YOLO Smart Contract Security Audit Report

May 6, 2019

Abstract

In this report, we consider the smart contract security of the YOLO project of KyberNetwork. Our task is to find and describe security issues in the smart contracts of the project.

Disclaimer

The audit does not give any warranties on the security of the code. SlowMist only issues this report based on the fact that has occurred or existed before the report is issued, and bears the corresponding responsibility in this regard. For the facts occur or exist later after the report, SlowMist cannot judge the security status of its smart contract. SlowMist is not responsible for it. The security audit analysis and other contents of this report are based on the documents and materials provided by the information provider to SlowMist as of the date of this report (referred to as "the provided information"). SlowMist assumes that: there has been no information missing, tampered, deleted, or concealed. If the information provided has been missed, modified, deleted, concealed or reflected and is inconsistent with the actual situation, SlowMist will not bear any responsibility for the resulting loss and adverse effects.

Summary

In this report we have considered the security of YOLO smart contracts. The security audit has shown **no critical/high-risk/medium-risk issues**. However, **a low-risk issue** have been found.

Project Overview

Project Description

In our analysis we consider YOLO smart contracts code (Git repository: https://github.com/KyberNetwork/eos_smart_contracts, branch: audit3, version on commit: 33c5d9896f89487d26386d28bd080592c0b1de0a).

Project Architecture

The project includes the following smart contract files:

```
$ tree ./contracts
./contracts
├── Common
│   └── common.hpp
├── Listener
│   ├── Listener.cpp
├── Mock (Not in the scope of the audit)
│   └── Token
│       ├── Token.cpp
│       ├── Token.hpp
├── Network
│   ├── Network.cpp
│   ├── Network.hpp
└── Reserve
    └── AmmReserve
        ├── AmmReserve.cpp
        ├── AmmReserve.hpp
        └── liquidity.hpp
```

Audit Methodology

Our security audit process for smart contract includes two stages:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using public and in-house automated analysis tools.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that was considered during the audit of the smart contract:

- Reentrancy attack and other Race Conditions
- Replay attack
- Denial of service attack
- Authority Control attack
- Integer Overflow and Underflow attack
- False Notice attack
- False Error Notification attack
- Counterfeit Token attack
- Rollback attack
- Random Number security audit
- Precision Control of double
- Logic Flaws

Audit Result

Critical issues

Critical issues seriously endanger smart contracts security. We highly recommend fixing them.

The audit has shown no critical issues.

High-Risk issues

High-Risk issues can affect the normal functioning of smart contracts. We strongly recommend fixing them.

The audit has shown no high-risk issues.

Medium-Risk issues

Medium-Risk issues can influence smart contracts operation in current implementation. We highly recommend addressing them.

The audit has shown no medium-risk issues.

Low-Risk issues

Low-Risk issues can influence smart contracts operation in future versions of code. We recommend taking them into account.

Missing parameter check

1. **Action:** **Network::setadmin(name admin)**

Suggestion: Check that new admin cannot be the same as the current admin;

2. **Action:** **Network::setenable(bool enable)**

   Suggestion: Check that new stat(enable) cannot be the same as the current stat;

3. **Action:** **Network::setlistener(name listener)**

   Suggestion: Check that new listener cannot be the same as the current listener;

4. **Action:** **AmmReserve::setadmin(name admin)**

   Suggestion: Same as the point 1;

5. **Action:** **AmmReserve::setnetwork(name network_contract)**

   Suggestion: Check that new network_contract cannot be the same as the current network_contract;

6. **Action:** **AmmReserve::setenable(bool enable)**

   Suggestion: Same as the point 2;

Appendix

Smart Contract Source Code

Common/common.cpp

```
#pragma once

#include <math.h>
#include <string>
#include <vector>
#include <eosiolib/eosio.hpp>
#include <eosiolib/asset.hpp>
#include <eosiolib/symbol.hpp>
#include <eosiolib/singleton.hpp>

using std::string;
using std::vector;
using std::make_tuple;
using std::stoi;
using namespace eosio;

#define EOS_PRECISION 4
#define EOS_SYMBOL symbol("EOS", EOS_PRECISION)
#define MAX_AMOUNT asset::max_amount
#define MAX_RATE 1000000 /* up to 1M tokens per EOS */
#define STAKE_ACCOUNT "eosio.stake"_n
#define RAM_ACCOUNT "eosio.ram"_n

struct account {
    asset    balance;
    uint64_t primary_key() const { return balance.symbol.code().raw(); }
};
typedef eosio::multi_index<"accounts"_n, account> accounts;

asset get_balance(name user, name token_contract, symbol symbol) {
    accounts fromAcc(token_contract, user.value);
    auto itr = fromAcc.find(symbol.code().raw());
    if (itr == fromAcc.end()) {
        /* balance was never created */
```

```
        return asset(0, symbol);
    }
    return itr->balance;
}

void async_pay(name from, name to, asset quantity, name dest_contract, string memo) {
    action {
        permission_level{from, "active"_n},
        dest_contract,
        "transfer"_n,
        std::make_tuple(from, to, quantity, memo)
    }.send();
}

vector<string> split(const string& str, const string& delim) {
    vector<string> tokens;
    size_t prev = 0, pos = 0;
    do {
        pos = str.find(delim, prev);
        if (pos == string::npos) pos = str.length();
        string token = str.substr(prev, pos-prev);
        tokens.push_back(token);
        prev = pos + delim.length();
    } while (pos < str.length() && prev < str.length());
    return tokens;
}

float stof(const char* s) {
    float rez = 0, fact = 1;
    if (*s == '-') {
        s++;
        fact = -1;
    }

    for (int point_seen = 0; *s; s++) {
        if (*s == '.') {
            point_seen = 1;
            continue;
        }

        int d = *s - '0';
        if (d >= 0 && d <= 9) {
```

```
            if (point_seen) fact /= 10.0f;
            rez = rez * 10.0f + (float)d;
        }
    }

    return rez * fact;
}

int64_t to_int64(double x) {
    eosio_assert(x <= MAX_AMOUNT, "fail max amount overflow validation");
    return int64_t(x);
}

double amount_to_damount(int64_t amount, uint64_t precision) {
    return (double(amount) / double(pow(10, precision)));
}

double asset_to_damount(asset quantity) {
    return (double(quantity.amount) / double(pow(10, quantity.symbol.precision())));
}

int64_t damount_to_amount(double damount, uint64_t precision) {
    return to_int64(damount * double(pow(10, precision)));
}

asset calc_dest(double rate, asset src, symbol dest_symbol) {
    double src_damount = amount_to_damount(src.amount, src.symbol.precision());
    double dest_damount = src_damount * rate;
    int64_t dest_amount = damount_to_amount(dest_damount, dest_symbol.precision());

    return asset(dest_amount, dest_symbol);
}
```

