密级 ★ | 项目组公开

智能合约审计报告

athenastoken

主测人：代其

知道创宇

版本说明

修订人	修订内容	修订时间	版本号	审阅人
代其	编写文档	**2019/06/21**	V1.0	代其

文档信息

文档名称	**文档版本号**	文档编号	**保密级别**
智能合约审计报告	V1.0	【SLKJ-DMSJ-20190621】	项目组公开

版权声明

目录

1. 综述

本次报告有效测试时间是从 2019 年 06 月 20 日开始到 2019 年 06 月 21 日结束，在此期间针对**智能合约代码**的安全性和规范性进行审计并以此作为报告统计依据。

此次测试中，知道创宇工程师对智能合约的常见漏洞（见第三章节）进行了全面的分析，发现合约代码中存在逻辑设计问题，综合评定为安全。

本次智能合约安全审计结果： 通过

由于本次测试过程在非生产环境下进行，所有代码均为最新备份，测试过程均与相关接口人进行沟通，并在操作风险可控的情况下进行相关测试操作，以规避测试过程中的生产运营风险、代码安全风险。

本次测试的目标信息：

模块名称	
Token 名称	athenastoken
代码类型	发行代币代码
合约地址	https://eospark.com/account/athenastoken
链接地址	https://eospark.com/account/athenastoken
代码语言	C++

本次项目测试人员信息：

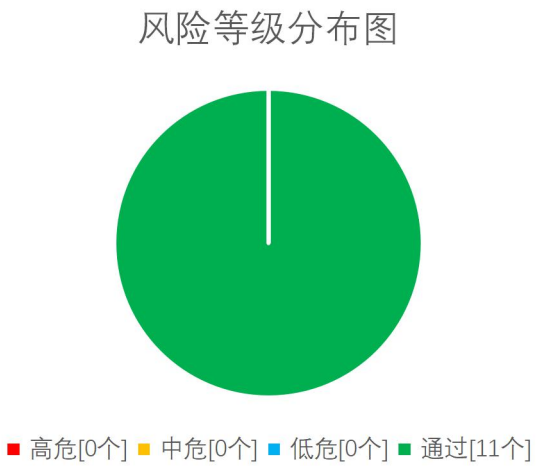
姓名	邮箱/联系方式	职务
代其	daiq@knownsec.com	安全工程师

2. 代码漏洞分析

2.1. 漏洞等级分布

本次漏洞风险按等级统计：

漏洞风险等级个数统计表			
高危	中危	低危	通过
0	0	0	11



2.2. 审计结果汇总

审计结果			
测试项目	测试内容	状态	描述
智能合约	**编码规范性检测**	通过	检查编码信息泄漏、类型转换等安全
	数值溢出检测	通过	检查 balance 使用安全、asset 类溢出校验
	权限限制检测	通过	检查各操作访问权限控制
	API 函数校验	通过	EOS API 函数的参数类型。在参数传递的过程中是否存在对相应的参数类型输入范围进行检测
	常规代码风险检测	通过	检测数据库 API 使用是否存在不严谨的情况
	逻辑设计检测	通过	检查代码中是否存在与业务设计相关的安全问题
	变量初始化风险检测	通过	检查代码在变量初始化是否存在缺陷
	Apply 校验检测	通过	检查在处理合约调用时，是否每个 action 与 code 均满足关联要求。
	Transfer 假通知检测	通过	检查智能合约代码中是否存在假通知漏洞
	随机数检测	通过	检查随机数生成算法不要引入可控或者可预测的种子
	回滚攻击检测	通过	检查是否存在回滚漏洞

3. 代码审计结果分析

3.1. 编码规范性检测【安全】

主要依据 EOS 官方给出的 eosio.token 示例进行实现对代币合约的编码规范性进行审查，具体如：

1. 合约中使用官方提供的 *asset* 数据结构描述代币，对代币的算数运算同样利用 *asset* 完成。

```
void transfer(name from, name to, asset quantity, string memo);
```

2. 在使用 *multi_index* 的 *find* 函数时，一定要进行返回值的检查。

```
302         auto playerLookup = _playergames.find(from.value);
303         if (playerLookup != _playergames.end()){ //game exists before
304             auto currentgame = _playergames.get(from.value);
305             eosio_assert(((currentgame.state == STATE_DRAW) || (currentgame.sta
306             _playergames.erase(playerLookup);
307         }
```

