PUBLIC

QIIBEE Security Audit

of QBX TOKEN Smart Contracts

June 28, 2018

Produced for

by

Table Of Content

Foreword . . . 1
Executive Summary . . . 1
System Overview . . . 2
Token Sale Overview . . . 2
Audit Overview . . . 3
Scope of the Audit . . . 3
Depth of Audit . . . 3
Terminology . . . 3
Limitations . . . 5
Details of the Findings . . . 6
Security Issues . . . 7
Unchecked arithmetic operations L ✓ Fixed . . . 7
Unbounded iteration M ✓ Addressed . . . 7
Bonus stealing is possible M ✓ Addressed . . . 7
`_checkLimits` ignores deposits M ✓ Fixed . . . 8
`_mintTokens` is public C ✓ Fixed . . . 8
Trust Issues . . . 9
Token can be changed by owner H ✓ Fixed . . . 9
Design Issues . . . 10
Inheritance from the OpenZeppelin library is poorly done M ✓ Acknowledged . . . 10
Recommendations / Suggestions . . . 11
Disclaimer . . . 12

Foreword

We first and foremost thank QIIBEE for giving us the opportunity to audit their smart contracts. This documents outlines our methodology, limitations, and results.

– ChainSecurity

Executive Summary

The QIIBEE smart contracts, specifically the Crowdsale and Vault contracts, have been analyzed under different aspects, with a variety of automated security analysis tools and by manual expert review.
Overall, we found that QIIBEE has put efforts into extensive documentation of their codebase.
We discovered several issues which needed to be fixed. QIIBEE has fixed all security issues and most of the other issues raised during the audit. No further concern remains regarding the security of QIIBEE's smart contracts.

System Overview

Token Name & Symbol	QBX TOKEN, QBX
Decimals	18 Decimals
Phases	Bonus, Normal
Exchange Rate	Variable
Refund	Yes
Soft and Hard Cap	To be defined
Minimum and Maximum contribution	To be defined
Token Type	ERC 20
Token Generation	Mintable
Token Amount	49% of total supply
Pausable	Yes
KYC	Off-chain

Table 1: Facts about the QBX token and the Token Sale.

In the following we describe the QBX TOKEN (QBX) and its corresponding Token Sale. The table above gives the general overview.

We note that the crowdsale and its properties summarized in this section hold true under the assumption that the address of the token contract linked to the QIIBEE crowdsale and vault indeed corresponds to the QBX TOKEN ERC20 contract, since it is not within the audit scope.

Token Sale Overview

The Token Sale of the QBX TOKEN will proceed within a predefined time range. The first seven days of the crowdsale serve as a bonus phase, during which any approved investor who passed the KYC procedure will obtain a 5% bonus on his QBX purchase.

An investor's individual as well as the total contribution will be capped during the sale. While purchases during the crowdsale are open to anyone, approval needs to be granted by QIIBEE in order for the buyer to obtain his QBX tokens. Until this happens, a contributor's funds are locked in a vault owned by QIIBEE. Once approval is given, the investor receives his QBX tokens and his funds are transferred from the vault to QIIBEE's wallet.

If a buyer does not pass the KYC procedure, his investment, which is locked up in the vault, will be refunded. The same applies for an investor contributing above the individual limit, meaning that the difference between the contribution and limit will be refunded.

Audit Overview

Scope of the Audit

The scope of the audit is limited to the following source code files. All of these source code files were initially received on May 29, 2018, and updated on May 20, 2018:

File	SHA-256 checksum
QiibeeCrowdsale.sol	9e729510c06c7cad9466532621d0dbe0a03919b5d5802a07e971705b9e384842
Vault.sol	824e153f0de1106521a9be9f723f5d5873f57db3be3eead155d73bc2292fb1f8

The corresponding Git commit is `f189f4d019c4f31a12546b74f9b8226c027e7a47`.

Depth of Audit

The scope of the security audit conducted by ChainSecurity was restricted to:

- Scan the contracts listed above for generic security issues using automated systems and manually inspect the results.
- Manual audit of the contracts listed above for security issues.