Listener/Listener.cpp

```
#include <eosiolib/eosio.hpp>
#include <eosiolib/asset.hpp>
#include "../Common/common.hpp"

using namespace eosio;

CONTRACT Listener : public contract {
```

```
public:
    using contract::contract;

    TABLE state {
        name   eos_contract;
        name   network_contract;
        double rebate_percent;
        asset  min_eos_for_rebate;
    };

    typedef eosio::singleton<"state"_n, state> state_type;

    ACTION config(name eos_contract,
                  name network_contract,
                  double rebate_percent,
                  asset min_eos_for_rebate) {
        eosio_assert(is_account(eos_contract), "eos contract not exist");
        eosio_assert(is_account(network_contract), "network contract does not exist");
        eosio_assert((rebate_percent >= 0) && (rebate_percent < 100.000), "illegal rebate");
        eosio_assert(min_eos_for_rebate.symbol == EOS_SYMBOL, "wrong symbol");

        require_auth(_self);

        state_type state_inst(_self, _self.value);
        state new_state = {eos_contract, network_contract, rebate_percent, min_eos_for_rebate};
        state_inst.set(new_state, _self);
    }

    ACTION posttrade(asset src, asset dest, name reserve, name sender) {
        state_type state_inst(_self, _self.value);
        if (!state_inst.exists()) return;
        auto state = state_inst.get();

        require_auth(state.network_contract);

        if (state.rebate_percent > 0) {
            asset eos_traded = (src.symbol == EOS_SYMBOL) ? src : dest;
            asset rebate = calc_dest(state.rebate_percent / 100.00, eos_traded, EOS_SYMBOL);
            asset eos_balance = get_balance(_self, state.eos_contract, EOS_SYMBOL);

            if ((rebate.amount > 0) &&
                (rebate <= eos_balance) &&
```

```
                    (eos_traded >= state.min_eos_for_rebate)) {
                    async_pay(_self, sender, rebate, state.eos_contract, "rebate");
                }
            }
        }
};

extern "C" {
    [[noreturn]] void apply(uint64_t receiver, uint64_t code, uint64_t action) {
        if (code == receiver) {
            switch (action) {
                EOSIO_DISPATCH_HELPER( Listener, (config)(posttrade))
            }
        }
        eosio_exit(0);
    }
}
```