3. 对所有输入都通过断言检查有效性，调用 API 函数前，检查参数类型和大小。

```
341 void v_poker::drawcards(const name account, vector<uint8_t> cardschoice) //0
342 {
343     require_auth(account); //check player account is valid
344
345     eosio_assert(cardschoice.size()==5, "Invalid card choices");
```

＊本项检查以注释的方式在合约源码中标注即可，不作为漏洞对待。

检测过程：经检测，该应用均检测了返回值。

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

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

检测结果：安全

3.2. 溢出检测

3.2.1. 数值溢出检测【安全】

检查合约是否存在对代币数值进行算数运算时未进行安全检查。在误操作时容易产生整型溢出错误，可能导致代币量归零甚至变成负数的严重后果！风险。问题代码样例：

样例一：

```
void token::add_balance( account_name owner, int64_t value, symbol_name sym_name, account_name ram_payer )
{
   accounts to_acnts( _self, owner );
   auto to = to_acnts.find(sym_name);
   if( to == to_acnts.end() ) {
      to_acnts.emplace( ram_payer, [&]( auto& a ){
        a.balance = value;
      });
   } else {
      to_acnts.modify( to, 0, [&]( auto& a ) {
        a.balance += value;
      });
   }
}
```

上述代码在 add_balance（增加资产）函数中存在整数溢出问题，在代币未开启白名单的情况下，使用 modify 方法修改项目后，修改代币数量中未对代币数量交易进行检测，导致数值运算存在溢出的风险。

样例二：

transfer 函数源码，同时向 4 个人转账：

```
typedef  struct acnts {
    account_name    name0;
    account_name    name1;
    account_name    name2;
    account_name    name3;
}account_names;

void transfer(symbol_name symbol, account_name from, account_names to, uint64_t balance) {
    print("transfer");
    require_auth(from);
    account fromaccount;

    require_recipient(from);
    require_recipient(to.name0);
    require_recipient(to.name1);
    require_recipient(to.name2);
    require_recipient(to.name3);

    eosio_assert(is_balance_within_range(balance), "invalid balance");
    eosio_assert(balance > 0, "must transfer positive balance");

    uint64_t amount = balance * 4;

    int itr = db_find_i64(_self, symbol, N(table), from);
    eosio_assert(itr >= 0, "Sub--wrong name");
    db_get_i64(itr, &fromaccount, sizeof(account));
    eosio_assert(fromaccount.balance >= amount, "overdrawn balance");

    sub_balance(symbol, from, amount);

    add_balance(symbol, to.name0, balance);
    add_balance(symbol, to.name1, balance);
    add_balance(symbol, to.name2, balance);
    add_balance(symbol, to.name3, balance);
}
```

假设攻击者调用该合约的 transfer 函数同时向 4 个人进行转账操作，并将 balance 参数的值设为（2^{63}），函数调用过程如图所示：

```
$ cleos push action lianan2 transfer '[usa,tester,[tester1 tester2 tester3 tester4], 9223372036854775808
b4155c9b36d10a34adb7a4bd40c33fdc3c421892689b1a409ac4c86ae0fea  280 bytes  121856 cycles
          {"symbol":"usa","from":"tester","to":{"name0":"tester1","name1":"tester2","name2":"tester3","n


          {"symbol":"usa","from":"tester","to":{"name0":"tester1","name1":"tester2","name2":"tester3","n

          {"symbol":"usa","from":"tester","to":{"name0":"tester1","name1":"tester2","name2":"tester3","n

          {"symbol":"usa","from":"tester","to":{"name0":"tester1","name1":"tester2","name2":"tester3","n

          {"symbol":"usa","from":"tester","to":{"name0":"tester1","name1":"tester2","name2":"tester3","n

          {"symbol":"usa","from":"tester","to":{"name0":"tester1","name1":"tester2","name2":"tester3","n
```

