Smart Contract Security Audit Report

The SlowMist Security Team received the Paycoin team's application for smart contract security audit of the Paycoin token contract on Apri. 09, 2019. The following are the details and results of these smart contracts security audit:

Contract Name：

kiesnet-token

knt-pci

kiesnet-contract

kiesnet-id

Operation Platform：

Hyperledger Fabric

Initial Audit Version：

SHA256(chaincodes.tar)=

46725f44cdfec3eae93b491c84226809de144a9ba3074c00bf8d6f4897bfc167

Review Version：

kiesnet-contract commit: b002e434c85ff985f856f7b22052f1f16f08d2c3

kiesnet-id commit: 6d2801be20a236c33bf31bc81421609382455a6f

The audit items and results：

(Other unknown security vulnerabilities are not included in the audit responsibility scope)

No.	Audit Items	Audit Subclass	Audit Subclass Result
1	Overflow Audit	-	Passed
2	Authority Control Audit	Authority Vulnerability Audit	Passed
		Authority Excessive Audit	Passed
3	Safety Design Audit	Hard-coded Audit	Passed
		Show Coding Audit	Passed
		Abnormal Check Audit	Passed
		Type Safety Audit	Passed
		Non-Determinism Coding Audit	Passed
4	Design Logic Audit	-	Passed
5	Race Conditions Audit	-	Passed

6	"False-Deposit" Vulnerability Audit	-	Passed

Directory Structure

This project contains following contract files:

```
.
├── kiesnet-contract
│   ├── Gopkg.lock
│   ├── Gopkg.toml
│   ├── META-INF
│   ├── README.md
│   ├── contract.go
│   ├── contract_stub.go
│   ├── contract_tx.go
│   ├── errors.go
│   ├── main.go
│   ├── payload.go
│   ├── queries.go
│   ├── query_result.go
│   ├── sign.go
│   └── vendor
├── kiesnet-id
│   ├── Gopkg.lock
│   ├── Gopkg.toml
│   ├── META-INF
│   ├── README.md
│   ├── certificate.go
│   ├── collections.json
│   ├── errors.go
│   ├── identity.go
│   ├── identity_stub.go
│   ├── kid.go
│   ├── main.go
│   ├── payload.go
│   ├── pin.go
│   ├── queries.go
│   ├── query_result.go
│   └── vendor
```

```
├── kiesnet-token
│   ├── Gopkg.lock
│   ├── Gopkg.toml
│   ├── META-INF
│   ├── README.md
│   ├── account.go
│   ├── account_stub.go
│   ├── account_tx.go
│   ├── address.go
│   ├── amount.go
│   ├── balance.go
│   ├── balance_stub.go
│   ├── balance_tx.go
│   ├── contract_tx.go
│   ├── errors.go
│   ├── fee.go
│   ├── fee_stub.go
│   ├── fee_tx.go
│   ├── identifiable.go
│   ├── main.go
│   ├── pay.go
│   ├── pay_stub.go
│   ├── pay_tx.go
│   ├── queries.go
│   ├── query_result.go
│   ├── token.go
│   ├── token_stub.go
│   ├── token_tx.go
│   ├── transfer_tx.go
│   └── vendor
└── knt-pci
    ├── README.md
    ├── main.go
    └── token.go
```

Audit Result : **Passed**

Audit Number : 0X001904180003

Initial Audit Date : Apri. 18, 2019

Review Date : Nov. 06, 2019

Audit Team : SlowMist Security Team

account.go: Account Model

```
// Account _
type Account struct {
    DOCTYPEID     string       `json:"@account"` // address
    Token         string       `json:"token"`
    Type          AccountType  `json:"type"`
    CreatedTime   *txtime.Time `json:"created_time,omitempty"`
    UpdatedTime   *txtime.Time `json:"updated_time,omitempty"`
    SuspendedTime *txtime.Time `json:"suspended_time,omitempty"` //SlowMist// risk control, suspend an account for some time.
}
```

address.go: Address Model

```
// Address _
type Address struct {
    Code string
    Type AccountType
    Hash []byte
}
```

