SingularityNet Crowdsale Audit

ZK Labs Auditing

AUTHOR: MATTHEW DI FERRANTE

2017-12-17

Audited Material Summary

The audit consists of the `AgiCrowdsale.sol` and `SingularityNetToken.sol` contracts. The git commit hash of the reviewed files is `ddc0d79580f797fb2b76cf56c31a633760970680`.

The contracts implement a fixed price crowdsale with participant whitelisting and a refunding option if the funding goal is not met.

The audit focuses on the custom code written by the SingularityNET Team.

Security

There is one main issue, the `setOwnership` function can be used by the contract owners to mint infinite tokens, due to the fact that the previous owner's balance is always set to INITIAL_SUPPLY - PUBLIC_SUPPLY and the function can be called without limit. Token allocation/minting/transfer should not happen on ownership change, and the contract should enforce that the INITIAL_SUPPLY assignment only happens once and only once.

Beyond this, the contracts are generally well constructed and use best practices and SafeMath for all sensitive logic. No major security issues that pose a risk to users or funds were found during the audit.

A security code style suggestion is to use modifiers instead of inline, top-of-function require statements.

AgiCrowdsale.sol

The `AgiCrowdsale` contract contains the logic for the Crowdsale. It is mostly self contained and inherits only from 2 helper contracts:

```
contract AgiCrowdsale is Ownable, ReentrancyGuard
```

Constructor

```
    function AgiCrowdsale(
        address _token,
        address _wallet,
        uint256 _startTime,
        uint256 _endTime,
        uint256 _rate,
        uint256 _cap,
```

```
        uint256 _firstDayCap,
        uint256 _goal
    ) {
        require(_startTime >= getBlockTimestamp());
        require(_endTime >= _startTime);
        require(_rate > 0);
        require(_goal > 0);
        require(_cap > 0);
        require(_wallet != 0x0);

        vault = new RefundVault(_wallet);
        token = SingularityNetToken(_token);
        wallet = _wallet;
        startTime = _startTime;
        endTime = _endTime;
        firstDay = startTime + 1 * 1 days;
        firstDayCap = _firstDayCap;
        rate = _rate;
        goal = _goal;
        cap = _cap;
    }
```

The constructor initializes state variables for the crowdsale, and performs some sanity checking on wallet address, dates and financial parameters.

Default Function

```
    function () external payable {
        buyTokens(msg.sender);
    }
```

The default function is simply a forwarder to `buyTokens`.

buyTokens

```
    function buyTokens(address beneficiary) internal {
        require(beneficiary != 0x0);
        require(whitelist[beneficiary]);
        require(validPurchase());

```

```
6           //derive amount in wei to buy
7           uint256 weiAmount = msg.value;
8
9           // check if contribution is in the first 24h hours
10          if (getBlockTimestamp() <= firstDay) {
11              require((contribution[beneficiary].add(weiAmount)) <=
                    firstDayCap);
12          }
13          //check if there is enough funds
14          uint256 remainingToFund = cap.sub(weiRaised);
15          if (weiAmount > remainingToFund) {
16              weiAmount = remainingToFund;
17          }
18          uint256 weiToReturn = msg.value.sub(weiAmount);
19          //Forward funs to the vault
20          forwardFunds(weiAmount);
21          //refund if the contribution exceed the cap
22          if (weiToReturn > 0) {
23              msg.sender.transfer(weiToReturn);
24              TokenRefund(beneficiary, weiToReturn);
25          }
26          //derive how many tokens
27          uint256 tokens = getTokens(weiAmount);
28          //update the state of weiRaised
29          weiRaised = weiRaised.add(weiAmount);
30          contribution[beneficiary] = contribution[beneficiary].add(
                weiAmount);
31
32          //Trigger the event of TokenPurchase
33          TokenPurchase(
34              msg.sender,
35              beneficiary,
36              weiAmount,
37              tokens
38          );
39          token.transferTokens(beneficiary,tokens);
40
41      }
```

The `buyTokens` function is the user's entry point (through the default payable) when participating in the crowdsale.

The function performs sanity checking to make sure the beneficiary is a valid address, in the whitelist,

and is performing a valid purchase (>0 wei).

If the purchase occurs within the first 24 hours of the crowdsale, the total contribution is checked against the firstDayCap, if it exceeds the cap the contract terminates.

Otherwise, the contribution is checked against the funding cap, and excess either is refunded to the sender. The succesfully invested funds are forwarded to the wallet, tokens are issued to the beneficiary, the amount raised is added to the `weiRaised` tracker variable, and the contribution is recorded for the beneficiary.

On success, a `TokenPurchase` event is emitted.

This function is internal and only ever called by the default function.

getTokens

```
    function getTokens(uint256 amount) internal constant returns (uint256)
        {
        return amount.mul(rate).div(WEI_TO_COGS);
    }
```

The `getTokens` function returns the amount of tokens that an amount of wei corresponds to, as follows:

`(wei * exchange_rate)/ wei_to_cogs)`

claimRefund

```
    function claimRefund() nonReentrant external {
        require(isFinalized);
        require(!goalReached());
        vault.refund(msg.sender);
    }
```

The `claimRefund` function allows contributors to claim a refund if the goal is not reached. It can only be called if the crowdsale is finalized but the goal has not been reached.

claimUnsold

```
    function claimUnsold() onlyOwner {
        require(endTime <= getBlockTimestamp());
```

```
        uint256 unsold = token.balanceOf(this);

        if (unsold > 0) {
            require(token.transferTokens(msg.sender, unsold));
        }
    }
```

The `claimUnsold` function allows the contract owner to claim any unsold tokens after the crowdsale ends.