Network/Network.cpp

```
#include "Network.hpp"
#include <math.h>

ACTION Network::init(name admin, name eos_contract, name listener, bool enable) {
    eosio_assert(is_account(admin), "admin account does not exist");
    eosio_assert(is_account(eos_contract), "eos contract does not exist");

    require_auth(_self);

    state_type state_inst(_self, _self.value);
    eosio_assert(!state_inst.exists(), "init already called");

    state new_state = {admin, eos_contract, listener, enable, false};
    state_inst.set(new_state, _self);
}

ACTION Network::setadmin(name admin) {
    eosio_assert(is_account(admin), "new admin account does not exist");

    auto state_inst = get_state_assert_admin();

    auto s = state_inst.get();
```

```
    s.admin = admin;
    state_inst.set(s, _self);
}

ACTION Network::setenable(bool enable) {
    auto state_inst = get_state_assert_admin();

    auto s = state_inst.get();
    s.enabled = enable;
    state_inst.set(s, _self);
}

ACTION Network::setlistener(name listener) {
    auto state_inst = get_state_assert_admin();

    auto s = state_inst.get();
    s.listener = listener;
    state_inst.set(s, _self);
}

ACTION Network::addreserve(name reserve, bool add) {
    eosio_assert(is_account(reserve), "reserve account does not exist");

    get_state_assert_admin();

    reserves_type reserves_inst(_self, _self.value);
    auto itr = reserves_inst.find(reserve.value);
    bool exists = (itr != reserves_inst.end());
    eosio_assert(((add && !exists) || (!add && exists)),
                 "can only add a non existing reserve or delete an existing one");
    if (add) {
        reserves_inst.emplace(_self, [&](auto& s) {
            s.contract = reserve;
            s.num_tokens = 0;
        });
    } else {
        eosio_assert(itr->num_tokens == 0, "reserve has listed tokens");
        reserves_inst.erase(itr);
    }
}

ACTION Network::listpairres(name reserve, symbol token_symbol, name token_contract, bool add) {
```

```
eosio_assert(is_account(token_contract), "token contract does not exist");

get_state_assert_admin();

reserves_type reserves_inst(_self, _self.value);
auto res_itr = reserves_inst.find(reserve.value);
eosio_assert(res_itr != reserves_inst.end(), "invalid reserve");
reserves_inst.modify(res_itr, _self, [&](auto& s) {
    s.num_tokens += (add ? 1 : -1);
});

reservespert_type reservespert_table_inst(_self, _self.value);
auto itr = reservespert_table_inst.find(token_symbol.raw());
auto token_exists = (itr != reservespert_table_inst.end());

if (add) {
    if (!token_exists) {
        reservespert_table_inst.emplace(_self, [&](auto& s) {
           s.symbol = token_symbol;
           s.token_contract = token_contract;
           s.reserve_contracts.push_back(reserve);
        });
    } else {
        reservespert_table_inst.modify(itr, _self, [&](auto& s) {
            auto res_it = find(s.reserve_contracts.begin(), s.reserve_contracts.end(), reserve);
            eosio_assert(res_it == s.reserve_contracts.end(), "already listed in reserve");
            s.reserve_contracts.push_back(reserve);
        });
    }
} else {
    eosio_assert(token_exists, "not listed at all");
    bool last_reserve_for_token = false;
    reservespert_table_inst.modify(itr, _self, [&](auto& s) {
        auto res_it = find(s.reserve_contracts.begin(), s.reserve_contracts.end(), reserve);
        eosio_assert(res_it != s.reserve_contracts.end(), "not listed in reserve");
        s.reserve_contracts.erase(res_it);
        if (!s.reserve_contracts.size()) {
            last_reserve_for_token = true;
        }
    });
    if (last_reserve_for_token) {
        reservespert_table_inst.erase(itr);
```

```
        }
    }

    /* Note: token stats entries are never deleted, so we can continue count on re-list. */
    if (add && !token_exists) {
        tokenstats_type tokenstats_table_inst(_self, _self.value);
        if (tokenstats_table_inst.find(token_symbol.raw()) == tokenstats_table_inst.end()) {
            tokenstats_table_inst.emplace(_self, [&](auto& s) {
               s.token_counter = asset(0, token_symbol);
               s.eos_counter = asset(0, EOS_SYMBOL);
            });
        }
    }
}

ACTION Network::withdraw(name to, asset quantity, name dest_contract, string memo) {
    eosio_assert(is_account(to), "to account does not exist");
    eosio_assert(is_account(dest_contract), "dest contract does not exist");
    eosio_assert(quantity.is_valid() && quantity.amount > 0, "illegal quantity");
    eosio_assert(to != _self, "can not witdraw to self");

    get_state_assert_admin();
    async_pay(_self, to, quantity, dest_contract, memo);
}

ACTION Network::getexprate(asset src, symbol dest_symbol) {
    eosio_assert(src.is_valid(), "invalid transfer");
    eosio_assert(src.amount >= 0, "src amount can not be negative");

    eosio_assert(src.symbol == EOS_SYMBOL || dest_symbol == EOS_SYMBOL, "src or dest must be EOS");
    eosio_assert(src.symbol != dest_symbol, "src symbol can not equal dest symbol");

    reservespert_type reservespert_table_inst(_self, _self.value);
    auto token_symbol = (src.symbol == EOS_SYMBOL) ? dest_symbol: src.symbol;
    auto token_entry = reservespert_table_inst.get(token_symbol.raw(), "unlisted token");

    async_search_best_rate(token_entry, src);
    SEND_INLINE_ACTION(*this, storeexprate, {_self, "active"_n}, {src, dest_symbol});
}

ACTION Network::storeexprate(asset src, symbol dest_symbol) {
    require_auth(_self);  // can only be called internally
```

```
    state_type state_inst(_self, _self.value);
    eosio_assert(state_inst.exists(), "init not called yet");

    double best_rate;
    name best_reserve;
    get_best_rate_results(src, dest_symbol, best_rate, best_reserve);

    asset dest = calc_dest(best_rate, src, dest_symbol);

    rate_type rate_inst(_self, _self.value);
    rate_inst.set({best_rate, dest}, _self);
}

void Network::trade(name from, name to, asset src, string memo, state &state) {
    reentrancy_check(true);

    eosio_assert(state.enabled, "trade not enabled");
    eosio_assert(memo.length() > 0, "needs a memo");

    /* gather all inputs together, non of them is trusted yet. */
    bool buy = (src.symbol == EOS_SYMBOL);
    auto info = create_trade_info(memo, from, src, _code);

    /* validate inputs. */
    eosio_assert(info.src.is_valid(), "invalid transfer");
    eosio_assert(info.src.amount > 0, "src must be positive");

    eosio_assert(info.src.symbol == EOS_SYMBOL || info.dest.symbol == EOS_SYMBOL, "no eos side");
    eosio_assert(info.src.symbol != info.dest.symbol, "src symbol can not equal dest symbol");

    auto token_symbol = buy ? info.dest.symbol: info.src.symbol;
    reservespert_type reservespert_table_inst(_self, _self.value);
