Security Audit of BlockV Smart Contract

This report is public.

ChainSecurity Ltd.

November 22, 2017

BLOCKV

Contents

Token Name	V Token (VEE)
Decimals	18
Smallest Unit (Atom)	10^{-18}VEE
Token Amount	$\approx 3.646 * 10^9$
Token Price	69164622576285 Wei
Percentage for sale	100%
Refund	None

Table 1: Facts about the token and the token sale.

We first and foremost thank BlockV for giving us the opportunity to audit your smart contract code. This documents outlines our methodology, limitations and results for your security audit.

1 System Overview

BlockV provides a platform for creating smart digital objects (vAtoms) on blockchains and distributing them among users.

In the following we describe the V Token (VEE) and its corresponding token sale. Table 1 gives the general overview.

1.1 TGE Overview

The VEEs are allocated in two phases. The TGE starts with a *contribution* phase during which contributors invest ETH. The contribution phase is split into pre-sale and main sale. Each of these sales is capped at USD 20M. The cap itself is not automatically enforced within the smart contracts. After the contribution phase, the TGE continues with a *token allocation phase* during which the price of tokens is determined and each contributor is allocated with tokens based on their contribution.

1.2 Token Price and Amount

The token price and amount of tokens to be minted are both determined after the contribution phase.

1.3 Token Distribution

The tokens are distributed among token holders during the *token allocation phase*. The total amount of issued tokens, determined as described in Section 1.2, will be allocated as follows:

- 35% of the tokens will be immediately transfer to the public (users who contributed during the crowdsale).
- 25% of the tokens will be allocated to two token holders with addresses:
  `0x2f09079059b85c11DdA29ed62FF26F99b7469950`,
  `0x3634acA3cf97dCC40584dB02d53E290b5b4b65FA`,
  `0x768D9F044b9c8350b041897f08cA77AE871AeF1C`,
  `0xb96De72d3fee8c7B6c096Ddeab93bf0b3De848c4`, and
  `0x2f97bfD7a479857a9028339Ce2426Fc3C62D96Bd`.
  20% of the tokens can be claimed immediately after token minting, while the remaining 80% of the tokens can be claimed in chunks of 20% every 6 months.
- 25% of the tokens will be allocated to the token holders with addresses:
  `0x0d02A3365dFd745f76225A0119fdD148955f821E`,
  `0x0deF4A4De337771c22Ac8C8D4b9C5Fec496841A5`,
  `0x467600367BdBA1d852dbd8C1661a5E6a2Be5F6C8`,
  `0x92E01739142386E4820eC8ddC3AFfF69de99641a`, and
  `0x1E0a7E0706373d0b76752448ED33cA1E4070753A`.
  20% of the tokens can be claimed immediately after token minting, while the remaining 80% of the tokens can be claimed in chunks of 20% every 6 months.
- 25% of the tokens will be allocated to the token holders with addresses:
  `0x4311F6F65B411f546c7DD8841A344614297Dbf62`,
  `0x3b52Ab408cd499A1456af83AC095fCa23C014e0d`, and
  `0x728D5312FbbdFBcC1b9582E619f6ceB6412B98E4`.
  15% of the tokens will be allocated to the token holder. 50% of the tokens can be claimed after 3 years while the remaining 50% can be claimed in chunks 1/36 - every month over three years.

Once the token price is fixed, the allocation of tokens and vesting of tokens is enforced by the smart contracts.

Pool A: 35Pool B: 25Pool C: 25Pool D : 15

1.4 Extra Features

Pausable BlockV has the power to pause the token sale **once** for the duration of two weeks. During this time no token transfers, token purchases or refunds can be made.

Upgradable BlockV can propose a token upgrade to new token version. This can happen at any time. Individual token owners can accept the upgrade by calling the `upgrade` function.

2 Audit Overview

2.1 Scope of the Audit

The audit was based on the Ethereum Virtual Machine (EVM) after `EIP-150` and solidity compiler `0.4.17+commit.bdeb9e52.Linux.g++`.