Terminology

For the purpose of this audit, we adopt the following terminology. For security vulnerabilities, we specify the *likelihood*, *impact* and *severity* (inspired by the OWASP risk rating methodology[1]).

Likelihood represents the likelihood of a security vulnerability to be encountered or exploited in the wild.

Impact specifies the technical and business related consequences of an exploit.

Severity is derived based on the likelihood and the impact calculated previously.

We categorize the findings into 4 distinct categories, depending on their severities:

- L Low: can be considered as less important
- M Medium: should be fixed
- H High: we strongly suggest to fix it before release
- C Critical: needs to be fixed before release

These severities are derived from the likelihood and the impact using the following table, following a standard approach in risk assessment.

	IMPACT		
LIKELIHOOD	High	Medium	Low
High	C	H	M
Medium	H	M	L
Low	M	L	L

[1] https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology

During the audit concerns might arise or tools might flag certain security issues. If our careful inspection reveals no security impact, we label it as ✓ No Issue. If during the course of the audit process, an issue has been addressed technically, we label it as ✓ Fixed, while if it has been addressed otherwise by improving documentation or further specification, we label it as ✓ Addressed. Finally, if an issue is meant to be fixed in the future without immediate changes to the code, we label it as ✓ Acknowledged.

Findings that are labelled as either ✓ Fixed or ✓ Addressed are resolved and therefore pose no security threat. Their severity is still listed, but just to give the reader a quick overview what kind of issues were found during the audit.

Limitations

Security auditing cannot uncover all existing vulnerabilities, and even an audit in which no vulnerabilities are found is not a guarantee for a secure smart contract. However, auditing allows to discover vulnerabilities that were overlooked during development and areas where additional security measures are necessary.

In most cases, applications are either fully protected against a certain type of attack, or they lack protection against it completely. Some of the issues may affect the entire smart contract application, while some lack protection only in certain areas. We therefore carry out a source code review trying to determine all locations that need to be fixed. Within the customer-determined timeframe, ChainSecurity has performed auditing in order to discover as many vulnerabilities as possible.

Details of the Findings

In this section we detail our results, including both positive and negative findings.

Security Issues

In the following, we discuss our investigation into security issues. Therefore, we highlight whenever we found specific issues but also mention what vulnerability classes do not appear, if relevant.

Unchecked arithmetic operations ✓ Fixed

Arithmetic operations should always be performed with checks to avoid resulting over- or underflows. Ideally, this should be done with the SafeMath [2] open source library. Concretely, the crowdsale contract performs an unchecked addition in the calculation of `bonusEndtime`.

Likelihood: Low
Impact: Medium

Fixed: `bonusEndTime` is now calculated using the `add` function from SafeMath.

Unbounded iteration ✓ Addressed

In the vault contract, the `refundAll` function iterates over the array of all fund owners. Since the length of this array cannot be known, the iteration is potentially unbounded. Assuming a block gas limit of 8 million, which is close to the current limit on the main network [3], 1040 contributors will already exceed it. Hence, if there are 1040 or more contributors, the function will revert and no owner will obtain his refund.

Likelihood: Medium
Impact: Medium

Fixed: Users can now individually get a refund by calling `claimVaultFunds.refundAll` could still cause the contract to exceed the gas limit though, and we think that this function should be removed.

Bonus stealing is possible ✓ Addressed

In the crowdsale contract, QIIBEE already acknowledged the following faulty behaviour:

```
186 /****
187 * TODO: FIX BUG (or let it be): Let's say Alice invests on 1st week, so bonus[Alice] = true but she does not
188 * manage to schedule the KYC call during that week. Then, she invests more on the 2nd week.
189 * Here is when bonus[Alice] is replaced by false.
190 * Later on, she goes through the KYC process (getting accepted) so her funds of the 2
191 * contributions (that are are deposited in the vault) are released but she does not receive
192 * the tokens of the contribution she made during the 1st week.
193 * MANUAL FIX: This situation is quite unlikely to happen but, if it happens, we can manually
194 * distribute the bonus tokens to the contributor afterwards.
195 * SOLUTION: TODO.
196 ****/
```

QiibeeCrowdsale.sol

However this does not cover a scenario in which an attacker can block the bonus of another investor if he is ready to lose a certain amount of funds. Especially if the spread between `minContrib` and `maxCumulativeContrib` is big, this attack becomes more likely.