    auto token_entry = reservespert_table_inst.get(token_symbol.raw(), "unlisted token");

    /* note: this is the check against _code, to prevent fake src token attacks. */
    name expected_src_contract = buy ? state.eos_contract : token_entry.token_contract;
    eosio_assert(info.src_contract == expected_src_contract, "unexpected src contract.");

    name expected_dest_contract = buy ? token_entry.token_contract : state.eos_contract;
    eosio_assert(info.dest_contract == expected_dest_contract, "unexpected dest contract.");
```

```
    async_search_best_rate(token_entry, info.src);
    SEND_INLINE_ACTION(*this, trade1, {_self, "active"_n}, {info});
}

ACTION Network::trade1(trade_info info) {
    require_auth(_self);  // can only be called internally

    double best_rate;
    name best_reserve;
    get_best_rate_results(info.src, info.dest.symbol, best_rate, best_reserve);
    eosio_assert(best_rate != 0, "got 0 rate.");
    eosio_assert(best_rate >= info.min_conversion_rate, "rate < min conversion rate.");
    eosio_assert(best_rate <= MAX_RATE, "rate > max rate.");

    asset dest = calc_dest(best_rate, info.src, info.dest.symbol);

    asset balance_pre = get_balance(info.sender, info.dest_contract, info.dest.symbol);

    /* do reserve trade */
    async_pay(_self, best_reserve, info.src, info.src_contract, (name{info.sender}).to_string());

    SEND_INLINE_ACTION(*this, trade2, {_self, "active"_n},
                       {best_reserve, info, info.src, dest, balance_pre});
}

ACTION Network::trade2(name reserve, trade_info info, asset src, asset dest, asset balance_pre) {
    require_auth(_self);  // can only be called internally

    /* verify dest balance was indeed added to dest account */
    auto balance_post = get_balance(info.sender, info.dest_contract, info.dest.symbol);
    eosio_assert(balance_post > balance_pre, "post balance not bigger than pre balance.");
    asset balance_diff = balance_post - balance_pre;
    eosio_assert(balance_diff >= dest, "trade dest amount not added.");

    /* update token stats */
    bool buy = (src.symbol == EOS_SYMBOL);
    asset eos = buy ? src : dest;
    asset token = buy ? dest : src;

    tokenstats_type tokenstats_table_inst(_self, _self.value);
    auto itr = tokenstats_table_inst.find(token.symbol.raw());
    tokenstats_table_inst.modify(itr, _self, [&](auto& s) {
```

```
        s.token_counter += token;
        s.eos_counter += eos;
    });

    state_type state_inst(_self, _self.value);
    name listener = state_inst.get().listener;
    if ((listener != name()) && (listener != "eosio"_n)) {
        action {permission_level{_self, "active"_n},
                listener,
                "posttrade"_n,
                make_tuple(src, dest, reserve, info.sender)}.send();
    }

    SEND_INLINE_ACTION(*this, trade3, {_self, "active"_n}, {});
}

ACTION Network::trade3() {
    require_auth(_self);  // can only be called internally
    reentrancy_check(false);
} /* end of trade process */

void Network::async_search_best_rate(reservespert &token_entry, asset src) {
    for (int i = 0; i < token_entry.reserve_contracts.size(); i++) {
        auto reserve = token_entry.reserve_contracts[i];
        action {permission_level{_self, "active"_n},
                reserve,
                "getconvrate"_n,
                make_tuple(src)}.send();
    }
}

void Network::get_best_rate_results(asset src, symbol dest_symbol, double &rate, name &reserve) {
    /* read stored rates from all reserves that hold the pair and decide on the best one */
    reservespert_type reservespert_table_inst(_self, _self.value);
    symbol token_symbol = (src.symbol == EOS_SYMBOL) ? dest_symbol : src.symbol;
    auto reservespert_entry = reservespert_table_inst.get(token_symbol.raw());

    rate = 0;
    for (int i = 0; i < reservespert_entry.reserve_contracts.size(); i++) {
        auto current_reserve = reservespert_entry.reserve_contracts[i];
        auto current_rate_entry = rate_type(current_reserve, current_reserve.value).get();
```

```
        if (current_rate_entry.stored_rate > rate) {
            reserve = current_reserve;
            rate = current_rate_entry.stored_rate;
        }
    }
}

void Network::reentrancy_check(bool enter) {
    state_type state_inst(_self, _self.value);
    auto s = state_inst.get();
    eosio_assert(((!s.during_trade && enter) || (s.during_trade && !enter)),
                 "re-entrancy during a trade");
    s.during_trade = enter;
    state_inst.set(s, _self);
}

Network::state_type Network::get_state_assert_admin() {
    state_type state_inst(_self, _self.value);
    eosio_assert(state_inst.exists(), "init not called yet");
    require_auth(state_inst.get().admin);
    return state_inst;
}

void Network::parse_memo(string memo, trade_info &res) {
    auto parts = split(memo, ",");
    eosio_assert(parts.size() == EXPECTED_MEMO_LENGTH, "wrong memo length");

    auto sym_parts = split(parts[0], " ");
    eosio_assert(sym_parts.size() == EXPECTED_SYMBOL_PARTS, "wrong num of symbol parts");
    res.dest = asset(0, symbol(sym_parts[1].c_str(), stoi(sym_parts[0].c_str())));

    res.dest_contract = name(parts[1].c_str());
    res.min_conversion_rate = stof(parts[2].c_str());
}

trade_info Network::create_trade_info(string memo, name from, asset src, name src_contract) {
    auto res = trade_info();
    res.sender = from;
    res.src = src;
    res.src_contract = src_contract;

    parse_memo(memo, res);
```

```
    return res;
}

void Network::transfer(name from, name to, asset quantity, string memo) {
    if (to != _self) return;

    state_type state_inst(_self, _self.value);
    if (!state_inst.exists()) {
        /* if init not called yet don't trade, instead allow anyone to deposit. */
        return;
    }

    auto state = state_inst.get();
    if (from == state.admin || from == STAKE_ACCOUNT || from == RAM_ACCOUNT) {
        /* admin and system accounts can deposit funds, but not trade */
        return;
    } else {
        /* this is a trade */
        trade(from, to, quantity, memo, state);
        return;
    }
    eosio_assert(false, "unreachable code");
}

extern "C" {
    [[noreturn]] void apply(uint64_t receiver, uint64_t code, uint64_t action) {
        if (action == "transfer"_n.value && code != receiver) {
            eosio::execute_action(eosio::name(receiver), eosio::name(code), &Network::transfer);
        } else if (code == receiver) {
            switch (action) {
                EOSIO_DISPATCH_HELPER( Network, (init)(setadmin)(setenable)(setlistener)(addreserve)
                                                (listpairres)(withdraw)(trade1)(trade2)(trade3)
                                                (getexprate)(storeexprate))
            }
        }
        eosio_exit(0);
    }
}
```

