This document aims to provide some **security guidelines** for developers of EOS smart contracts and list analysis of some **known contract vulnerabilities**. We invite the community to suggest for modifications or improvements of this document and welcome various kinds of pull requests. You are also welcomed to add relevant articles or blogs published, please add them to reference.

Directories

Security Guidelines

EOS is still in its early stages and has some experimental characteristics. As a result, as new bugs and security vulnerabilities are discovered and new features are developed, the security threats we face are constantly changing. This article is just the start for developers to create secure smart contracts.

Developing smart contracts requires a new kind of engineering mindset, which is different from the development of our previous projects. Because the cost of making mistakes are so high, it's so hard to make up for it by patching, as centralized software does. As with hardware programming or software development for financial services, there are greater challenges to face than Web development or mobile development. Therefore, it is not enough to guard against the known vulnerabilities, but to learn new development concepts:

Be prepared for possible mistakes. Any meaningful smart contract is more or less wrong, so your code must be able to properly handle the bugs and vulnerabilities that arise. Always follow these rules:

stop the smart contract when an error occurs

manage the risk of the account, such as limit (transfer) rate and set maximum (transfer) limit

figure out effective ways to fix bugs and improve functionality

Be prudent of releasing smart contracts.Try your best to find out and fix the potential bugs before the smart contract is officially released.

- Test smart contracts thoroughly and retest them in time after any new attacks are discovered (testing those contracts that have been issued as well)
- Invite professional security audit firms for audition and provide Bug Bounty Program from the start of the alpha release on the cryptokylin-testnet, Jungle-testnet, or other public test nets.
- Release in several phases, at each phrase, ensure adequate testings are provided.

Keep the smart contracts simply。Increased complexity will increase the risk of error.

- Ensure the logic of the smart contracts are concise
- Ensure that contracts and functions are modular
- Use contracts or tools that are already widely adapted(For example, don't write a random number generator yourself)
- Clarity is more important than performance when allows
- Use blockchain tech only for the decentralized part of your system

Keep updated. Ensure access to the latest security developments by disclosing resources.

- Check your smart contract when any new vulnerabilities are discovered
- Update the library or tool as quickly as possible when possible
- Use the latest security technologies

Get clear understanding of blockchain features。Although your previous programming experience is also applicable to smart contract development, there are still pitfalls to keep an eye out for：

- `require_recipient(account_name name)` will trigger notification， and call the function with the same name within `name` contract(if account `name` already deployed contract)，see official doc here

Known Vulnerabilities

Numerical Overflow

When doing arithmetic operations, failing to check the boundaries may cause the values to overflow, causing loss of users assets.

Vulnerability Sample

codes with vulnerability：`batchtransfer` batch transfer

```
typedef struct acnts {

account_name name0;

account_name name1;

account_name name2;

account_name name3;

} account_names;

void batchtransfer(symbol_name symbol, account_name from, account_names to,
uint64_t balance)

{

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; //Multiplication overflow

int itr = db_find_i64(_self, symbol, N(table), from);

eosio_assert(itr >= 0, "Sub-- wrong name");

db_get_i64(itr, &fromaccount, (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);
}
```

Defense Method

As far as possible, use the `asset` structure for operations, rather than extract `balance` for operations.

The Real Case

- 【Don't play EOS Fomo3D Game】the Wolf game is under overflow attack and go die

Authorization Check

When making relevant operations, please do strictly determine whether the parameters passed into the function are consistent with the actual caller , use `require_auth` for authorization check.

Vulnerability Sample

codes with vulnerability：`transfer`

```
void token::transfer( account_name from,
```

```
    account_name to,
    asset        quantity,
    string       memo )
    {
    eosio_assert( from != to, "cannot transfer to self" );
    eosio_assert( is_account( to ), "to account does not exist");
    auto sym = quantity.symbol.name();
    stats statstable( _self, sym );
    const auto& st = statstable.get( sym );
    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 );
    }
```

Defense Method

Use `require_auth( from )` method to check whether the asset transfer account is consistent with the calling account

The Real Case

None

Apply Check

When processing contract calls, ensure that each action and codes meet the associated requirements.

Vulnerability Sample

codes with vulnerability：

```
// 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))
```

Defense Method

Use the codes below:

```
    if( ((code == self  && action != N(transfer) ) || (code == N(eosio.token)
&& action == N(transfer)) || action == N(onerror)) ) { }
```

Bind each key action and code to meet the requirements, in order to avoid abnormal and illegal calls.

The Real Case

EOSBet Transfer Hack Statement

Transfer Error Prompt

When processing a notification triggered by `require_recipient`, ensure that `transfer.to` is `_self`.

Vulnerability Sample

codes with vulnerability：

```
    // source code:
https://gitlab.com/EOSBetCasino/eosbetdice_public/blob/master/EOSBetDice.cp
p#L115void transfer(uint64_t sender, uint64_t receiver) {

    auto transfer_data = unpack_action_data<st_transfer>();
```

```
if (transfer_data.from == _self || transfer_data.from == N(eosbetcasino)){
return;
}
eosio_assert( transfer_data.quantity.is_valid(), "Invalid asset");
}
```

Defense Method

add

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

The Real Case

- EOS DApp recharge "error prompt" vulnerability analysis

Random Number Practice

Random number generator algorithm should not introduce controllable or predictable seeds

Vulnerability Sample

codes with vulnerability：

```
// source code: https://github.com/loveblockchain/eosdice/blob/3c6f9bac570cac236302e94b62432b73f6e74c3b/eosbocai2222.hpp#L174uint8_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);

    total_eos = pool_eos + ram_eos + betdiceadmin_eos + newdexpocket_eos +
chintailease_eos + eosbiggame44_eos;

    auto mixd = tapos_block_prefix() * tapos_block_num() + name + game_id -
current_time() + total_eos.amount;

    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 *)(&result.hash[16]) + *(uint64_t
*)(&result.hash[24]);

    return (uint8_t)(random_num % 100 + 1);

    }
```

Defense Method

True random numbers cannot be generated on the EOS. It is recommended to refer to the official example when designing a random class application.

- Randomization in Contracts

The Real Case

- SlowMist warning: Well-known DApp EOSDice is hacked again due to random number problems

Rollback Attack

- Technique 1: Detect execution results in the transaction (such as collection amount, account balance, table record, random number calculation result, etc.), and call `eosio_assert` when the result meets certain conditions, so that the current transaction fails to rollback.
- Technique 2: Initiate a transaction using the super-node blacklist account to trick the normal node to respond, but the transaction will not be packaged.

Vulnerability Sample

common modes with vulnerability:

- After the gambling game is bet, the draw will be opened and transferred. A malicious contract can detect if the balance is increased by `inline_action`, thus rolling back the failed lottery
- After the gambling game is bet, the result of the draw will be written into the form. A malicious contract can detect if the balance is increased by `inline_action`, thus rolling back the failed lottery
- The gambling game lottery results and in-game lottery number associated. A malicious contract can initiate a number of small bet transactions and a large bet transaction at the same time, and roll back the transaction when a small amount of winning is received, thereby achieving the purpose of "transferring" the prizeable lottery number to a large bet.
- The gambling game lottery is not associated with the betting transaction, an attacker can roll back a bet transaction with a blacklist account or a malicious contract

Defense Method

- Use `defer action` to transfer and send receipts
- Establish a dependency of reveal function, such as order dependency. And check if the record is existed on the blockchain when reveal. Even if the reveal function was excuted successful on the node

server, since the record is rolled back in bp, the corresponding record will be rolled back.

The Real Case

- Roll Back Attack about Blacklist in EOS