Assuming an investor deposited some significant amount (i.e. close to `maxCumulativeContrib`) within the first week, in order to receive a high bonus, an attacker can later additionally contribute `minContrib` in the investor's name after the bonus phase. This leads to the investor losing his bonus, which can be a higher amount than the attacker's contribution.

Likelihood: Medium
Impact: Medium

Fixed: `buyTokens` now checks that `msg.sender == beneficiary`, so the attack we described is now impossible. The bug described by Qiibee (see red text above) still remains.

[2] https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/math/SafeMath.sol
[3] Real-time changes can be tracked at https://ethstats.net/

_checkLimits ignores deposits M ✓ Fixed

When checking that an address' contributions do not exceed `maxCumulativeContrib`, `checkLimits` should also take `vault.deposited(beneficiary)` into account. Otherwise, the time period after a user calls `buyTokens` but before the owner calls `validatePurchase` presents a vulnerability. For example: User A calls `buyTokens` with `maxCumulativeContrib` wei. Then, user B calls `buyTokens` with `x` wei, and names user A as beneficiary. Because `checkLimits` doesn't check A's vault, B's call to `buyTokens` also passes. When the owner calls `validatePurchase`, `checkLimits` is called again (because of `mintTokens`) and fails because A's vault contains `maxCumulativeContrib + x` wei. This means that B has successfully blocked A from buying tokens.

Likelihood: Medium
Impact: Medium

Fixed: Because `buyTokens` now requires that `msg.sender == beneficiary`, this vulnerability no longer exists.

_mintTokens is public C ✓ Fixed

The `_mintTokens` function is public and can be called by anyone. This allows an attacker to mint QBX tokens for free, completely bypass the KYC procedure and avoid the `_preValidatePurchase` check, making it possible for the attacker to obtain tokens before the sale opens.

Likelihood: High
Impact: High

Fixed: `_mintTokens` is now `internal`.

Trust Issues

The issues described in this section are not security issues but describe functionality which is not fixed inside the smart contract and hence requires additional trust into QIIBEE.

Token can be changed by owner

The owner of the crowdsale contract can change the token address at any time when he calls the `setToken` function. This allows a malicious owner to sell useless tokens first and only later switch the sale to real ones, or to introduce other issues by swapping addresses during an ongoing sale.

Impact: High

Fixed: `setToken` now uses a `beforeOpen` modifier to ensure that the token can only be changed before `openingTime`. We still think that this solution could be improved, by setting the token address in the constructor and not allowing it to be changed later on.

Design Issues

The points listed here are general recommendations about the design and style of QIIBEE's project. They highlight possible ways for QIIBEE to further improve the code.

Inheritance from the OpenZeppelin library is poorly done

QIIBEE seems to use the OpenZeppelin library, but only uses it partly.

Contracts are inherited, but most of their functions and modifiers are not used. In the library, the documentation mentions that `buyTokens` should not be overridden.

```
/**
 * @dev low level token purchase ***DO NOT OVERRIDE***
 * @param _beneficiary Address performing the token purchase
 */
```

Crowdsale.sol

This function is overridden in QIIBEE case.

The various crowdsale contracts inherited are the following:

- TimedCrowdsale
- CappedCrowdsale
- FinalizableCrowdsale
- Pausable

In those, constraints are often enforced through `_preValidatePurchase`, but this function is overridden in QIIBEE's case. In the case of FinalizableCrowdsale at least, it seems that all functions could have been implemented correctly, by refraining from overriding `finalize`, and overriding only `finalization` instead. This indicates that there is room for improvement in that regard.

We are not arguing here that checks are not in place; where they are lacking, this has been reported in previous sections. Rather, we highlight the fact that some of the previous issues can be traced back to this lack of rigor when reusing the OpenZeppelin library. QIIBEE would benefit from trying to maximize the reuse of the code from the OpenZeppelin library, and minimize the code written specifically for QIIBEE's project.