Network/Network.hpp

```
#pragma once
```

```
#include <vector>
#include <string>
#include <eosiolib/eosio.hpp>
#include <eosiolib/print.hpp>
#include <eosiolib/asset.hpp>
#include <eosiolib/time.hpp>
#include "../Common/common.hpp"

#define EXPECTED_MEMO_LENGTH 3
#define EXPECTED_SYMBOL_PARTS 2

using namespace eosio;

struct trade_info {
    name        sender;
    name        src_contract;
    asset       src;
    name        dest_contract;
    asset       dest;
    double      min_conversion_rate;
};

CONTRACT Network : public contract {
    public:
        using contract::contract;

        TABLE state {
            name        admin;
            name        eos_contract;
            name        listener;
            bool        enabled;
            bool        during_trade;
        };

        TABLE reserve {
            name        contract;
            uint64_t    num_tokens;
            uint64_t    primary_key() const { return contract.value; }
        };

        TABLE reservespert {
```

```
    symbol          symbol;
    name            token_contract;
    vector<name>    reserve_contracts;
    uint64_t        primary_key() const { return symbol.raw(); }
};

TABLE tokenstats {
    asset           token_counter;
    asset           eos_counter;
    uint64_t        primary_key() const { return token_counter.symbol.raw(); }
};

TABLE rate {
    double      stored_rate;
    asset       dest;
};

typedef eosio::singleton<"state"_n, state> state_type;
typedef eosio::multi_index<"reserve"_n, reserve> reserves_type;
typedef eosio::multi_index<"reservespert"_n, reservespert> reservespert_type;
typedef eosio::multi_index<"tokenstats"_n, tokenstats> tokenstats_type;
typedef eosio::singleton<"rate"_n, rate> rate_type;

/**
 * Init the contract.
 * Should be called right after deploying the contract.
 * Can only be called once, and only by the network account authority (owner).
 *
 * @param admin - the only account that can deposit/withdraw tokens,
 * and configure the network contract.
 * @param eos_contract - account of EOS native token, usually eosio.token.
 * @param listener - listener contract to call hooks.
 * @param enable - whether to initiate the network in an operating state,
 * or otherwise wait for a setenable operation.
 */
ACTION init(name admin, name eos_contract, name listener, bool enable);

/**
 * Change the admin account.
 * Can only be called by the admin.
 *
 * @param admin - the new admin account.
```

```
 */
ACTION setadmin(name admin);

/**
 * Enable or disable the network.
 * Can only be called by the admin.
 *
 * @param enable - enable or disable.
 */
ACTION setenable(bool enable);

/**
 * Set listener contract to process hooks.
 * Can be used for various tasks such as promotions, traffic calculations, fee burning.
 * Can only be called by the admin.
 *
 * @param listener - listener contract.
 */
ACTION setlistener(name listener);

/**
 * Add/Remove a reserve to/from the network.
 * Can only be called by the admin.
 *
 * @param reserve - account of the reserve contract.
 * @param add - add or remove.
 */
ACTION addreserve(name reserve, bool add);

/**
* List/Unlist a trade pair on the network.
* Can only be called by the admin.
*
* @param reserve - account of the reserve where the pair is added/removed to/from.
* @param token_symbol - symbol of the added/removed token, which will be paired with EOS.
* @param token_contract - account contract implementing the added/removed token.
* @param add - add or remove the pair.
*/
ACTION listpairres(name reserve, symbol token_symbol, name token_contract, bool add);

/**
 * Withdraw funds from the network account. Can only be called by the admin.
```

```
     *
     * @param to - account to withdraw to.
     * @param quantity - asset to withdraw.
     * @param dest_contract - account implementing the withdrawn token.
     * @param memo - optional memo to be added to the end withdraw transfer operation.
     */
    ACTION withdraw(name to, asset quantity, name dest_contract, string memo);

    /**
     * Get expected rate for a specific pair.
     * Result is written to the "rate" table.
     * Should only be used for on chain integration.
     * Only in such on-chain contracts integration the table can be read in atomic manner.
     *
     * @param src - src asset in the pair to query rate for.
     * @param dest_symbol - destination token symbol for the rate query.
     */
    ACTION getexprate(asset src, symbol dest_symbol);

    /*
     * The following functions are internal actions.
     * They are purposed to only be called internally by the network contract.
     * We implement these as separate actions (and not regular functions/logic),
     * since when calling a preceding action, the only way to assure their logic
     * would happen after that action is to make them a separate action on their own.
     */

    /** internal */
    ACTION storeexprate(asset src, symbol dest_symbol);

    /** internal */
    ACTION trade1(trade_info info);

    /** internal */
    ACTION trade2(name reserve, trade_info info, asset src, asset dest, asset balance_pre);

    /** internal */
    ACTION trade3();

    /**
     * Notification handler for transfer events from/to this contract.
     * Before init() is called anyone can deposit to the contract.
```

```
         * After init() is called only the admin can deposit.
         * At that stage any other transfer to the contract is regarded as a trade attempt,
         * and expected to have a valid memo for a trade.
         * Note that the memo's min conversion rate parameter is the way for
         * the user to ensure that the actual rate for the trade is not less
         * than what he expects.
         *
         * @param from - sender.
         * @param to - recipient, this contract.
         * @param quantity - sent asset.
         * @param memo - Expected as "<dest symbol>,<dest contract>,<min conversion rate>"
         * For example: "4 KARMA,therealkarma,7200.0000"
         */
        void transfer(name from, name to, asset quantity, string memo);

    private:
        void trade(name from, name to, asset src, string memo, state &current_state);

        void async_search_best_rate(reservespert &token_entry, asset src);

        void get_best_rate_results(asset src, symbol dest_symbol, double &rate, name &reserve);

        void reentrancy_check(bool enter);

        state_type get_state_assert_admin();

        void parse_memo(string memo, trade_info &info);

        trade_info create_trade_info(string memo, name from, asset src, name _code);
};
```

