GUTS Token Sale Audit

AUTHOR: MATTHEW DI FERRANTE

2017-10-22

Audited Material Summary

The audit consists of the following contracts:

- GetCrowdsale.sol
- GetFinalizeAgent.sol
- GetPreCrowdsale.sol
- GetPreFinalizeAgent.sol
- GetPrePricingStrategy.sol
- GetPricingStrategy.sol
- GetToken.sol
- GetWhitelist.sol

The git commit hash of the reviewed files is `cec35f217108f60eb50a6c607245c4494c546293`.

Description

The GUTS Token sale is implemented through TokenMarket's ICO contracts. The main crowdsale logic for these contracts is inherited from Crowdsale.sol + MintedTokenCappedCrowdsale.sol, both GetCrowdsale.sol and GetPreCrowdsale.sol extend the functionality and set some state variables.

The Presale and Crowdsale are both structured to use the same token, but different smart contracts (GetPreCrowdsale and GetCrowdsale respectively).

When the presale finishes, a finalization agent set when GetCrowdsale.sol is deployed calls `logPresaleResults` to migrate the results of the presale into the main Crowdsale.

The TokenMarket contracts have been used in a variety of other ICOs and have been previously audited:

- Civic
- Storj
- Monaco
- DENT
- Bitquence
- InsureX

See the following repository for a breakdown of the TokenMarket contracts: https://github.com/TokenMarketNet/ico

This audit focuses on GUTS' specific usage of the TokenMarket tokens.

Security summary

The contracts are correctly constructed have no major security issues that pose a threat to users or their balances.

GetCrowdsale.sol

GetCrowdsale is an implementation of MintedTokenCappedCrowdsale from the TokenMarket contracts, and inherits from it directly:

```
contract GetCrowdsale is MintedTokenCappedCrowdsale
```

Security note

The only recommendation I have here is, due to the presale being a different contract, the `invest` should not be available until `logPresaleResults` has been called, to ensure that there are no numerical conflicts if it's done during the Crowdsale, and also ensuring Crowdsale participants can see how much was raised in the presale through the contract before investing.

Constructor

```
function GetCrowdsale(
    uint _lockTime, FinalizeAgent _presaleFinalizeAgent,
    address _token, PricingStrategy _pricingStrategy, address
        _multisigWallet,
    uint _start, uint _end, uint _minimumFundingGoal, uint
        _maximumSellableTokens)

    MintedTokenCappedCrowdsale(_token, _pricingStrategy, _multisigWallet,
         _start, _end, _minimumFundingGoal, _maximumSellableTokens)
{
    require(_presaleFinalizeAgent.isSane());
    require(_lockTime > 0);
    lockTime = _lockTime;
    presaleFinalizeAgent = _presaleFinalizeAgent;
}
```

The constructor passes variables through to parent constructor of `MintedTokenCappedCrowdsale`, and does sanity checking on the presale agent and lock time variables, and commits the presale agent to storage.

logPresaleResults

```
    function logPresaleResults(uint tokenAmount, uint weiAmount) returns (
        bool) {

        require(msg.sender == address(presaleFinalizeAgent));
        weiRaised = weiRaised.plus(weiAmount);
        tokensSold = tokensSold.plus(tokenAmount);
        presaleWeiRaised = presaleWeiRaised.plus(weiAmount);

        PresaleUpdated(weiAmount, tokenAmount);
        return true;
    }
```

The `logPresaleResults` function records presale token and ether allocation in the crowdsale contract and emits a `PresaleUpdated` event.

It can only be called by `presaleFinalizeAgent`.

All arithmetic operations use SafeMath.

preallocate

```
    function preallocate(address receiver, uint fullTokens, uint weiPrice)
         public onlyOwner {

        uint tokenAmount = fullTokens * 10**token.decimals();
        uint weiAmount = weiPrice * fullTokens; // This can be also 0, we
            give out tokens for free

        weiRaised = weiRaised.plus(weiAmount);
        tokensSold = tokensSold.plus(tokenAmount);

        presaleWeiRaised = presaleWeiRaised.plus(weiAmount);

        investedAmountOf[receiver] = investedAmountOf[receiver].plus(
            weiAmount);
```

```
12          tokenAmountOf[receiver] = tokenAmountOf[receiver].plus(tokenAmount
                );
13
14          assignTokens(receiver, tokenAmount);
15
16          // Tell us invest was success
17          Invested(receiver, weiAmount, tokenAmount, 0);
18      }
```

The `preallocate` function preallocates tokens to an address, and does some accounting to keep the `weiRaised`, `tokensSold` and `presaleWeiRaised` variables consistent.