If the code cannot be reused as it is, then these contracts should not be used at all.

Acknowledged: QIIBEE improved the code, but acknowledges that it is still not ideal in its current state. In particular, the function `buyTokens` is still overridden in QIIBEE's code, even though the OpenZeppelin documentation explicitly says "do not override". CHAINSECURITY thinks that this issue should be solved before any deployment.

Recommendations / Suggestions

- In `_preValidatePurchase`, it is not necessary to check that `_beneficiary != address(0)` because the require statement `require(msg.sender == _beneficiary)` already ensures that this condition is met. Overall, the `beneficiary` argument used in various functions could be removed and replaced by `msg.sender`.
- (Note: This suggestion has now been implemented.) In the vault contract, specifically the `refund` function, the special case `depositedValue > 0` is unnecessary: having an event with a refund of 0 should be considered, just like ERC20 with an amount of 0 still emit events. If this not an option, putting this check in the function where it is used instead, and using an `assert` inside the function could make more sense.
- (Note: This suggestion has now been implemented.) Time constraints in the crowdsale contract are enforced by the `_preValidatePurchase` function, while the cap is enforced using a modifier. This lowers the code readability. More so, the same constraint can be enforced with the `onlyWhileOpen` modifier implemented in the `TimedCrowdSale` from which the QIIBEE crowdsale inherits.
- (Note: This suggestion has now been implemented.) The `_checkBonus` function takes an argument that is never used and can hence be dropped.
- (Note: This suggestion has now been implemented.) The vault contract is not tested except for a simple ownership transfer. Considering the vault holds funds and plays a central role, additional tests should be added.
- (Note: This suggestion has now been implemented.) An investor's bonus eligibility is tested by the function `_checkBonus`, which can be simplified to `return now <= bonusEndTime;`, saving gas and lines of code.
- (Note: This suggestion has now been implemented.) As referenced in the trust section, the `setToken` function allows the owner to change the token while the sale is ongoing. It should be either used during migration, or the address value should be hardcoded.
- (Note: This suggestion has now been implemented.) In `finalization()`, an assert statement should be added to ensure that `foundationSupply` is set correctly, for example `assert(QiibeeToken(token).totalSupply() == totalSupply.add(foundationSupply)`
- Testing and migration should be separated. Because the migration is currently written in the tests, the test cases will be irrelevant if the migration that is used in practice is not exactly copied from the tests. In practice, migration scripts get rewritten from scratch in this case, which indeed makes the test cases less relevant.
- (Note: This suggestion has now been implemented.) The `validatePurchase` function takes a boolean as an argument, which is used to distinguish two cases. Instead, two separate functions can be implemented, corresponding to the two different behaviours.
- (Note: This suggestion has now been implemented.) The `_mintTokens` function is fairly complex and can be split it into smaller, separate functions to improve readability. These could be `computeOverflow`, `computeBonus` and `processDeposit`.
- (Note: This suggestion has now been implemented.) `require(_maxCumulativeContrib > 0);` in the constructor is useless, given the one which follows.
- (Note: This suggestion has now been implemented.) Within a contract, functions should be in the following order: `constructor`, `fallback`, `external`, `public`, `internal`, `private`.[4]

[4] http://solidity.readthedocs.io/en/latest/style-guide.html

Disclaimer

UPON REQUEST BY QIIBEE, CHAINSECURITY LTD. AGREES MAKING THIS AUDIT REPORT PUBLIC. THE CONTENT OF THIS AUDIT REPORT IS PROVIDED “AS IS”, WITHOUT REPRESENTATIONS AND WARRANTIES OF ANY KIND, AND CHAINSECURITY LTD. DISCLAIMS ANY LIABILITY FOR DAMAGE ARISING OUT OF, OR IN CONNECTION WITH, THIS AUDIT REPORT. COPYRIGHT OF THIS REPORT REMAINS WITH CHAINSECURITY LTD..