接着，查询以上地址余额可发现，被转账人（tester）的余额（100）没有减少，而接收者（tester1、tester2、tester3、tester4）的账户余额由于 amount 变量的溢出产生了非常大的数目（2^{63}），如图 3 所示:

```
ethan@ubuntu:~/eos/my_contracts/token$ cleos push action lianan2 check '[usa, tester1]' -p user
executed transaction: 9e899ae2f4532496040a412aa14bd9b8ec494e4f3c14310c6134ab3acdafc11a  240 byte
#       lianan2 <= lianan2::check              {"symbol":"usa","name":"tester1"}
>> 9223372036854775908
ethan@ubuntu:~/eos/my_contracts/token$ cleos push action lianan2 check '[usa, tester2]' -p user
executed transaction: 1ece6c05c5f0f49666f874e4f85a1da4956ccb5c3d7afd5be8190fafe7539665  240 byte
#       lianan2 <= lianan2::check              {"symbol":"usa","name":"tester2"}
>> 9223372036854775908
ethan@ubuntu:~/eos/my_contracts/token$ cleos push action lianan2 check '[usa, tester3]' -p user
executed transaction: f5c1c7a95973dc2770e041c1a57766f17d7ec31c7f773e4fbc844fe8c784ad95  240 byte
#       lianan2 <= lianan2::check              {"symbol":"usa","name":"tester3"}
>> 9223372036854775908
ethan@ubuntu:~/eos/my_contracts/token$ cleos push action lianan2 check '[usa, tester4]' -p user
executed transaction: 75b843e9fd60ebfee5837aca38dde0b7a4b14cd143a6ce8c216ccc0a5d82a0a9  240 byte
#       lianan2 <= lianan2::check              {"symbol":"usa","name":"tester4"}
>> 9223372036854775908
ethan@ubuntu:~/eos/my_contracts/token$ cleos push action lianan2 check '[usa, tester]' -p user
executed transaction: b2c534e541ef5562bbdcaccf2642d0c7d2e70511208bb6e60693de0da6093b76  240 byte
#       lianan2 <= lianan2::check              {"symbol":"usa","name":"tester"}
>> 100
```

balance 是 uint64 数据类型，当取值为 2^{63} 时，由于小于 uint64 可取值的最大值，于是绕过了对 balance 的溢出边界检查；但是，当 amount=balance*4 计算时，amount 便发生了溢出，使其值等于 0，由于 amount 此时绕过了被减数大于减数的检查，从而实现不消耗被转账人的 balance 的情况下，让转账人的 balance 获取非常大的值（2^{63}）

解决方法：

合约开发者使用 EOS 区块链平台提供的智能合约编程 Math API 接口可防止该类型溢出漏洞。如合约开发者可将 uint 类型的数据，先转换成 doule 类型的数据，然后再使用 EOS 区块链平台提供的 Math API 中的 double_add、double_mult 等函数进行运算，最后将计算结果再转换成 uint 类型数据输出。

检测过程：经检测，未发现有溢出点

```
23
24  void token::add_balance( name owner, asset value, name ram_payer )
25  {
26     accounts to_acnts( _self, owner.value );
27     auto to = to_acnts.find( value.symbol.code().raw() );
28     if( to == to_acnts.end() ) {
29        to_acnts.emplace( ram_payer, [&]( auto& a ){
30          a.balance = value;
31        });
32     } else {
33        to_acnts.modify( to, same_payer, [&]( auto& a ) {
34          a.balance += value;
35        });
36     }
37  }
```

检测结果：安全

修复建议：无

3.2.2. asset 类溢出检测【安全】

asset 是 EOS 官方头文件中提供的用来代表货币资产（如官方货币 EOS 或自己发布的其它货币单位）的一个结构体。在使用 asset 进行乘法运算（operator *=）时，由于官方代码的 bug，导致其中的溢出检测无效化。造成的结果是，如果开发者在智能合约中使用了 asset 乘法运算，则存在发生溢出的风险。

问题代码存在于：contracts/eosiolib/asset.hpp 检测引用官方的头文件是否已更新：

```
      asset& operator*=( int64_t a ) {
        eosio_assert( a == 0 || (amount * a) / a == amount, "multiplication overflow
or underflow" );                                          <== (1)
        eosio_assert( -max_amount <= amount, "multiplication underflow" );  <= (2)
        eosio_assert( amount <= max_amount,  "multiplication overflow" );   <= (3)
        amount *= a;
        return *this;
     }
```

漏洞分析详情请看：https://paper.seebug.org/658/

检测过程：该合约中未使用乘法运算。

检测结果：安全

修复建议：无

3.3. 权限限制检测【安全】

检查合约是否正确对访问权限、方法调用进行权限检查，是否存在检查不严谨造成逻辑漏洞。

样例代码：

```
void token::issue( account_name to, asset quantity, string memo )
{
    /* Other Code */
    statstable.modify( st, 0, [&]( auto& s ) {
       s.supply += quantity;
    });
    /* Other Code */
}

void token::transfer( account_name from,account_name to,asset quantity)
{
    /* Other Code */
    sub_balance( from, quantity, st );
    add_balance( to, quantity, st, from );
}

void token::sub_balance( account_name owner, asset value, const currency_stat& st ) {
    /*Other Code*/
   eosio_assert( !st.can_freeze || !from.frozen, "account is frozen by issuer" );
   eosio_assert( !st.can_freeze || !st.is_frozen, "all transfers are frozen by issuer" );
   eosio_assert( !st.enforce_whitelist || from.whitelist, "account is not white listed" );
    /*Other Code*/
}
```

