CertiK Audit Report For BQCC

Request Date: 2019-07-25
Revision Date: 2019-08-01
Platform Name: EOS

CERTIK

Contents

Disclaimer

This Report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Verification Services Agreement between CertiK and BQCC(the "Company"), or the scope of services/verification, and terms and conditions provided to the Company in connection with the verification (collectively, the "Agreement"). This Report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This Report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes without CertiK's prior written consent.

About CertiK

CertiK is a technology-led blockchain security company founded by Computer Science professors from Yale University and Columbia University built to prove the security and correctness of smart contracts and blockchain protocols.

CertiK, in partnership with grants from IBM and the Ethereum Foundation, has developed a proprietary Formal Verification technology to apply rigorous and complete mathematical reasoning against code. This process ensures algorithms, protocols, and business functionalities are secured and working as intended across all platforms.

CertiK differs from traditional testing approaches by employing Formal Verification to mathematically prove blockchain ecosystem and smart contracts are hacker-resistant and bug-free. CertiK uses this industry-leading technology together with standardized test suites, static analysis, and expert manual review to create a full-stack solution for our partners across the blockchain world to secure 1.4B in assets.

For more information: https://certik.org/

Testing Summary

PASS

CERTIK *believes this smart contract passes security qualifications to be listed on digital asset exchanges.*

Aug 01, 2019

Exective Summary

This report has been prepared as the product of the Smart Contract Audit request by BQCC. This audit was conducted to discover issues and vulnerabilities in the source code of BQCC's Smart Contracts. Utilizing CertiK's Formal Verification Platform, Static Analysis, and Manual Review, a comprehensive examination has been performed. The auditing process pays special attention to the following considerations.

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessment of the codebase for best practice and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line by line manual review of the entire codebase by industry experts.

Review Notes

Source Code SHA-256 Checksum

- **LockableToken.cpp**
  `08d8b219e765de0dbaa572e6746940679cfb46499b4b682e902be67bc40d3653`
- **LockableToken.hpp**
  `8956171072fd1cc86ea708201aef528074ac4d62c6515c618497ceaa3a520990`

Summary

CertiK was chosen by BQCC to audit the design and implementation of its `LockableToken` smart contract. To ensure comprehensive protection,the source code has been analyzed by the proprietary CertiK formal verification engine and manually reviewed by our smart contract experts and engineers. That end-to-end process ensures proof of stability as well as a hands-on, engineering-focused process to close potential loopholes and recommend design changes in accordance with the best practices in the space.

Recommendations

Items in this section are low impact to the overall aspects of the smart contracts, thus will let client to decide whether to have those reflected in the final deployed version of source codes.

LockableToken.cpp

- INFO `lock()`, `transfer()`: Recommend providing default error messages for `.get()`.
- INFO `retire()`: Recommend adding checking for `s.supply` and `quantity`.

Source Code

File LockableToken.hpp

```
1  #pragma once
2
3  #include <eosiolib/asset.hpp>
4  #include <eosiolib/eosio.hpp>
5  #include <string>
6
7  #define DAY_MICROSECONDS 86400000000
8
9  namespace eosiosystem {
10     class system_contract;
11 }
12
13 namespace eosio {
14 using std::string;
15
16 CONTRACT LockableToken : public contract {
17    using contract::contract;
18    public:
19        struct transfer_args {
20            name   from;
21            name   to;
22            asset  quantity;
23            string memo;
24        };
25
26        ACTION create(name issuer, asset maximum_supply);
27
28        ACTION issue(name to, asset quantity, string memo);
29        ACTION retire(asset quantity, string memo);
30
31        ACTION transfer(name from, name to, asset quantity, string memo);
32
33        ACTION open(name owner, symbol_code symbol, name ram_payer);
34        ACTION close(name owner, symbol_code symbol);
35
36        ACTION lock(name owner, asset quantity, uint64_t days);
37        ACTION unlock(name owner, symbol_code symbol);
38
39        static asset get_supply(name token_contract_account, symbol_code sym) {
40            stats statstable(token_contract_account, sym.raw());
41            const auto& st = statstable.get(sym.raw());
42            return st.supply;
43        }
44
45        static asset get_balance(name token_contract_account, name owner, symbol_code
              sym) {
46            accounts accountstable(token_contract_account, owner.value);
47            const auto& ac = accountstable.get(sym.raw());
48            return ac.balance;
49        }
50
51    private:
52        TABLE account {
53            asset balance;
```

```
54              uint64_t primary_key() const { return balance.symbol.code().raw(); }
55          };
56
57          TABLE lock_account {
58              asset lock_balance;
59              uint64_t unlock_date = 0;
60              uint64_t primary_key() const { return lock_balance.symbol.code().raw(); }
61          };
62
63          TABLE currency_stats {
64              asset  supply;
65              asset  max_supply;
66              name   issuer;
67              uint64_t primary_key() const { return supply.symbol.code().raw(); }
68          };
69
70          typedef eosio::multi_index<name("accounts"), account> accounts;
71          typedef eosio::multi_index<name("lock"), lock_account> lock_accounts;
72          typedef eosio::multi_index<name("stat"), currency_stats> stats;
73
74          void sub_balance(name owner, asset value);
75          void add_balance(name owner, asset value, name ram_payer);
76  };
77
78  } /// namespace eosio
```