balance.go: Balance Model

```
// Balance _
type Balance struct {
```

```
    DOCTYPEID      string      `json:"@balance"` // address
    Amount         Amount      `json:"amount"`
    CreatedTime    *txtime.Time `json:"created_time,omitempty"`
    UpdatedTime    *txtime.Time `json:"updated_time,omitempty"`
    LastPrunedPayID string     `json:"last_pruned_pay_id,omitempty"`
}
```

pay.go: Payment Model

```
// Pay _
type Pay struct {
    DOCTYPEID   string      `json:"@pay"`                //address
    PayID       string      `json:"pay_id"`              //used for the external client to pass the pay id for refund
    Amount      Amount      `json:"amount"`              //can be positive(pay) or negative(refund)
    Fee         Amount      `json:"fee"`                 //can be positive(pay) or negative(refund, to return fee to merchant when she prune her pays)
    TotalRefund Amount      `json:"total_refund,omitempty"` //total refund value
    RID         string      `json:"rid"`                 //related id. user who pays to the merchant or receives refund from the merchant.
    ParentID    string      `json:"parent_id,omitempty"`    //parent id. this value exists only when the pay type is refund(negative amount)
    OrderID     string      `json:"order_id,omitempty"`     // order id. vendor specific unique identifier.
    Memo        string      `json:"memo"`
    CreatedTime *txtime.Time `json:"created_time,omitempty"`
}
```

token.go: Token Model

```
// Token _
type Token struct {
    DOCTYPEID      string      `json:"@token"` // Code,validate:"required,min=3,max=6,alphanum"
    Decimal        int         `json:"decimal"`
    MaxSupply      Amount      `json:"max_supply"`
    Supply         Amount      `json:"supply"`
    LastPrunedFeeID string     `json:"last_pruned_fee_id,omitempty"`
    GenesisAccount  string     `json:"genesis_account"`
    FeePolicy      *FeePolicy  `json:"fee_policy,omitempty"` // FeePolicy is nil if and only if knt fee is never yet imported. Once knt is initiated/upgraded with fee, it wil always exists.
    CreatedTime    *txtime.Time `json:"created_time,omitempty"`
```

```
    UpdatedTime     *txtime.Time `json:"updated_time,omitempty"`
}
```

fee.go: Transcation Fee model

```
// Fee is a transfer/pay fee utxo which will be pruned to genesis account
type Fee struct {
    DOCTYPEID   string      `json:"@fee,required"` // token code
    FeeID       string      `json:"fee_id"`       // unique sequential identifier
(timestamp + txid)
    Account     string      `json:"account"`      // account address who payed fee
    Amount      Amount      `json:"amount"`
    CreatedTime *txtime.Time `json:"created_time"`
}

// FeePolicy _
type FeePolicy struct {
    TargetAddress string           `json:"target_address"`
    Rates         map[string]FeeRate `json:"rates"`
}
```