样例代币合约设置了冻结账户和代币的功能，然而用户将检查 “冻结” 的代码仅仅放在 transfer（转账）函数中，从而导致执行 issue（发行代币）的时候不受“冻结”状态影响，可以任意增发代币。

检测过程：该应用使用了对应的权限校验

```
void token::close( name owner, const symbol& symbol )
{
   require_auth( owner );
   accounts acnts( _self, owner.value );
   auto it = acnts.find( symbol.code().raw() );
   eosio_assert( it != acnts.end(), "Balance row already deleted or ne
ver existed. Action won't have any effect." );
   eosio_assert( it->balance.amount == 0, "Cannot close because the ba
lance is not zero." );
   acnts.erase( it );
}
```

检测结果：安全

修复建议：无

3.4. API 函数校验检测【安全】

检测 EOS API 函数的参数类型。在参数传递的过程中是否存在对相应的参数类型输入范围进行检测。

```
void token::freeze( account_name free_name , string strsym )
{
    symbol_type sym = string_to_symbol(strsym.size()+1, strsym.c_str());
    /*
     * other code
     */
}
```

如 string_to_symbol(uint8_t , const char *)，第一个参数传入的整型变量需要小于 256，若使用该 API 前未对输入进行检查，则可能导致整型溢出，从而导致操作了错误类型的代币，带来严重后果。

检测结果：安全
修复建议：无

3.5. 常规代码风险检测【安全】

数据库 API 使用不严谨，如 multi_index 中提供的 get 和 find。其中 get 会检查数据是否查询成功，数据未找到则断言退出，而 find 不会检查数据查询情况，需要用户自行判断，如果缺少判断直接使用将会导致指针使用问题。

```
    stats statstable( _self, sym_name );
    auto existing = statstable.find( sym_name );
    const auto& st = *existing;
```

检测过程：
经检测使用了 eosio_assert 进行验证。
检测结果：安全

3.6. 逻辑设计检测【安全】

检查代码中是否存在与业务设计相关的安全问题
检测过程：经检测该应用中不涉及该问题。
检测结果：安全

3.7. ＊变量初始化风险检测【安全】

参考如下：

1、对函数中的变量不先进行初始化，下次调用后，其中的局部变量还保留着上次执行的结果！！！（codeBlocks 版本，出现过此结果）***

2、计算结果不正确

3、程序逻辑与期望逻辑不一致

4、对于使用函数指针的程序，将导致进程崩溃。

5、写入硬盘的数据产生错误

6、严重的可能导致系统甚至硬件出现故障

检测过程：配置文件中明确了各配置变量初始值，且按规定进行调用
检测结果：安全
修复建议：无

3.8. apply 校验检测【安全】

在处理合约调用时，应确保每个 action 与 code 均满足关联要求。

```
// extend from EOSIO_ABI
#define EOSIO_ABI_EX( TYPE, MEMBERS )
extern "C" {
   void apply( uint64_t receiver, uint64_t code, uint64_t action ) {
      auto self = receiver;
      if( action == N(onerror)) {
         /* onerror is only valid if it is for the "eosio" code account and authorized by "eosio"'s "active permission */
         eosio_assert(code == N(eosio), "onerror action's are only valid from the \"eosio\" system account");
      }
      if( code == self || code == N(eosio.token) || action == N(onerror) ) {
         TYPE thiscontract( self );
         switch( action ) {
            EOSIO_API( TYPE, MEMBERS )
         }
         /* does not allow destructor of thiscontract to run: eosio_exit(0); */
      }
   }
}

EOSIO_ABI_EX(eosio::charity, (hi)(transfer))

```

检测过程：
该应用不涉及该 apply 调用

检测结果：安全

修复建议：无

3.9. transfer 假通知检测【安全】

在处理合约调用时，应确保每个 action 与 code 均满足关联要求。

在处理 require_recipient 触发的通知时，应确保 transfer.to 为_self。

```
652  void transfer(uint64_t sender, uint64_t receiver) {
653
654      auto transfer_data = unpack_action_data<st_transfer>();
655
656      if (transfer_data.from == _self || transfer_data.from == N(eosbetcasino)){
657          return;
658      }
659
660      eosio_assert( transfer_data.quantity.is_valid(), "Invalid asset");
661  }
```