The scope of the audit is limited to the following source code files. All of these source code file were received on November 6th, 2017:

- BlockvPublicLedger.sol
  - Final SHA-256: `aaf0f458f18c42e9f9b437fe8eaf9328f8e81e28e16d80572651b7505697de73`
- BlockvToken.sol
  - Final SHA-256: `bd1124fe6a0116053a68abcfbbf18b7291809c0f066ee87a330800095fc2ab8a`
- BlockvTokenV2.sol
  - Final SHA-256: `c60f51f73c8a6758995233e390f97561a8530de331f5acb624cf660f9c9ce2fd`
- MigrationAgent.sol
  - Final SHA-256: `925c17f72c9e2fda9a3e4998e8c7f538434db55c429c3532feb4f09ec3bebaf4`
- Migrations.sol
  - Final SHA-256: `388bc56104f1408de7760120571e098c26a86b524a01c285c16ea6f07b0895df`
- PoolAContract.sol
  - Final SHA-256: `27a8d536f8e12c2c38fe63c38d7f9dbe40bf6004cd00c199584e60a733748d43`
- PoolBLock.sol
  - Final SHA-256: `f2c0680d8d9cfe5f0cb99aee0d494446c016d496d713d35e4b567549baeb9c35`
- PoolCLock.sol
  - Final SHA-256: `7c4de886898fab38bd01c3a0ed2441f3cfa7484361784490e49ba3684a49da12`
- PoolDLock.sol
  - Final SHA-256: `c6bb3c153416eeba4ce934e2feab4f7545f9ee6b5d6f7beece18673a827f58f3`
- PoolAllocations.sol
  - Final SHA-256: `21bdc6b3797b97423e7e37d7dcc3ff9e75872fa6724143ca1901f57ff2d648fd`

2.2 Depth of Audit

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

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

2.3 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 3 distinct categories, depending on their criticality:

- Low - can be considered as less important
- Medium - needs to be considered to be fixed
- High - should be fixed very soon
- Critical - needs to be fixed immediately

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. Finally, 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 we label it as ✓ Addressed.

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.

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

3 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 Ltd. has performed auditing in order to discover as many vulnerabilities as possible.

4 Details of the Findings

4.1 No Reentrancies ✓ No Issue

The BlockV contracts do not contain any vulnerabilities that would allow reentrancy attacks. This is because no untrusted code is ever invoked.

4.2 No Callstack Bugs ✓ No Issue

The BlockV contracts do not contain any vulnerabilities that would allow attacks based on a callstack overflow. This is because all exception are properly handled and propagated.

4.3 Ether Transfers ✓ No Issue

The smart contracts do not send or receive ether. There are neither payable functions declared in the contracts, nor calls to `transfer`, `send`, or `call.value()()`. Therefore, there are no issues related to unexpected ether transfers.

4.4 Unsafe Math Calculations in Token Distributions High ✓ Fixed

The function `fixDistribution` in the contract `BlockvPublicLedger` contract uses the following unsafe math operations that may result in incorrect token amounts allocated to the contributions. The relevant code is:

```
for(i = 0; i < distributionEntryCount; i++) {
  de = distributionList[i];
  de.tokenAmount = (de.amountContributed * _usdToEthConversionRate * 100) / (
      _tokenPrice  * de.discount / 100);
  distributionList[i] = de;
}
distributionFixed = true;
```

The calculated token price in USD-cents is `(_tokenPrice  * de.discount / 100)`. The early division by `100` can result in wrong results. Consider a scenario where the contribution is USD 10 (`de.amountContributed * _usdToEthConversionRate = 10`), the token price is 2 USD-cents per token (`_tokenPrice = 2`) and the discount is 20% (`de.discount = 80`). The expected amount of tokens allocated to the contributor is 625 tokens, which is calculated based on a token price of 1.6 USD-cents per token. However, due to the early division in the second part of the formula, we get $(2 * 80)/100 = 160/100 = 1$, and the calculated token price in 1 USD-cents per token. Consequently, the calculated amount of tokens is 1000 instead of 625.

Likelihood High

Impact High

Post-Audit Comments This function is removed. The functionality was moved to the `PoolA` contract.

4.5 Incorrect Calculation of Token Amounts for Contributors High ✓ Addressed

The calculation of tokens via the function `getTokenAmount` in `PoolAContract` does unnecessary and likely incorrect computations:

```
function getTokenAmount(uint256 amount, uint8 discountGroup) private constant
    returns(uint256) {
  uint256 discount = getTokenDiscount(discountGroup);
  return amount * tokenMultiplier * 10 ** decimals / (oneTokenInWei * discount /
      100 / discountMultiplier);
}
```

- According to the comments in the `BlockvPublicLedger` contract, the value of the variable `discount` are of the form 80, 90, etc., to represent the discounted value of tokens. However, in the `PoolAContract`, the discount is obtained via a call to the function `getTokenDiscount`. This function returns non-default discounts (discounts different than 100) only for values 1, 2, 3, and 100. These discount IDs are defined in the constructor of `PoolAContract`. This means that a discounted

value of 80 stored in the public ledger would be resolved to the default discounted value 100 in the `PoolAContract`.