It transfers the token balance of the crowdsale cotract to the caller (onlyOwner) of this function.

updateWhitelist

```
    function updateWhitelist(address[] addresses, bool status) public
        onlyOwner {
        for (uint256 i = 0; i < addresses.length; i++) {
            address contributorAddress = addresses[i];
            whitelist[contributorAddress] = status;
        }
    }
```

The `updateWhitelist` function allows the contract owner to update the participant whitelist. It takes an array of addresses and a status boolean, and applies the status to each address in the array.

finalize

```
    function finalize() onlyOwner {
        require(!isFinalized);
        require(hasEnded());

        if (goalReached()) {
            //Close the vault
            vault.close();
            //Unpause the token
            token.unpause();
        } else {
            //else enable refunds
            vault.enableRefunds();
        }
        //update the sate of isFinalized
```

```
15          isFinalized = true;
16          //trigger and emit the event of finalization
17          Finalized();
18      }
```

The `finalize` function allows the crowdsale owner to finalize the sale once the sale end time has elapsed.

If the goal has been reached, the vault is closed and the tokens are `unpaused`, i.e. made available for transfer.

If the goal was not reached, refunds are enabled on the vault.

The `isFinalized` variable is set to true, ensuring the function can not be called again, and a `Finalize` event is emitted on successful execution.

forwardFunds

```
1       function forwardFunds(uint256 weiAmount) internal {
2           vault.deposit.value(weiAmount)(msg.sender);
3       }
```

The `forwardFunds` function is an internal function called by `buyTokens` which deposits the given amount of wei in the vault address.

hasEnded

```
1       function hasEnded() public constant returns (bool) {
2           bool passedEndTime = getBlockTimestamp() > endTime;
3           return passedEndTime || capReached();
4       }
```

The `hasEnded` function is a read-only function that returns true if the block time is greater than the crowdsale's end time or if the crowdsale's cap has been reached, and false otherwise.

capReached

```
1       function capReached() public constant returns (bool) {
2           return weiRaised >= cap;
3       }
```

The `capReached` function is a read-only function that returns true if the amount of wei raised so far is equal to or greater than the cap.

goalReached

```
    function goalReached() public constant returns (bool) {
        return weiRaised >= goal;
    }
```

The `goalReached` function is a read-only function that returns true if the amount of wei raised so far is equal to or greater than the funding goal.

isWhitelisted

```
    function isWhitelisted(address contributor) public constant returns (
        bool) {
        return whitelist[contributor];
    }
```

The `isWhitelisted` function returns the whitelisted state of an address by checking it against the `whitelist` map.

validPurchase

```
    function validPurchase() internal constant returns (bool) {
        bool withinPeriod = getBlockTimestamp() >= startTime &&
            getBlockTimestamp() <= endTime;
        bool nonZeroPurchase = msg.value != 0;
        bool capNotReached = weiRaised < cap;
        return withinPeriod && nonZeroPurchase && capNotReached;
    }
```

The `validPurchase` function returns true if a transaction is eligible for participation, if and only if all of the below conditions are true:

- The time is between the crowdsale's start and end time
- The amount sent is greater than 1 wei
- The crowdsale cap has not yet been reached

getBlockTimestamp

```
    function getBlockTimestamp() internal constant returns (uint256) {
        return block.timestamp;
    }
```

The `getBlockTimestamp` function is simply an "alias" for block.timestamp, which is also known as `now`.

SingularityNetToken.sol

The `SingularityNet` token is an ERC20 token with `pausable` and `burnable` functionality:

```
contract SingularityNetToken is PausableToken, BurnableToken
```

Constructor

```
    function SingularityNetToken() {
        totalSupply = INITIAL_SUPPLY;
        balances[msg.sender] = INITIAL_SUPPLY;
    }
```

The contract's constructor sets the initial supply, and assigns all tokens to the creator of the contract.

setOwnership

```
    function setOwnership(address _owner) onlyOwner {
        require(_owner != owner);
        require(address(_owner) != address(0));
        pause();
        //assign to current owner
        balances[owner] = INITIAL_SUPPLY.sub(PUBLIC_SUPPLY);
        transferOwnership(_owner);
        require(_owner == owner);
        balances[owner] = PUBLIC_SUPPLY;
    }
```

The `setOwnership` function can be called by the contract owner to transfer ownership to a new address, along with transferring the public supply of tokens to the new owner.

When this function is called, the token contract is paused such that no transfers can take place until the tokens are unpaused.

Issues

This function can be used to effectively mint infinite tokens, as the previous owner's balance is always set to INITIAL_SUPPLY - PUBLIC SUPPLY unconditionally.

transferTokens

```
1     function transferTokens(address beneficiary, uint256 amount) onlyOwner
          returns (bool) {
2         require(amount > 0);
3
4         balances[owner] = balances[owner].sub(amount);
5         balances[beneficiary] = balances[beneficiary].add(amount);
6         Transfer(owner, beneficiary, amount);
7
8         return true;
9     }
```

The `transferTokens` function allows the contract owner to transfer tokens from the owner's balance to an arbitrary beneficary even if the token contract is paused.

The function emits a `Transfer` event on success.

Disclaimer

This audit concerns only the correctness of the Smart Contracts listed, and is not to be taken as an endorsement of the platform, team, or company.

Audit Attestation

This audit has been signed by the key provided on https://keybase.io/mattdf - and the signature is available on https://github.com/mattdf/audits/