The `investedAmountOf` and `tokenAmountOf` maps for the receiver are also updated for the number of wei raised and number of tokens allocated, respectively.

The tokens are assigned by calling the internal function `assignTokens`, and on success an `Invested` event is emitted.

Note: The `tokenAmount` and `weiAmount` calculations do not use SafeMath.

setEarlyParicipants

```
1       function setEarlyParicipantWhitelist(address addr, bool status)
            onlyOwner {
2           // We don't need this function, we have external whitelist
3           revert();
4       }
```

The `setEarlyParticipants` function is unused due to the whitelist functionality, therefore Crowdsale.sol's implementation is overridden here with a `revert()`.

The function can only be called by the owner of GetCrowdsale, but it does nothing either way.

assignTokens

```
1       function assignTokens(address receiver, uint tokenAmount) private {
2           MintableToken mintableToken = MintableToken(token);
3           mintableToken.mint(receiver, tokenAmount);
4       }
```

The `assignTokens` function allocates an amount of tokens to an address. It is a private function and cannot be called directly, and is only ever called by the `preallocate` function.

This function assumes `token` implements a MintableToken interface and typecasts the more generic `token` variable which is a `FractionalERC20` type as defined in Crowdsale.sol.

finalize

```
    function finalize() public inState(State.Success) onlyOwner
        stopInEmergency {
        require(now > endsAt + lockTime);
        super.finalize();
    }
```

The `finalize` function ensures that the current time is past the crowdsale end date + the lock time, and calls Crowdsale.sol's parent implementation of `finalize`.

This function can only be called by the owner of GetCrowdsale, and cannot be called if the contract is halted.

This function can only succeed if the crowdsale is in the `State.Success` stage.

Default Function

```
    function() payable {
        invest(msg.sender);
    }
```

The default function in this contract is merely a passthrough to Crowdsale.sol's invest function, passing `msg.sender` as the argument.

GetToken.sol

The `GetToken` contract is a very simple initializer for the underlying `CrowdsaleToken` + `BurnableToken` implementation:

```
contract GetToken is CrowdsaleToken, BurnableToken {
    function GetToken() CrowdsaleToken(
            "Guaranteed Entrance Token",
            "GET",
            0,  // We don't want to have initial supply
            18,
```

```
            true // Mintable
        )
    {}
}
```

It takes no arguments and simply passes some hardcoded variables to the inherited `CrowdsaleToken` constructor.

The full inheritance chain makes `GetToken` contain the following functionality:

- ReleasableToken: ERC20 + transfer freeze / unfreeze functionality, transfers frozen by default
- MintableToken: ERC20 + mint functionality, where a mintAgent can create new tokens, with minting permanently disabled once the tokens are "released"
- BurnableToken: ERC20 + burn functionality, which allows a token holder (contract or account) to burn their own tokens
- UpgradeableToken: ERC20 + upgrade functionality, where an upgrade agent can set a new revision for the token contract and users can opt-in to have their balances transferred to the new contract.

Security note

The inheritance hierarchy of the TokenMarket contracts is non-trivial - any additional function overrides should be triple checked in case the function is implemented by more than one parent, as solidity's super call order is somewhat unintuitive.

GetPreCrowdsale.sol

`GetPreCrowdsale` inherits from `MintedTokenCappedCrowdsale`, just like `GetCrowdsale`, but lacks any additional functions beyond the override function to `setEarlyParticipantsWhitelist`.

```
contract GetPreCrowdsale is MintedTokenCappedCrowdsale
```

Constructor

```
    function GetPreCrowdsale(
        address _token, PricingStrategy _pricingStrategy, address
           _multisigWallet,
        uint _start, uint _end, uint _maximumSellableTokens)
```

```
        MintedTokenCappedCrowdsale(_token, _pricingStrategy,
            _multisigWallet,
            _start, _end, 0, _maximumSellableTokens)
    {
    }
```

The constructor for `GetPreCrowdsale` simply passes all of its arguments to the parent constructor of `MintedTokenCappedCrowdsale`, and implements no additional logic.

Default Function

Just like in `GetCrowdsale`, the default function for the presale is simply a `msg.sender` passthrough to `invest`, and is the only way to participate in the sale.