Reserve/AmmReserve/AmmReserve.cpp

```
#include "AmmReserve.hpp"
#include "liquidity.hpp"

using namespace eosio;

ACTION AmmReserve::init(name    admin,
                        name    network_contract,
                        symbol  token_symbol,
                        name    token_contract,
```

```
                        name    eos_contract,
                        bool    enable_trade) {
    eosio_assert(is_account(admin), "admin account does not exist");
    eosio_assert(is_account(network_contract), "network account does not exist");
    eosio_assert(is_account(token_contract), "token account does not exist");
    eosio_assert(is_account(eos_contract), "eos contract does not exist");

    require_auth(_self);

    state_type state_inst(_self, _self.value);
    eosio_assert(!state_inst.exists(), "init already called");

    state new_state;
    new_state.admin = admin;
    new_state.network_contract = network_contract;
    new_state.token_symbol = token_symbol;
    new_state.token_contract = token_contract;
    new_state.eos_contract = eos_contract;
    new_state.trade_enabled = enable_trade;
    state_inst.set(new_state, _self);
}

ACTION AmmReserve::quickset(double p) {
    auto state_inst = get_state_assert_admin();
    params_type params_inst(_self, _self.value);
    params new_params;

    new_params.p_min = p / 2.0;
    new_params.max_eos_cap_buy = asset(MAX_AMOUNT, EOS_SYMBOL);
    new_params.max_eos_cap_sell = asset(MAX_AMOUNT, EOS_SYMBOL);
    new_params.profit_percent = 0.0;
    new_params.ram_fee = 0.0;
    new_params.max_sell_rate = p * 2.0;
    new_params.min_sell_rate = p / 2.0;
    new_params.max_buy_rate = 1.0 / new_params.min_sell_rate;
    new_params.min_buy_rate = 1.0 / new_params.max_sell_rate;
    new_params.fee_wallet = name();

    /* (p/p_min) = 2.0 = e^(rE) => r = ln(2)/E */
    asset eos_balance = get_balance(_self, state_inst.get().eos_contract, EOS_SYMBOL);
    eosio_assert(eos_balance.is_valid() && eos_balance.amount > 0, "no balance");
    new_params.r = 0.69314 / amount_to_damount(eos_balance.amount, EOS_PRECISION);
```

```
    params_inst.set(new_params, _self);
}

ACTION AmmReserve::setparams(double r,
                             double p_min,
                             asset  max_eos_cap_buy,
                             asset  max_eos_cap_sell,
                             double profit_percent,
                             double ram_fee,
                             double max_sell_rate,
                             double min_sell_rate,
                             name   fee_wallet) {
    get_state_assert_admin();

    eosio_assert(r >= 0, "illegal r");
    eosio_assert(p_min > 0, "illegal p_min");

    eosio_assert(max_eos_cap_buy.is_valid() && max_eos_cap_buy.amount > 0,
                 "illegal max_eos_cap_buy");
    eosio_assert(max_eos_cap_sell.is_valid() && max_eos_cap_sell.amount > 0,
                 "illegal max_eos_cap_sell");

    eosio_assert(profit_percent >= 0 && profit_percent < 100.0, "illegal profit_percent");
    eosio_assert(ram_fee >= 0, "illegal ram_fee");
    if (profit_percent || ram_fee) {
        eosio_assert(((fee_wallet != name()) && (fee_wallet != "eosio"_n)), "no fee wallet");
    }

    eosio_assert(max_sell_rate > 0, "illegal max_sell_rate");
    eosio_assert(min_sell_rate >= 0, "illegal min_sell_rate");
    eosio_assert(min_sell_rate <= max_sell_rate, "max_sell_rate smaller than min_sell_rate ");

    params_type params_inst(_self, _self.value);
    params new_params;
    new_params.r = r;
    new_params.p_min = p_min;
    new_params.max_eos_cap_buy = max_eos_cap_buy;
    new_params.max_eos_cap_sell = max_eos_cap_sell;
    new_params.profit_percent = profit_percent;
    new_params.ram_fee = ram_fee;
    new_params.max_buy_rate = 1.0 / min_sell_rate;
```

```
    new_params.min_buy_rate = 1.0 / max_sell_rate;
    new_params.max_sell_rate = max_sell_rate;
    new_params.min_sell_rate = min_sell_rate;
    new_params.fee_wallet = fee_wallet;
    params_inst.set(new_params, _self);
}

ACTION AmmReserve::setadmin(name admin) {
    eosio_assert(is_account(admin), "new admin account does not exist");

    auto state_inst = get_state_assert_admin();

    auto s = state_inst.get();
    s.admin = admin;
    state_inst.set(s, _self);
}

ACTION AmmReserve::setnetwork(name network_contract) {
    eosio_assert(is_account(network_contract), "network account does not exist");

    auto state_inst = get_state_assert_admin();

    auto s = state_inst.get();
    s.network_contract = network_contract;
    state_inst.set(s, _self);
}

ACTION AmmReserve::setenable(bool enable) {
    auto state_inst = get_state_assert_admin();

    auto s = state_inst.get();
    s.trade_enabled = enable;
    state_inst.set(s, _self);
}

ACTION AmmReserve::getconvrate(asset src) {
    eosio_assert(src.is_valid(), "src amount");
    eosio_assert(src.amount >= 0, "src amount can not be negative");

    /* for simplicity and safety only network can get conversion rate */
    state_type state_inst(_self, _self.value);
    eosio_assert(state_inst.exists(), "init not called yet");
```

```
    require_auth(state_inst.get().network_contract);

    asset dest = asset();
    double charged_fee;
    double rate_result = reserve_get_conv_rate(src, false, dest, charged_fee);

    rate_type rate_inst(_self, _self.value);
    rate s = {rate_result, dest};
    rate_inst.set(s, _self);
}

ACTION AmmReserve::withdraw(name to, asset quantity, name dest_contract, string memo) {
    eosio_assert(is_account(to), "to account does not exist");
    eosio_assert(is_account(dest_contract), "dest contract does not exist");
    eosio_assert(quantity.is_valid() && quantity.amount > 0, "illegal quantity");

    get_state_assert_admin();
    async_pay(_self, to, quantity, dest_contract, memo);
}

double AmmReserve::reserve_get_conv_rate(asset src,
                                         bool subtract_src,
                                         asset &dest,
                                         double &charged_fee) {
    dest = asset();

    state_type state_inst(_self, _self.value);
    /* if reserve not ready return gracefully (store 0 rate) to continue queries in network */
    if (!state_inst.exists()) return 0;
    auto state = state_inst.get();
    if (!state.trade_enabled) return 0;

    /* verify params were set */
    params_type params_inst(_self, _self.value);
    if (!params_inst.exists()) return 0;
    auto params = params_inst.get();

    bool buy = (EOS_SYMBOL == src.symbol) ? true : false;

    asset eos_balance = get_balance(_self, state.eos_contract, EOS_SYMBOL);
    if(subtract_src) {
        /* disregard eos src quantity, so it will not affect e used for rate calc. */
```

```
        if (src > eos_balance) return 0;
        eos_balance = eos_balance - src;
    }

    double rate = liquidity_get_rate(_self,
                                     eos_balance,
                                     buy,
                                     src,
                                     params.r,
                                     params.p_min,
                                     params.profit_percent,
                                     params.ram_fee,
                                     charged_fee);
    if (!rate || rate == INFINITY) return 0;

    double min_allowed_rate = buy ? params.min_buy_rate : params.min_sell_rate;
    double max_allowed_rate = buy ? params.max_buy_rate : params.max_sell_rate;
    if ((rate > max_allowed_rate) || (rate < min_allowed_rate) || (rate > MAX_RATE)) return 0;

    symbol dest_symbol = buy ? state.token_symbol : EOS_SYMBOL;
    dest = calc_dest(rate, src, dest_symbol);

    asset eos_trade_quantity = buy ? src : dest;
    asset max_eos_cap = buy ? params.max_eos_cap_buy : params.max_eos_cap_sell;
    if (eos_trade_quantity > max_eos_cap) {
        dest = asset();
        return 0;
    }

    /* make sure reserve has enough of the dest token */
    name dest_contract = buy ? state.token_contract : state.eos_contract;
    if (get_balance(_self, dest_contract, dest_symbol) < dest) {
        dest = asset();
        return 0;
    }

    return rate;
}

void AmmReserve::trade(name from, asset src, string memo, name code, state &state) {
    eosio_assert(state.trade_enabled, "trade disabled");
    eosio_assert(from == state.network_contract, "only network can perform a trade");
```

```
    bool buy = (src.symbol == EOS_SYMBOL) ? true : false;

    name expected_src_contract = buy ? state.eos_contract : state.token_contract;
    eosio_assert(code == expected_src_contract, "wrong src contract");

    eosio_assert(src.is_valid(), "invalid transfer");
    eosio_assert(src.amount > 0, "src amount must be positive");
    eosio_assert(src.symbol == EOS_SYMBOL || src.symbol == state.token_symbol, "unrecognized src");

    params_type params_inst(_self, _self.value);
    eosio_assert(params_inst.exists(), "params were not set");
    auto params = params_inst.get();

    name receiver = name(memo.c_str());
    eosio_assert(receiver != _self, "receiver can not be current contract");

    symbol dest_symbol = buy ? state.token_symbol : EOS_SYMBOL;
    name dest_contract = buy ? state.token_contract : state.eos_contract;

    /* get conversion rate again */
    asset dest = asset();
    double charged_fee = 0;
    double conversion_rate = reserve_get_conv_rate(src, buy, dest, charged_fee);
    eosio_assert(conversion_rate > 0, "conversion rate must be bigger than 0");
    eosio_assert(conversion_rate < MAX_RATE, "fail overflow validation");

    async_pay(_self, receiver, dest, dest_contract, "trade dest");

    asset charged_fee_asset = asset(damount_to_amount(charged_fee, EOS_PRECISION), EOS_SYMBOL);
    if (charged_fee_asset.amount > 0) {
        async_pay(_self, params.fee_wallet, charged_fee_asset, state.eos_contract, "send fee");
    }
}

AmmReserve::state_type AmmReserve::get_state_assert_admin() {
    state_type state_inst(_self, _self.value);
    eosio_assert(state_inst.exists(), "init not called yet");
    require_auth(state_inst.get().admin);
    return state_inst;
}

void AmmReserve::transfer(name from, name to, asset quantity, string memo) {
```

```
    if (to != _self) return;

    state_type state_inst(_self, _self.value);
    if (!state_inst.exists()) {
        /* if init not called yet don't trade, instead allow anyone to deposit. */
        return;
    }

    auto state = state_inst.get();
    if (from == state.admin || from == STAKE_ACCOUNT || from == RAM_ACCOUNT) {
        /* admin and system accounts can deposit funds, but not trade */
        return;
    } else {
        trade(from, quantity, memo, _code, state);
        return;
    }
    eosio_assert(false, "unreachable code");
}

extern "C" {
    [[noreturn]] void apply(uint64_t receiver, uint64_t code, uint64_t action) {
        if (action == "transfer"_n.value && code != receiver) {
            eosio::execute_action(eosio::name(receiver), eosio::name(code), &AmmReserve::transfer);
        } else if (code == receiver) {
            switch (action) {
                EOSIO_DISPATCH_HELPER(AmmReserve, (init)(quickset)(setparams)(setadmin)(setnetwork)
                                                  (setenable)(getconvrate)(withdraw))
            }
        }
        eosio_exit(0);
    }
}
```