Overall, the function `getTokenAmount` requires a careful examination and testing as the current version is very likely to be incorrect.

Post-Audit Comments The discount fields in the public ledger contract will be updated to discount IDs 1, 2, 3, and 100.

4.6 No Tokens can be Transfered by the PoolAContract High ✓ Addressed

The `distribution` function inside the `PoolAContract` sends tokens to the token buyers. It therefore uses the following function:

```
1  assert(blockVContract.transferFrom(msg.sender, to, tokenAmount));
```

However, the use of the `transferFrom` function requires the previous approval of this transfer. As seen in the code:

```
1  function transferFrom(address _from, address _to, uint256 _value)
       onlyPayloadSize(3) returns (bool) {
2  ...
3      allowed[_from][msg.sender] = _allowance.sub(_value);
```

However, it seems that this allowance is never provided. Therefore, the tokens cannot be transferred.

Likelihood High

Impact High

Post-Audit Comments Approval will be set up off-chain by the deployment script.

4.7 Hardcoded Token Amounts for Transfers in PoolBLock High ✓ Fixed

The tokens allocated to the `PoolBLock` smart contract are claimed using the function:

```
function claim() {
  var elem = allocations[msg.sender];

  require(now >= elem.releaseTime);
  require(elem.numPayoutCycles > 0);

  uint256 amount = token.balanceOf(this);
  require(amount >= elem.amount);

  elem.numPayoutCycles -= 1;
  elem.releaseTime = now + releaseTimeDelay;

  assert(token.transfer(msg.sender, elem.amount));
}
```

Two addresses can claim tokens. The amount of tokens that can be claimed (`amount`) as well as the number of payout cycles (`numPayoutCycles`) are initialized as follows:

```
allocations[0xFba09655dE6FCb113A1733Cf980d58a9b226e031] = lockEntry(25, now, 5);
allocations[0x77373d8bfD31D25102237F9A8D2d838d25707782] = lockEntry(1000, now, 5);
```

The amount of tokens that can be claimed by each of these contracts is hardcoded to 25 and 1000 tokens, respectively. The number of payout cycles is fixed to 5 for both contracts. This means that the maximum amount of tokens that can be claimed by the contracts are 125 and 5000, respectively. If `PoolBLock` is given more than $5,125$ tokens, then the extra tokens would remain locked in this contract as the two addresses given above would not be allowed to claim the rest.

The total number of tokens that can be allocated using `PoolBLock`, 5125 tokens, does not seem correct for two reasons. First, this number is too low as the expected number of allocated tokens is 25% of approximately 3B tokens. Second, this number is grains, which means that it is a tiny fraction of a single token.

Likelihood High

Impact High

Post-Audit Comments The total amount of tokens in the PoolBLock contract has been updated with the correct amount of tokens allocated to PoolB.

4.8 Possibly Confused Default Non-Payable Function Low ✓ Fixed

The `BlockvToken` and the `BlockvTokenV2` contracts define the following default function (without an explicit name):

```
function () {
  //if ether is sent to this address, send it back.
  revert();
}
```

According to the comments, this function intents to return any ether sent to the contract. However, this function is *not* marked as `payable` and cannot receive ether.

Likelihood Low

Impact Low

Fix The function was removed.