File LockableToken.cpp

```
#include "LockableToken.hpp"

namespace eosio {

ACTION LockableToken::create(name issuer, asset maximum_supply) {
    require_auth(_self);

    auto sym = maximum_supply.symbol;
    eosio_assert(sym.is_valid(), "invalid symbol name");
    eosio_assert(maximum_supply.is_valid(), "invalid supply");
    eosio_assert(maximum_supply.amount > 0, "max-supply must be positive");

    stats statstable(_self, sym.code().raw());
    auto existing = statstable.find(sym.code().raw());
    eosio_assert(existing == statstable.end(), "token with symbol already exists");

    statstable.emplace(_self, [&](auto& s) {
        s.supply.symbol = maximum_supply.symbol;
        s.max_supply   = maximum_supply;
        s.issuer       = issuer;
    });
}

ACTION LockableToken::issue(name to, asset quantity, string memo) {
    auto sym = quantity.symbol;
    eosio_assert(sym.is_valid(), "invalid symbol name");
    eosio_assert(memo.size() <= 256, "memo has more than 256 bytes");

    auto sym_name = sym.code().raw();
    stats statstable(_self, sym_name);
    auto existing = statstable.find(sym_name);
    eosio_assert(existing != statstable.end(), "token with symbol does not exist,
        create token before issue");
    const auto& st = *existing;
    require_auth(st.issuer);
    eosio_assert(quantity.is_valid(), "invalid quantity");
    eosio_assert(quantity.amount > 0, "must issue positive quantity");

    eosio_assert(quantity.symbol == st.supply.symbol, "symbol precision mismatch");
    eosio_assert(quantity.amount <= st.max_supply.amount - st.supply.amount, "quantity
         exceeds available supply");

    statstable.modify(st, eosio::same_payer, [&](auto& s) {
        s.supply += quantity;
    });

    add_balance(st.issuer, quantity, st.issuer);

    if (to != st.issuer) {
        SEND_INLINE_ACTION(*this, transfer, {st.issuer,name("active")}, {st.issuer, to,
             quantity, memo});
    }
}

ACTION LockableToken::retire(asset quantity, string memo) {
    auto sym = quantity.symbol;
    eosio_assert(sym.is_valid(), "invalid symbol name");
```

```
55      eosio_assert(memo.size() <= 256, "memo has more than 256 bytes");
56
57      auto sym_name = sym.code().raw();
58      stats statstable(_self, sym_name);
59      auto existing = statstable.find(sym_name);
60      eosio_assert(existing != statstable.end(), "token with symbol does not exist");
61      const auto& st = *existing;
62      require_auth(st.issuer);
63      eosio_assert(quantity.is_valid(), "invalid quantity");
64      eosio_assert(quantity.amount > 0, "must retire positive quantity");
65
66      eosio_assert(quantity.symbol == st.supply.symbol, "symbol precision mismatch");
67
68      statstable.modify(st, eosio::same_payer, [&](auto& s) {
69          s.supply -= quantity;
70      });
71
72      sub_balance(st.issuer, quantity);
73  }
74
75  ACTION LockableToken::transfer(name from, name to, asset quantity, string memo) {
76      eosio_assert(from != to, "cannot transfer to self");
77      require_auth(from);
78      eosio_assert(is_account(to), "to account does not exist");
79      auto sym = quantity.symbol.code().raw();
80      stats statstable(_self, sym);
81      const auto& st = statstable.get(sym);
82
83      require_recipient(from);
84      require_recipient(to);
85
86      eosio_assert(quantity.is_valid(), "invalid quantity");
87      eosio_assert(quantity.amount > 0, "must transfer positive quantity");
88      eosio_assert(quantity.symbol == st.supply.symbol, "symbol precision mismatch");
89      eosio_assert(memo.size() <= 256, "memo has more than 256 bytes");
90
91      auto payer = has_auth(to) ? to : from;
92
93      sub_balance(from, quantity);
94      add_balance(to, quantity, payer);
95  }
96
97  void LockableToken::sub_balance(name owner, asset value) {
98      auto sym = value.symbol.code().raw();
99      accounts from_acnts(_self, owner.value);
100     const auto& from = from_acnts.get(sym, "no balance object found");
101