GetWhitelist.sol

The `GetWhitelist` contract is the only contract that is mostly written from scratch, inheriting only `Ownable` from Zeppelin and using `SafeMath` as a library:

```
contract GetWhitelist is Ownable
```

It implements both whitelisting functionality and tranche allowances per investor.

Constructor

```
    function GetWhitelist(uint _presaleCap, uint _tier1Cap, uint _tier2Cap
        , uint _tier3Cap, uint _tier4Cap) {
        presaleCap = _presaleCap;
        tier1Cap = _tier1Cap;
        tier2Cap = _tier2Cap;
        tier3Cap = _tier3Cap;
        tier4Cap = _tier4Cap;
    }
```

The constructor simply stores the supplied caps to their respective storage variables.

isGetWhiteList

```
    function isGetWhiteList() constant returns (bool) {
        return true;
    }
```

The `isGetWhiteList` function is used by calling contracts to check whether an address implements the `GetWhitelist` interface.

acceptBatched

```
    function acceptBatched(address[] _addresses, bool _isEarly)
       onlyWhitelister {
        // trying to save up some gas here
        uint _presaleCap;
        if (_isEarly) {
            _presaleCap = presaleCap;
        } else {
            _presaleCap = 0;
        }
        for (uint i=0; i<_addresses.length; i++) {
            entries[_addresses[i]] = WhitelistInfo(
                _presaleCap,
                tier1Cap,
                tier2Cap,
                tier3Cap,
                tier4Cap,
                true
            );
        }
        NewBatch();
    }
```

The `acceptBatched` function takes a list of addresses and a boolean specifying whether the addresses are allowed to participate in the crowdsale, and initializes the structure for each address in the `entries` map with the default caps. It emits a `NewBatch` event when successful.

This function can only be called by an address in the set of whitelisters.

accept

```
    function accept(address _address, bool _isEarly) onlyWhitelister {
        require(!entries[_address].isWhitelisted);
        uint _presaleCap;
        if (_isEarly) {
            _presaleCap = presaleCap;
        } else {
            _presaleCap = 0;
        }
        entries[_address] = WhitelistInfo(_presaleCap, tier1Cap, tier2Cap,
            tier3Cap, tier4Cap, true);
        NewEntry(_address);
    }
```

The `accept` function is single-address version of `acceptBatch`, but otherwise functionally equivalent other than refusing to run if the address is already whitelisted.

It can only be called by an address in the set of whitelisters.

subtractAmount

```
    function subtractAmount(address _address, uint _tier, uint _amount)
       onlyWhitelister {
        require(_amount > 0);
        require(entries[_address].isWhitelisted);
        if (_tier == 0) {
            entries[_address].presaleAmount = entries[_address].
               presaleAmount.minus(_amount);
            EdittedEntry(_address, 0);
            return;
        }else if (_tier == 1) {
            entries[_address].tier1Amount = entries[_address].tier1Amount.
               minus(_amount);
            EdittedEntry(_address, 1);
            return;
        }else if (_tier == 2) {
            entries[_address].tier2Amount = entries[_address].tier2Amount.
               minus(_amount);
            EdittedEntry(_address, 2);
            return;
        }else if (_tier == 3) {
```

```
            entries[_address].tier3Amount = entries[_address].tier3Amount.
                minus(_amount);
            EdittedEntry(_address, 3);
            return;
        }else if (_tier == 4) {
            entries[_address].tier4Amount = entries[_address].tier4Amount.
                minus(_amount);
            EdittedEntry(_address, 4);
            return;
        }
        revert();
    }
```

The `subtractAmount` function ensures that an address is in the whitelist (otherwise it throws), and subtracts their allowance for the specified tranche accordingly. Due to the SafeMath usage underflows cause the contract to throw. If the tranche is beyond 4 it will also throw.

It can only be called by an address in the set of whitelisters.

setWhitelister

```
    function setWhitelister(address _whitelister, bool _isWhitelister)
        onlyOwner {
        whitelisters[_whitelister] = _isWhitelister;
        WhitelisterChange(_whitelister, _isWhitelister);
    }
```

The `setWhitelister` function allows the contract owner to add or remove an address from the set of whitelisters.

It can only be called by the contract owner.

setCaps

```
    function setCaps(uint _presaleCap, uint _tier1Cap, uint _tier2Cap,
        uint _tier3Cap, uint _tier4Cap) onlyOwner {
        presaleCap = _presaleCap;
        tier1Cap = _tier1Cap;
        tier2Cap = _tier2Cap;
        tier3Cap = _tier3Cap;
        tier4Cap = _tier4Cap;
```

```
    }
```

The `setCaps` function allows the contract owner to set or update the tranche caps for the crowdsales.