检测过程：

合约中对不涉及 apply 外合约调用 action，故不存在该风险。

检测结果：安全

修复建议：无

3.10. 随机数检测【安全】

随机数生成算法不要引入可控或者可预测的种子

```
// source code: https://github.com/loveblockchain/eosdice/blob/3c6f9bac570cac236302e94b62432b73f6e74c3b/eos
uint8_t random(account_name name, uint64_t game_id)
{
    auto eos_token = eosio::token(N(eosio.token));
    asset pool_eos = eos_token.get_balance(_self, symbol_type(S(4, EOS)).name());
    asset ram_eos = eos_token.get_balance(N(eosio.ram), symbol_type(S(4, EOS)).name());
    asset betdiceadmin_eos = eos_token.get_balance(N(betdiceadmin), symbol_type(S(4, EOS)).name());
    asset newdexpocket_eos = eos_token.get_balance(N(newdexpocket), symbol_type(S(4, EOS)).name());
    asset chintailease_eos = eos_token.get_balance(N(chintailease), symbol_type(S(4, EOS)).name());
    asset eosbiggame44_eos = eos_token.get_balance(N(eosbiggame44), symbol_type(S(4, EOS)).name());
    asset total_eos = asset(0, EOS_SYMBOL);
    //攻击者可通过inline_action改变余额total_eos，从而控制结果
    total_eos = pool_eos + ram_eos + betdiceadmin_eos + newdexpocket_eos + chintailease_eos + eosbiggame44_
    auto mixd = tapos_block_prefix() * tapos_block_num() + name + game_id - current_time() + total_eos.amou
    const char *mixedChar = reinterpret_cast<const char *>(&mixd);

    checksum256 result;
    sha256((char *)mixedChar, sizeof(mixedChar), &result);

    uint64_t random_num = *(uint64_t *)(&result.hash[0]) + *(uint64_t *)(&result.hash[8]) + *(uint64_t *)(&
    return (uint8_t)(random_num % 100 + 1);
}
```

检测过程：该合约中未使用随机数。

检测结果：安全

修复建议：无

3.11. 回滚攻击检测【安全】

- 手法 1：在事务中探测执行结果(如收款金额、账号余额、表记录、随机数计算结果等)，当结果满足一定条件时调用 eosio_assert ，使得当前事务失败回滚。
- 手法 2：利用超级节点黑名单账号发起事务，欺骗普通节点做出响应，但此事务不会被打包。

如：

- 博弈类游戏下注随即开奖并转账，恶意合约可通过 inline_action 检测余额是否增加，从而回滚失败的开奖
- 博弈类游戏下注随即将开奖结果写入表内，恶意合约可通过 inline_action 检测表中记录，从而回滚失败的开奖
- 博弈类游戏开奖结果与游戏内奖券号相关联，恶意合约可通过同时发起多笔小额下注事务和一笔大额下注事务，当收到小额中奖时回滚事务，从而达到将可中奖的奖券号“转让”给大额下注的目的。
- 博弈类游戏开奖事务与下注事务没有关联，攻击者可用黑名单账号或者恶意合约回滚下注