102     lock_accounts lock_acnts(_self, owner.value);
103     auto itr = lock_acnts.find(sym);
104     if (itr != lock_acnts.end() && itr->unlock_date > current_time()) {
105         eosio_assert(from.balance.amount - itr->lock_balance.amount >= value.amount, "
                overdrawn available balance");
106     } else {
107         eosio_assert(from.balance.amount >= value.amount, "overdrawn balance");
108     }
109
110     from_acnts.modify(from, owner, [&](auto& a) {
111         a.balance -= value;
```

```
112     });
113 }
114
115 void LockableToken::add_balance(name owner, asset value, name ram_payer) {
116     accounts to_acnts(_self, owner.value);
117     auto to = to_acnts.find(value.symbol.code().raw());
118     if (to == to_acnts.end()) {
119         to_acnts.emplace(ram_payer, [&](auto& a) {
120             a.balance = value;
121         });
122     } else {
123         to_acnts.modify(to, eosio::same_payer, [&](auto& a) {
124             a.balance += value;
125         });
126     }
127 }
128
129 ACTION LockableToken::open(name owner, symbol_code symbol, name ram_payer) {
130     require_auth(ram_payer);
131
132     auto sym = symbol.raw();
133
134     stats statstable(_self, sym);
135     const auto& st = statstable.get(sym, "symbol does not exist");
136     eosio_assert(st.supply.symbol.code().raw() == sym, "symbol precision mismatch");
137
138     accounts acnts(_self, owner.value);
139     auto it = acnts.find(sym);
140     if (it == acnts.end()) {
141         acnts.emplace(ram_payer, [&](auto& a) {
142             a.balance = asset{0, st.supply.symbol};
143         });
144     }
145 }
146
147 ACTION LockableToken::close(name owner, symbol_code symbol) {
148     require_auth(owner);
149
150     accounts acnts(_self, owner.value);
151     auto it = acnts.find(symbol.raw());
152     eosio_assert(it != acnts.end(), "Balance row already deleted or never existed.
        Action won't have any effect.");
153     eosio_assert(it->balance.amount == 0, "Cannot close because the balance is not
        zero.");
154     acnts.erase(it);
155 }
156
157 ACTION LockableToken::lock(name owner, asset quantity, uint64_t days) {
158     require_auth(_self);
159     require_recipient(owner);
160
161     auto sym = quantity.symbol.code().raw();
162     stats statstable(_self, sym);
163     const auto& st = statstable.get(sym);
164     eosio_assert(is_account(owner), "owner account does not exist");
165     eosio_assert(quantity.is_valid(), "invalid quantity");
166     eosio_assert(quantity.amount > 0, "must lock positive quantity");
167     eosio_assert(quantity.symbol == st.supply.symbol, "symbol precision mismatch");
```

```
168     eosio_assert(days > 0, "must lock positive days");
169
170     accounts acnts(_self, owner.value);
171     const auto& acnt = acnts.get(sym, "no balance object found");
172     eosio_assert(acnt.balance.amount >= quantity.amount, "overdrawn balance");
173     lock_accounts lock_acnts(_self, owner.value);
174     auto itr = lock_acnts.find(sym);
175     if (itr == lock_acnts.end()) {
176         lock_acnts.emplace(_self, [&](auto& a) {
177             a.lock_balance = quantity;
178             a.unlock_date = current_time() + DAY_MICROSECONDS * days;
179         });
180     } else {
181         lock_acnts.modify(itr, eosio::same_payer, [&](auto& a) {
182             a.lock_balance = quantity;
183             a.unlock_date = current_time() + DAY_MICROSECONDS * days;
184         });
185     }
186 }
187
188 ACTION LockableToken::unlock(name owner, symbol_code symbol) {
189     require_auth(_self);
190     require_recipient(owner);
191
192     lock_accounts lock_acnts(_self, owner.value);
193     auto it = lock_acnts.find(symbol.raw());
194     eosio_assert(it != lock_acnts.end(), "Lock row already deleted or never existed.
            Action won't have any effect.");
195     lock_acnts.erase(it);
196 }
197
198 } /// namespace eosio
199
200 EOSIO_DISPATCH(eosio::LockableToken, (create)(issue)(transfer)(open)(close)(retire)(
        lock)(unlock))
```