Reserve/AmmReserve/AmmReserve.hpp

```
#pragma once

#include <string>
#include <eosiolib/eosio.hpp>
#include <eosiolib/print.hpp>
#include <eosiolib/asset.hpp>
#include <eosiolib/singleton.hpp>
```

```
#include "../../Common/common.hpp"

CONTRACT AmmReserve : public contract {
    public:
        using contract::contract;

        TABLE state {
            name        admin;
            name        network_contract;
            symbol      token_symbol;
            name        token_contract;
            name        eos_contract;
            bool        trade_enabled;
        };

        TABLE params {
            double      r;
            double      p_min;
            asset       max_eos_cap_buy;
            asset       max_eos_cap_sell;
            double      profit_percent;
            double      ram_fee;
            double      max_buy_rate;
            double      min_buy_rate;
            double      max_sell_rate;
            double      min_sell_rate;
            name        fee_wallet;
        };

        TABLE rate {
            double      stored_rate;
            asset       dest;
        };

        typedef eosio::singleton<"state"_n, state> state_type;
        typedef eosio::singleton<"params"_n, params> params_type;
        typedef eosio::singleton<"rate"_n, rate> rate_type;

        /**
         * Init the reserve.
         * Should be called right after deploying the contract.
         * Can only be called once, and only by the reserve account authority.
```

```
     *
     * @param admin - the only account that can deposit/withdraw tokens,
     * and configure the reserve contract.
     * @param network_contract - contract of the network the reserve is listed on.
     * Only the network contract is allowed to trade through the reserve.
     * @param token_symbol - the symbol of the token traded on the reserve.
     * @param token_contract - the contract implementing the token traded on the reserve.
     * @param eos_contract - account of eos native token, usually eosio.token.
     * @param enable_trade - whether to initiate the reserve in an operating state,
     * or otherwise wait for a setenable operation.
     */
    ACTION init(name    admin,
                name    network_contract,
                symbol  token_symbol,
                name    token_contract,
                name    eos_contract,
                bool    enable_trade);

    /**
     * Set reserve parameters quickly, using default values.
     * Can only be called by the reserve admin.
     * Liquidity rate (r) will be auto calculated according to the contract's EOS amount.
     * Supported price movements will be half to twice of the initial price.
     * No cap restrictions per single trade will be enforced.
     * Profit percent will be set to 0.
     *
     * @param p - initial token to EOS sell price.
     */
    ACTION quickset(double p);

    /**
    * Set reserve parameters.
    * Can only be called by the reserve admin.
    *
    * @param r - liquidity rate.
    * @param p_min - minimum supported price, in token sell convention.
    * @param max_eos_cap_buy - maximum single buy amount in EOS units.
    * @param max_eos_cap_sell - maximum single sell amount in EOS units.
    * @param profit_percent - percent of reserve tokens profit per trade.
    * @param ram_fee - eos fee per eos->token trade, purposed to cover transfer ram expenses.
    * @param max_sell_rate - maximum rate allowed, in token sell convention.
    * @param min_sell_rate - minimum rate allowed, in token sell convention.
```

```
   * @param fee_wallet - account to send profit and fee to.
   */
   ACTION setparams(double r,
                    double p_min,
                    asset  max_eos_cap_buy,
                    asset  max_eos_cap_sell,
                    double profit_percent,
                    double ram_fee,
                    double max_sell_rate,
                    double min_sell_rate,
                    name   fee_wallet);

   /**
    * Change the admin account.
    * Can only be called by the reserve admin.
    *
    * @param admin - the new admin account.
    */
   ACTION setadmin(name admin);

   /**
    * Change the registered network contract.
    * Can only be called by the reserve admin.
    * Only the registered network account can send trades to the reserve.
    *
    * @param network_contract - the new network contract.
    */
   ACTION setnetwork(name network_contract);

   /**
    * Enable or disable the reserve.
    * Can only be called by the reserve admin.
    * When disabled, both trade and get conversion rate are disabled.
    *
    * @param enable - enable or disable.
    */
   ACTION setenable(bool enable);

   /**
    * Get conversion rate.
    * Can only be called by the network contract, as registered in the reserve.
    * Result will be written to the trade table.
```

```
         *
         * @param src - src asset for the rate query. Can be either EOS or the reserve's token.
         */
        ACTION getconvrate(asset src);

        /* Withdraw funds from the reserve account.
         * Can only be called by the reserve admin.
         *
         * @param to - account to withdraw to.
         * @param quantity - asset to withdraw.
         * @param dest_contract - account implementing the withdrawn token.
         * @param memo - optional memo to be added to the withdraw end transfer operation.
         */
        ACTION withdraw(name to, asset quantity, name dest_contract, string memo);

        /* Notification handler for transfer events from/to this contract.
         * Before init() is called anyone can deposit to the contract.
         * After init() is called only the contract admin can deposit.
         * Any other transfer to the contract is regarded as a trade attempt.
         * A trade is expected to come from the network account and have a valid memo.
         *
         * @param name - sender.
         * @param to - recipient, this contract.
         * @quantity - sent asset
         * @memo - for trades expected as "<dest account>". For example: "bob111111111".
         */
        void transfer(name from, name to, asset quantity, string memo);

    private:
        double reserve_get_conv_rate(asset src,
                                     bool subtract_src,
                                     asset &dest,
                                     double &charged_fee);

        void trade(name from, asset src, string memo, name code, state &state);

        state_type get_state_assert_admin();
};
```

