(beneficiary, tokens);
    }
```

This function is basically overriding the parent *buyTokens* function with the addition of the contribution log. It is payable and it will be called also from the default fallback function passing msg.sender as *beneficiary*. It checks that it is a valid purchase from *beneficiary*, gets the amount of tokens to be minted to *beneficiary*, mints them, emits a *TokenPurchase* event, calls the parent *forwardFunds* function to move ether to the deposit *wallet* and then calls the *addBalance* function on *contributions* as described above.

closeTokenSale

```
    function closeTokenSale(address _presaleContract) onlyOwner public {
        require(hasEnded());
        require(_presaleContract != address(0));

        token.transferOwnership(_presaleContract);
        contributions.transferOwnership(_presaleContract);
    }
```

This function closes the token sale and transfers the ownership of *token* and *contributions* to a new address that is supposed to be the presale contract. It checks that crowdsale has ended and that is not trying to transfer to the zero address and then calls the parent *transferOwnership* function.

Suggestion: Visibility modifier "public" should come before other modifiers.

transferAnyERC20Token

```
 function transferAnyERC20Token(address _tokenAddress, uint256 _tokens) onlyOwner
public returns (bool success) {
        return ERC20Basic(_tokenAddress).transfer(owner, _tokens);
    }
```

This function makes the contract owner able to recover any ERC20 token sent into the contract for error.

Suggestion: Visibility modifier "public" should come before other modifiers.

SkillChainPresale

Source code:
https://github.com/Skillchain/smart-contracts/blob/master/contracts/SkillChainPresale.sol

This contract is responsible for the main crowdsale logic. It is for a pre-sale phase.

SkillChainPresale extends **CappedCrowdsale** and **Ownable** from the OpenZeppelin library and uses **SafeMath** for uint256.
Basically it is a classic **Crowdsale** with a defined max cap in wei.

```
contract SkillChainPresale is CappedCrowdsale, Ownable {
```

Properties

```
SkillChainContributions public contributions;
uint256 public minimumContribution = 1 ether;
```

The first property *contributions* is a reference to a **SkillChainContributions** contract, that will be passed as reference in constructor and that is shared within all the sale phases.
The *minimumContribution* value, instead, indicates the minimum value accepted to enter the token sale.

Constructor

```
function SkillChainPresale(
    uint256 _startTime,
    uint256 _endTime,
    uint256 _rate,
    address _wallet,
    uint256 _cap,
    address _token,
    address _contributions
)
CappedCrowdsale(_cap)
Crowdsale(_startTime, _endTime, _rate, _wallet)
public
{
    token = SkillChainToken(_token);
    contributions = SkillChainContributions(_contributions);
}
```

The contract constructor initiates the **CappedCrowdsale** by setting the max value of wei acceptable. It also initiates the *Crowdsale* contract by setting *_startTime*, *_endTime*, *_rate* and *_wallet*.
The constructor set the **SkillChainToken** contract by the *_token* parameter address and it should be the previously deployed token.
The constructor set the **SkillChainContributions** contract by the *_contributions* parameter address and it should be the previously deployed *contributions*.

Suggestion: Defining constructor as function with the same name as the contract is deprecated. Use "constructor(...) { ... }" instead.

Internal methods

createTokenContract

```
function createTokenContract() internal returns (MintableToken) {
    return MintableToken(0);
}
```

The *createTokenContract* is an internal function referencing a **Mintable** contract to the zero address. It is only ever used inside of OpenZeppelin's **Crowdsale**.

Note: This function overrides the default implementation and it is used in this way because of the **Crowdsale** implementation of OpenZeppelin v1.6.0 creates the token during contract deployment. In this case token already exists and it is a workaround to continue using the well tested OpenZeppelin library. In the following versions of OpenZeppelin they removed the *createTokenContract* function by adding a token param to **Crowsdale** constructor so token should be deployed in a previous transaction. This should be the right way to proceed but at the time of the deploy it was aligned with the library.

validPurchase