检测过程：从源代码分析，该合约不涉及下注开奖，不涉及该检测

检测结果：安全

4. 附录 A：合约代码

本次测试代码来源：

#code

```
/**
 *  @file
 *  @copyright defined in eos/LICENSE.txt
 */

#include <eosio.token/eosio.token.hpp>

namespace eosio {

void token::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;
    });
}


void token::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" );

    stats statstable( _self, sym.code().raw() );
    auto existing = statstable.find( sym.code().raw() );
    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, 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, "active"_n} },
                          { st.issuer, to, quantity, memo }
      );
    }
}

void token::retire( 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" );

   stats statstable( _self, sym.code().raw() );
   auto existing = statstable.find( sym.code().raw() );
   eosio_assert( existing != statstable.end(), "token with symbol does not exist" );
   const auto& st = *existing;

   require_auth( st.issuer );
   eosio_assert( quantity.is_valid(), "invalid quantity" );
   eosio_assert( quantity.amount > 0, "must retire positive quantity" );

   eosio_assert( quantity.symbol == st.supply.symbol, "symbol precision mismatch" );

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

   sub_balance( st.issuer, quantity );
}

void token::transfer( name    from,
                  name    to,
                  asset   quantity,
                  string  memo )
{
   eosio_assert( from != to, "cannot transfer to self" );
   require_auth( from );
   eosio_assert( is_account( to ), "to account does not exist");
   auto sym = quantity.symbol.code();
   stats statstable( _self, sym.raw() );
   const auto& st = statstable.get( sym.raw() );

   require_recipient( from );
   require_recipient( to );

   eosio_assert( quantity.is_valid(), "invalid quantity" );
   eosio_assert( quantity.amount > 0, "must transfer positive quantity" );
   eosio_assert( quantity.symbol == st.supply.symbol, "symbol precision mismatch" );
   eosio_assert( memo.size() <= 256, "memo has more than 256 bytes" );

   auto payer = has_auth( to ) ? to : from;

   sub_balance( from, quantity );
   add_balance( to, quantity, payer );
}

void token::sub_balance( name owner, asset value ) {
  accounts from_acnts( _self, owner.value );

  const auto& from = from_acnts.get( value.symbol.code().raw(), "no balance object
found" );
  eosio_assert( from.balance.amount >= value.amount, "overdrawn balance" );

  from_acnts.modify( from, owner, [&]( auto& a ) {
       a.balance -= value;
     });
}

void token::add_balance( name owner, asset value, name ram_payer )
{
  accounts to_acnts( _self, owner.value );
  auto to = to_acnts.find( value.symbol.code().raw() );
  if( to == to_acnts.end() ) {
     to_acnts.emplace( ram_payer, [&]( auto& a ){
       a.balance = value;
     });
  } else {
     to_acnts.modify( to, same_payer, [&]( auto& a ) {
       a.balance += value;
     });
  }
}
```

```
void token::open( name owner, const symbol& symbol, name ram_payer )
{
   require_auth( ram_payer );

   auto sym_code_raw = symbol.code().raw();

   stats statstable( _self, sym_code_raw );
   const auto& st = statstable.get( sym_code_raw, "symbol does not exist" );
   eosio_assert( st.supply.symbol == symbol, "symbol precision mismatch" );

   accounts acnts( _self, owner.value );
   auto it = acnts.find( sym_code_raw );
   if( it == acnts.end() ) {
      acnts.emplace( ram_payer, [&]( auto& a ){
        a.balance = asset{0, symbol};
      });
   }
}

void token::close( name owner, const symbol& symbol )
{
   require_auth( owner );
   accounts acnts( _self, owner.value );
   auto it = acnts.find( symbol.code().raw() );
   eosio_assert( it != acnts.end(), "Balance row already deleted or never existed. Action
won't have any effect." );
   eosio_assert( it->balance.amount == 0, "Cannot close because the balance is not zero." );
   acnts.erase( it );
}

} /// namespace eosio

EOSIO_DISPATCH( eosio::token, (create)(issue)(transfer)(open)(close)(retire) )
}
```

5. 附录 B：漏洞风险评级标准

智能合约漏洞评级标准	
漏洞评级	**漏洞评级说明**
高危漏洞	能直接造成代币合约或用户资金损失的漏洞，如：能造成代币价值归零的数值溢出漏洞、能造成交易所损失代币的假充值漏洞、能造成合约账户损失 ETH 或代币的重入漏洞等； 能造成代币合约归属权丢失的漏洞，如：关键函数的访问控制缺陷、call 注入导致关键函数访问控制绕过等； 能造成代币合约无法正常工作的漏洞，如：因向恶意地址发送 ETH 导致的拒绝服务漏洞、因 gas 耗尽导致的拒绝服务漏洞。
中危漏洞	需要特定地址才能触发的高风险漏洞，如代币合约拥有者才能触发的数值溢出漏洞等；非关键函数的访问控制缺陷、不能造成直接资金损失的逻辑设计缺陷等。
低危漏洞	难以被触发的漏洞、触发之后危害有限的漏洞，如需要大量 ETH 或代币才能触发的数值溢出漏洞、触发数值溢出后攻击者无法直接获利的漏洞、通过指定高 gas 触发的事务顺序依赖风险等。

知道创宇

[咨询电话] +86(10)400 060 9587
[投诉电话] 13811527185
[邮　　箱] sec@knownsec.com
[网　　址] www.knownsec.com
[地　　址] 北京市朝阳区望京SOHO T3 A座15层

北京知道创宇信息技术有限公司