Reserve/AmmReserve/liquidity.hpp

```
#pragma once
```

```
#include <eosiolib/eosio.hpp>
#include <eosiolib/asset.hpp>
#include "../../Common/common.hpp"

#include <math.h>

using namespace eosio;

struct liq_info {
    double      r;
    double      p_min;
    double      profit_percent;
    double      ram_fee;
};

double p_of_e(struct liq_info &info, double e) {
    return info.p_min * exp(info.r * e);
}

double get_delta_t(struct liq_info &info, double e, double delta_e) {
    return (-1) * (exp(-info.r * delta_e) - 1.0) / (info.r * p_of_e(info, e));
}

double get_delta_e(struct liq_info &info, double e, double delta_t) {
    return ((log(1 + info.r * p_of_e(info, e) * delta_t)) / info.r);
}

double liquidity_get_rate(name self_contract,
                          asset eos_balance,
                          bool buy,
                          asset src,
                          double r,
                          double p_min,
                          double profit_percent,
                          double ram_fee,
                          double &charged_fee) {

    liq_info info = {r, p_min, profit_percent, ram_fee};
    double e = asset_to_damount(eos_balance);
    double src_damount = asset_to_damount(src);
    double dest_damount;
```

```
    double rate;

    if (!src_damount) {
        double pre_profit_rate = buy ? (1 / p_of_e(info, e)) : p_of_e(info, e);
        rate = ((100.0 - info.profit_percent) * pre_profit_rate) / 100.0;
    } else {
        if (buy) {
            charged_fee = (info.profit_percent * src_damount) / 100.0;
            if (info.ram_fee >= (src_damount - charged_fee)) {
                charged_fee = 0;
                return 0;
            }
            charged_fee += info.ram_fee;
            dest_damount = get_delta_t(info, e, src_damount - charged_fee);
        } else {
            double delta_e = get_delta_e(info, e, src_damount);
            charged_fee = (info.profit_percent * delta_e) / 100.0;
            dest_damount = delta_e - charged_fee;
        }
        rate = dest_damount / src_damount;
    }
    return rate;
}
```

Official Website
www.slowmist.com

E-mail
team@slowmist.com

Twitter
@SlowMist_Team

WeChat Official Account