It can only be called by the contract owner.

Default Function

```
    function() payable {
        revert();
    }
```

The default function simply throws, to prevent payment to this contract.

GetPricingStrategy.sol

The `GetPricingStrategy` contract sets the pricing mechanism for `GetCrowdsale`. It implements a tranche based pricing strategy, inheriting from `EthTranchePricing`:

```
contract GetPricingStrategy is EthTranchePricing
```

It also checks any incoming investor addresses against a whitelist and throws if the investor is not whitelisted.

Constructor

```
    function GetPricingStrategy(GetWhitelist _whitelist, uint[] _tranches)
        EthTranchePricing(_tranches) {
        assert(_whitelist.isGetWhiteList());
        whitelist = _whitelist;
    }
```

The constructor takes a `GetWhitelist` contract address that it uses ensure purchasing addresses are whitelisted, and enforces tiered caps for each investing address.

The parent constructor for `EthTrancePricing` is called with the list of tranches as an argument.

isPresalePurchase

```
    function isPresalePurchase(address purchaser) public constant returns
        (bool) {
        return false;
    }
```

The `isPresalePurchase` function returns false, as the tranche pricing strategy is only for the main Crowdsale.

setCrowdsale

```
    function setCrowdsale(address _crowdsale) onlyOwner {
        require(_crowdsale != 0);
        crowdsale = _crowdsale;
    }
```

The `setCrowdsale` function allows the contract owner to set the address of the crowdsale. The function sanity checks that the supplied address is not 0.

isSane

```
    function isSane(address _crowdsale) public constant returns (bool) {
        return crowdsale == _crowdsale;
    }
```

The `isSane` function returns true if the supplied address matches the address set as the crowdsale in the storage state, and false otherwise.

getCurrentTrancheIndex

```
    function getCurrentTrancheIndex(uint weiRaised) public constant
        returns (uint) {
        uint i;

        for(i=0; i < tranches.length; i++) {
            if(weiRaised < tranches[i].amount) {
                return i-1;
            }
```

```
        }
    }
```

The `getCurrentTrancheIndex` function iterates through the tranches supplied at deployment time and returns the last tranche that closes the `weiRaised` amount.

Security note

The counter `i` in this function can underflow in case `weiRaised` is 0 at Crowdsale time.

calculatePrice

```
    function calculatePrice(uint value, uint weiRaised, uint tokensSold,
       address msgSender, uint decimals) public constant returns (uint) {
        require(msg.sender == crowdsale);
        uint amount;
        bool isEarly;
        bool isWhitelisted;
        uint trancheIndex = getCurrentTrancheIndex(weiRaised);
        whitelist.subtractAmount(msgSender, trancheIndex + 1, value);

        uint multiplier = 10 ** decimals;
        return value.times(multiplier) / tranches[trancheIndex].price;
    }
```

The `calculatePrice` function is used by Crowdsale.sol to calculate the tokens received for an investment.

Based on the amount of wei raised so far, it retrieves the corresponding tranche and subtracts the amount from the sender's allowance in that tranche.

It then returns `weiInvested*10^18 / current_tranche_price` as the amount of tokens to allocate.

Security note

The division by tranche price does not use SafeMath.

`isEarly`, `isWhitelisted` and `amount` are not used and should be removed for code clarity.

Default Function

The default function in this contract simply throws.

GetPrePricingStrategy.sol

The `GetPrePricingStrategy` contract is almost the same as the `GetPricingStrategy` contract, except for it implements a flat pricing mechanism instead of tranche pricing:

```
contract GetPrePricingStrategy is FlatPricing, Ownable
```

In the interest of brevity I will only analyze the functions that are not functionally equivalent to `GetPricingStrategy`.

Constructor

```
    function GetPrePricingStrategy(GetWhitelist _whitelist, uint
        _oneTokenInWei) FlatPricing(_oneTokenInWei) {
        assert(_whitelist.isGetWhiteList());
        whitelist = _whitelist;
    }
```

The constructor simply assigns the whitelist like in `GetPricingStrategy`, and calls the parent constructor `FlatPricing`, taking an argument for the amount one token should cost, denominated in wei.

calculatePrice

```
    function calculatePrice(uint value, uint weiRaised, uint tokensSold,
        address msgSender, uint decimals) public constant returns (uint) {
        // only precrowdsale can call this
        require(msg.sender == precrowdsale);
        // 0 is the presale tier.
        whitelist.subtractAmount(msgSender, 0, value);
        return super.calculatePrice(value, weiRaised, tokensSold,
            msgSender, decimals);
    }
```

