BCShop Crowdsale Audit

AUTHOR: MATTHEW DI FERRANTE

2017-10-07

Audited Material Summary

The audit covers the BCShop.io crowdsale contracts, which implements an investor-restricted sale for an ERC20 Token with various extensions.

The audited material consists of:

- /contracts/token/BCSToken.sol
- /contracts/token/TokenPool.sol
- /contracts/crowdsale/BCSCrowdsale.sol
- /contracts/crowdsale/ParticipantInvestRestrictions.sol
- /contracts/crowdsale/TrancheWallet.sol

… and their dependencies.

The git commit of the audited material is `8d0d68b` for repo https://github.com/bcshop-io/bcshop.io.

BCSToken.sol

BCSToken implements an ERC20 Token with some extra utility functions that allow the contract manager to lock/unlock transfers for the entire token or specific participants, and allows the contract manager to burn the tokens they own.

The contract inherits from `ValueToken`, `ReturnableToken`, and `IBurnableToken` directly:

```
1 contract BCSToken is ValueToken, ReturnableToken, IBurnableToken
```

The indirect inheritance chain, in order, is:

- ValueToken
  - Manageable
    - Owned
  - ERC20StandardToken
    - IERC20Token
    - SafeMath
- ReturnableToken
  - Manageable
    - Owned
- ERC20StandardToken - IERC20Token - SafeMath
- IBurnableToken

The callchain for doTransfer is as follows:

ECSToken.doTransfer -> ReturnableToken.doTransfer -> ValueToken.doTransfer -> ERC20StandardToken.doTransfer

Security Issues

There aren't any code level issues in this contract, but the inheritance chain in this contract is non-trivial and it would be better to coalesce ReturnableToken and ValueToken into one contract.

ValueToken.sol

ValueToken wraps an ERC20 Token with "reserve" functionality, such that if an address is part of the reserve, transferring tokens out of it reduces the `reservedAmount`, and transferring tokens into a reserve associated address increases `reservedAmount`.

The contract inheritance is as specified ECSToken.sol's breakdown.

The contract has a `valueAgent` contract variable, which if set, executes `tokenIsBeingTransferred` as a preexecution hook in `doTransfer`. It also has a preexecution hook of `valueAgent.tokenChanged` in setReserved.

The related contract `ValueAgent.sol` is straightforward and merely implements a modifier, constructor and interface.

Security issues

None, SafeMath is used throughout.

ReturnableToken.sol

ReturnableToken wraps an ERC20 Token with "return agent" functionality. As a post-execution hook in doTransfer it checks if the target address implements a return agent (that is, it implements a `returnToken` method), and if so, executes `ReturnTokenAgent(_to).returnToken(_from, _value);`.

The contract manager is able to set or remove return agent address associations.

The contract inheritance is as specified in ECSToken.sol's breakdown.

The related contract `ReturnAgent.sol` is straightforward and merely implements a modifier, constructor and interface, plus two manager functions `setReturnableToken` and `removeReturnableToken`.

Security Issues

None, the contract is straightforward, though there is some unnecessary duplication between `ReturnableToken` and `ReturnAgent`.

ERC20StandardToken.sol

ERC20StandardContract implements a standard ERC20 Token, with SafeMath in all arithmetic operations.

Security Issues

None.

BCSCrowdsale.sol & TokenPool.sol

The BCSCrowdsale contract implements the main crowdsale logic, and supports a token pool of ERC20 Tokens instead of importing BCSToken directly. The term "pool" is somewhat misleading as the contract only supports returning one token anyway. The associated TokenPool contract simply sets the BCSCrowdsale as the trustee and approves it to manage its balance.

It inherits from ICrowdsaleFormula, Manageable and SafeMath directly:

```
1 contract BCSCrowdsale is ICrowdsaleFormula, Manageable, SafeMath
```

It also uses `ITokenPool` and `IInvestRestrictions` interfaces for internal variables.

The crowdsale mainly operates through the `invest` function, which takes Ether and, if the crowdsale is active, and investor eligible/not restricted, allocates tokens from the `tokenPool` to the sender of the transaction.

One major deviation is that if excess Ether is sent, such that not enough tokens are available to be allocated, it is not returned by default but rather deposited and tracked through an `overpays` map, and the investor must then call `withdrawOverpay` to retrieve their funds.

It also allows investors to get a refund if the crowdsale "fails", i.e. if the contract owner calls the `makeFail` function in case of an emergency.

Security Issues / Suggestions

While I don't see any real security issues, the level of indirection in this contract is unncessary and wastes gas. There's no reason to define getter functions like `getCurrentBonusPct` and `amountToBeneficiary` for simple variables. It would also be better to use SafeMath consistently and throughout the contract, in functions like `howManyTokensForEther` and not just in the balance affecting functions.

Unless the code is written as a library and *meant* to be reusable, there is also no need to define interfaces for everything like `IInvestorRestrictions` and instead just inherit the contract with the actual implementation so that the flow is easier to audit and follow. Less indirection also makes the deployment process less prone to mistakes.

In terms of gas costs it is also advisable to reduce the number of storage variables used and not use them for intermediary calculation.

ParticipantInvestRestrictions & FloorInvestRestrictions.sol

These two contracts are used in BCSCrowdsale to apply two main restrictions to the token sale process:

- Maximum number of investors
- Reserves tokens such that investors not in the reserved list can only buy up to totalTokens - tokensReserved, and only if there are still free places.

The manager of this contract is ECSCrowdsale. The only non-overridden function from FloorInvestRestrictions is `changeFloor`.

The only functions actually called by BCSCrowdsale are `canInvest` and `investHappened`, all other functions must be called manually by one of the managers.

The contract is structured to limit the number of investors for the Pre-ICO, who must pass some soft KYC.

Security Issues

The "free places" can all be taken up by one person if they split their investments across multiple addresses. I recommend filling up the `reservedInvestors` structure with as many places as possible to prevent this from being a major issue.

TrancheWallet.sol

TrancheWallet implements a wallet that allows withdrawal in “tranches”, that is, a limited amount for each epoch.

It is relatively self contained, only inheriting a single function interface from `LockableWallet`, which itself inherits only from `Manageable`:

```
1 contract TrancheWallet is LockableWallet
```

The wallet can be locked for a certain number of days by the manager, after which only the amount permitted in the current tranche can be withdrawn until the lock expires.

The first tranche is immediately withdrawable as the `periodsSinceLock - tranchesSent + 1;` will always return 1 even if `periodsSinceLock` and `tranchesSent` are 0.

If there is an amount available to withdraw, it will be send to the beneficiary address which is set on contract creation.

Security Issues

No critical issues, but if funds are accidentally sent to this contract through the default payable function after the wallet is “locked”, the user will need to wait until the lock expires to withdraw them, because they will not be accounted for in the tranche calculation due to the fact that it uses `initialFunds` to figure out the amount to unlock.

Owned.sol and Manageable.sol

Owned and Manageable implement the standard Owner pattern and an extension to it which allows the owner to add multiple “managers” to the contract, respectively.

Security Issues

None, the contracts are pretty standard in functionality.

Disclaimer

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

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/