```
function validPurchase() internal view returns (bool) {
    bool validContribution = msg.value >= minimumContribution;

    uint256 remainingBalance = cap.sub(weiRaised);
    if (remainingBalance <= minimumContribution) {
        validContribution = true;
    }

    return validContribution && super.validPurchase();
}
```

This function checks if user is sending more than minimum contribution. It also accepts less than the minimum contribution only if crowdsale is closing and a little amount of wei is required to cover the cap. It returns this check value AND the parent *validPurchase* check value.

Public methods

buyTokens (payable)

```
function buyTokens(address beneficiary) public payable {
    require(beneficiary != address(0));
    require(validPurchase());

    uint256 weiAmount = msg.value;
```

```
        // calculate token amount to be created
        uint256 tokens = getTokenAmount(weiAmount);

        // update state
        weiRaised = weiRaised.add(weiAmount);

        token.mint(beneficiary, tokens);
        TokenPurchase(
            msg.sender,
            beneficiary,
            weiAmount,
            tokens
        );

        forwardFunds();

        // log contribution
        contributions.addBalance(beneficiary, tokens);
    }
```

This function is basically overriding the parent *buyTokens* function with the addition of the contribution log. It is payable ad it will be called also from the default fallback function passing msg.sender as *beneficiary*. It checks that it is a valid purchase from *beneficiary*, gets the amount of tokens to be minted to *beneficiary*, mints them, emits a *TokenPurchase* event, calls the parent *forwardFunds* function to move ETH to the deposit wallet and then calls the *addBalance* function on *contributions* as described above.

closeTokenSale

```
    function closeTokenSale(address _icoContract) onlyOwner public {
        require(hasEnded());
        require(_icoContract != address(0));

        token.transferOwnership(_icoContract);
        contributions.transferOwnership(_icoContract);
    }
```

This function closes the token sale and transfers the ownership of token and contributions to a new address that is supposed to be the ICO contract. It checks that crowdsale has ended and that is not trying to transfer to the zero address and then calls the parent *transferOwnership* function.

Suggestion: Visibility modifier "public" should come before other modifiers.

Note: we noticed that there wasn't a public ICO done with smart contract. It is supposed to have the *finishMinting* function called by this contract but it was called manually at https://etherscan.io/tx/0x6ac1a96e3f862ea57e800c4c5caac56b2c64121052ea40305291db3a17ec0a45 so there is no way to generate other tokens and the totalSupply is now limited to 190,800,000 SKI.

transferAnyERC20Token

```
    function transferAnyERC20Token(address _tokenAddress, uint256 _tokens) onlyOwner
public returns (bool success) {
        return ERC20Basic(_tokenAddress).transfer(owner, _tokens);
    }
```

This function makes the contract owner able to recover any ERC20 token sent into the contract for error.

Suggestion: Visibility modifier "public" should come before other modifiers.

started

```
    function started() public view returns(bool) {
        return now >= startTime;
    }
```

It returns false if the Crowdsale is not started, true if the Crowdsale is started and running, true if the Crowdsale is completed.

Suggestion: Avoid using 'now' (alias to 'block.timestamp').

ended

```
    function ended() public view returns(bool) {
        return hasEnded();
    }
```

It returns false if the Crowdsale is not started, false if the Crowdsale is started and running, true if the Crowdsale is completed.

totalTokens

```
    function totalTokens() public view returns(uint) {
        return rate.mul(cap);
    }
```

It returns the total number of the tokens available for the sale.

remainingTokens

```
function remainingTokens() public view returns(uint) {
    return rate.mul(cap).sub(rate.mul(weiRaised));
}
```

It returns the number of the tokens available for the ico. At the moment that the Crowdsale starts it must be equal to *totalTokens*, then it will decrease. It is used to calculate the percentage of sold tokens as *remainingTokens* / *totalTokens*.

price

```
function price() public view returns(uint) {
    return rate;
}
```

It returns the price as number of tokens released for each wei sent.

Conclusion

No security issues were found.
As the code is already deployed with solc 0.4.19 it can be considered to be compliant with all the specifications.
Some minor changes in code style can be made to be compliant to solidity 0.5 requirements.