The `calculatePrice` function, just like in `GetPriceStrategy`, subtracts allowance for the sender from the whitelist, and then defers the price calculation to the parent `calculatePrice` implementation in `FlatPricing`, which simply returns:

```
weiInvested*10^18 / token_cost_in_wei
```

GetFinalizeAgent.sol

The `GetFinalizeAgent` contract extends a `FinalizeAgent` interface which mints reserve tokens and finalizes the crowdsale, releasing the tokens and making them available for transfer and use..

```
contract GetFinalizeAgent is FinalizeAgent
```

Constructor

```
    function GetFinalizeAgent(CrowdsaleToken _token, Crowdsale _crowdsale,
            address _userGrowthMultisig, address _stabilityMultisig,
               address _bountyMultisig) {
        token = _token;
        crowdsale = _crowdsale;
        if(address(crowdsale) == 0) {
            revert();
        }

        require(_userGrowthMultisig != 0);
        require(_stabilityMultisig != 0);
        require(_bountyMultisig != 0);

        userGrowthMultisig = _userGrowthMultisig;
        stabilityMultisig = _stabilityMultisig;
        bountyMultisig = _bountyMultisig;
    }
```

The constructor does some sanity checking on arguments and assigns them to their respective storage variables.

isSane

```
    function isSane() public constant returns (bool) {
        return (token.mintAgents(address(this)) == true) && (token.
            releaseAgent() == address(this));
    }
```

The `isSane` function returns true if the GetFinalizeAgent is one of the token's mintAgents and also set as the token's releaseAgent.

finalizeCrowdsale

```
    function finalizeCrowdsale() {
        if(msg.sender != address(crowdsale)) {
            revert();
        }

        uint tokensSold = crowdsale.tokensSold();
        uint decimals = token.decimals();

        // maximum digits here (10 + 18 + 12)
        token.mint(userGrowthMultisig, tokensSold.times(73170731707) /
            100000000000);

        token.mint(stabilityMultisig, 12600000 * (10**decimals));
        token.mint(bountyMultisig, 1800000 * (10**decimals));

        // Make token transferable
        token.releaseTokenTransfer();
    }
```

The `finalizeCrowdsale` function mints tokens for user growth, stability and bounty multisigs, and releases the tokens, disabling minting.

Security note

Would be better to have constant variables instead of magic numbers hardcoded inline for token issuance.

Default Function

```
    function() payable {
        revert();
    }
```

The default function just prevents payments to this contract.

GetPreFinalizeAgent.sol

The `GetPreFinalizeAgent` contract is the same functionally as `GetFinalizeAgent`, except it is meant to finalize the presale.

It inherits from `FinalizeAgent` and `Ownable`:

```
contract GetPreFinalizeAgent is FinalizeAgent, Ownable
```

Constructor

```
    function GetPreFinalizeAgent(GetCrowdsale _preCrowdsale) {

        if(address(_preCrowdsale) == 0) {
            revert();
        }
        preCrowdsale = _preCrowdsale;

    }
```

The constructor only assigns the crowdsale address to the storage variable afterr ensuring it isn't 0.

setCrowdsale

```
    function setCrowdsale(GetCrowdsale _crowdsale) onlyOwner {
        require(address(_crowdsale) != 0);
        crowdsale = _crowdsale;
    }
```

The `setCrowdsale` function allows the contract owner to set the address for the crowdsale contract.

isSane

```
    function isSane() public constant returns (bool) {
        // cannot check crowdsale yet since it is not set.
        return true;
    }
```

The `isSane` function is a dummy function that always returns true.

finalizeCrowdsale

```
    function finalizeCrowdsale() {
        if(msg.sender != address(preCrowdsale)) {
            revert();
        }

        // log the results to the main crowdsale
        uint tokensSold = preCrowdsale.tokensSold();
        uint weiRaised = preCrowdsale.weiRaised();
        if (!crowdsale.logPresaleResults(tokensSold, weiRaised)) {
            revert();
        }
    }
```

The `finalizeCrowdsale` function finalizes the presale by retrieving the tokens sold and wei raised from the presale contract and calling `logPresaleResults` on the crowdsale contract, to migrate the accounting over.

It can only be called by the presale contract.

Default Function

```
    function() payable {
        revert();
    }
```

The default function prevents any ether from being sent to this contract.

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/