4.9 Tokens Can Get Locked During Allocation Low ✓ Fixed

25% of the minted tokens will get distributed by the `PoolCLock` smart contract. As described in Section 1.3, 20% these tokens are available immediately and the remaining 80% can be claimed in chunks of 20% percent every 6 months using the following function:

```
function claim() {
  require(msg.sender == beneficiary);
  require(now >= releaseTime);

  uint256 amount = token.balanceOf(this);
  require(amount >= tokensToBeReleased);

  releaseTime = now + releaseTimeDelay;
  assert(token.transfer(beneficiary, tokensToBeReleased));
}
```

The value of `tokensToBeRelease` is calculated as:

```
tokensToBeReleased = SafeMath.div(SafeMath.mul(_totalAmount, percentOfTokens),
    100);
```

All tokens are expected to be claimed after five calls to `claim`. Due to integer rounding, few tokens can remain after the 5th call to `claim`. Furthermore, tokens that remain would be locked if the condition `(token.balanceOf(this) >= tokensToBeReleased)` is false. For example, if 999 tokens are allocated to `PoolCLock`, then the tokens are released in chunks of 199 tokens per call to `claim`. After the 5th call to `claim`, 4 tokens would remain locked.

Since the tokens allocated by the `PoolDLock` contract are handled similarly, this contract may also lock tokens.

Likelihood Low

Impact Low

Post-Audit Comments This issue is addressed by an explicit PoolAllocation contracts that allocates the extra tokens to the first claim.

4.10 Vesting Can Take Longer than Expected Low ✓ Fixed

The vesting scheme implemented in `PoolBLock` is expected to take 2 years. The vesting scheme can take longer if token holders do not claim their tokens on time. The `claim` function is implemented as follows:

```
1  /**
2  * @dev claims tokens held by time lock
3  */
4  function claim() {
5    var elem = allocations[msg.sender];
6
7    require(now >= elem.releaseTime);
8    require(elem.numPayoutCycles > 0);
9
10   uint256 amount = token.balanceOf(this);
11   require(amount >= elem.amount);
12
13   elem.numPayoutCycles -= 1;
14   elem.releaseTime = now + releaseTimeDelay;
15
16   assert(token.transfer(msg.sender, elem.amount));
17 }
```

The `releaseTime` is set to 6 months after the current time when the tokens are claimed. Therefore, if a token holder waits, for example, 3 years before the first claim is made, he or she would be unable to make a subsequent claim in the next 6 months.

Since token holders have incentive to claim their tokens on time, we classify this unexpected behavior as low likelihood and impact.

Likelihood Low

Impact Low

Post-Audit Comments This issue is addressed by an explicit PoolAllocation contracts that allocates the extra tokens to the first claim. Additionally, users can now issue multiple, consecutive claims if the necessary amount of time has passed.

5 Recommendations

- The use of `discountMultiplier` seems unnecessary. The discounts are first multiplied with `discountMultiplier` and then they are divided by `discountMultiplier` when calculating the amount of tokens for a given contributor (in function `getTokenAmount` of the `PoolAContract` contract).
- Please include brackets to emphasize the operator precedence in the calculation of tokens (function `getTokenAmount` in the `PoolAContract` contract):
  `amount * tokenMultiplier * 10 ** decimals / (oneTokenInWei * discount / 100 / discountMultiplier)`
- Information on the website and inside the One-Page documentation say that the 15% pool will be locked for 6 years, whereas the white paper (correctly) says that it will be locked for **up to** 6 years, as half of these tokens already become available after 3 years.
- `PoolBLock`, `PoolCLock` and `PoolDLock` all contain code like:

```
1     uint256 amount = token.balanceOf(this);
2     require(amount >= elem.amount);
```

  This code is redundant as this exact check is performed inside the `transfer` function that is subsequently called.

- There is this comment in BlockvToken.sol, which is confusing:

```
1 /**
2  * VEE Tokens are divisible by 1e8 (100,000,000) base
3  * units referred to as 'Grains'.
4  */
```

  However, VEE Tokens have 18 decimals as defined here:

```
1   uint8 public constant decimals = 18;
```

  Therefore each token is divisible into 10^{18} grains and not 10^8 as claimed in the comment.

- The function `claim` in the `PoolBLock` contract does not restrict access to users that can claim tokens. This issue us not directly exploitable because the condition `require(elem.numPayoutCycles > 0)` would evaluate to false for addresses other than `0xFba09655dE6FCb113A1733Cf980d58a9b226e031` and `0x77373d8bfD31D25102237F9A8D2d838d25707782`, which have initialized lock entries in the `allocations` map.

6 Conclusion

The BlockV smart contracts have been analyzed under different aspects, with different open-source tools as well as our fully fledge proprietary in-house tool. Overall, we found that BlockV employs good coding practices and has clean, documented code. We have no remaining security concerns about the BlockV smart contracts, as all detected issues were either fixed or addressed.

7 Disclaimer