transfer_tx.go: Transfer analysis

```
// params[0] : sender address (empty string = personal account)
// params[1] : receiver address
// params[2] : amount (big int string)
// params[3] : memo (see MemoMaxLength)
// params[4] : pending time (time represented by int64 seconds)
// params[5] : expiry (duration represented by int64 seconds, multi-sig only)
// params[6:] : extra signers (personal account addresses)
func transfer(stub shim.ChaincodeStubInterface, params []string) peer.Response {
    if len(params) < 3 {
        return shim.Error("incorrect number of parameters. expecting 3+")
    }

    // authentication

    kid, err := kid.GetID(stub, true) //SlowMist// check authority

    if err != nil {
        return shim.Error(err.Error())
    }

    // amount
    amount, err := NewAmount(params[2])
```

```
    if err != nil {
        return shim.Error(err.Error())
    }

    if amount.Sign() <= 0 { //SlowMist// transfer amount should <= 0
        return shim.Error("invalid amount. must be greater than 0")
    }

    // addresses
    rAddr, err := ParseAddress(params[1])
    if err != nil {
        logger.Debug(err.Error())
        return shim.Error("failed to parse the receiver's account address")
    }
    var sAddr *Address
    if len(params[0]) > 0 {
        sAddr, err = ParseAddress(params[0])
        if err != nil {
            logger.Debug(err.Error())
            return shim.Error("failed to parse the sender's account address")
        }

        if rAddr.Code != sAddr.Code { // not same token
            return shim.Error("different token accounts")
        }
    } else {
        sAddr = NewAddress(rAddr.Code, AccountTypePersonal, kid) //SlowMist// make a default
address
    }

    // IMPORTANT: assert(sender != receiver)
    if sAddr.Equal(rAddr) { //SlowMist// receiver should not be sender
        return shim.Error("can't transfer to self")
    }

    ab := NewAccountStub(stub, rAddr.Code)

    // sender
    sender, err := ab.GetAccount(sAddr)
    if err != nil {
```

```
        logger.Debug(err.Error())
        return shim.Error("failed to get the sender account")
    }

    if !sender.HasHolder(kid) { //SlowMist// check authority
        return shim.Error("invoker is not holder")
    }
    if sender.IsSuspended() {
        return shim.Error("the sender account is suspended")
    }

    // receiver
    receiver, err := ab.GetAccount(rAddr)
    if err != nil {
        logger.Debug(err.Error())
        return shim.Error("failed to get the receiver account")
    }

    if receiver.IsSuspended() { //SlowMist// check blacklist
        return shim.Error("the receiver account is suspended")
    }

    // sender balance
    bb := NewBalanceStub(stub)
    sBal, err := bb.GetBalance(sender.GetID())
    if err != nil {
        logger.Debug(err.Error())
        return shim.Error("failed to get the sender's balance")
    }

    fb := NewFeeStub(stub)
    fee, err := fb.CalcFee(sAddr, "transfer", *amount)
    if err != nil {
        logger.Debug(err.Error())
        return shim.Error("failed to get the fee amount")
    }

    // fee is not nil
    applied := amount.Copy().Add(fee)

    if sBal.Amount.Cmp(applied) < 0 { //SlowMist// verify balance
        return shim.Error("not enough balance")
```

```
    }

    // receiver balance
    rBal, err := bb.GetBalance(receiver.GetID())
    if err != nil {
        logger.Debug(err.Error())
        return shim.Error("failed to get the receiver's balance")
    }

    // options
    memo := ""
    var pendingTime *txtime.Time
    var expiry int64
    signers := stringset.New(kid)
    if a, ok := sender.(*JointAccount); ok {
        signers.AppendSet(a.Holders)
    }
    // memo
    if len(params) > 3 {
        if len(params[3]) > MemoMaxLength { // length limit
            memo = params[3][:MemoMaxLength] //SlowMist// max length of memo is 1024, it seems to long for memo.
        } else {
            memo = params[3]
        }
        // pending time
        if len(params) > 4 {
            seconds, err := strconv.ParseInt(params[4], 10, 64)
            if err != nil {
                return shim.Error("invalid pending time: need seconds since 1970")
            }
            ts, err := stub.GetTxTimestamp()
            if err != nil {
                return shim.Error("failed to get the timestamp")
            }
            if ts.GetSeconds() < seconds { // meaning pending time
                pendingTime = txtime.Unix(seconds, 0)
            }
            // expiry
```

```
            if len(params) > 5 && len(params[5]) > 0 {
                expiry, err = strconv.ParseInt(params[5], 10, 64)
                if err != nil {
                    return shim.Error("invalid expiry: need seconds")
                }
                // extra signers
                if len(params) > 6 {
                    addrs := stringset.New(params[6:]...) // remove duplication
                    for addr := range addrs.Map() {
                        kids, err := ab.GetSignableIDs(addr)
                        if err != nil {
                            return shim.Error(err.Error())
                        }
                        signers.AppendSlice(kids)
                    }
                }
            }
        }

        var log *BalanceLog // log for response

        if signers.Size() > 1 { // multi-sig
            if signers.Size() > 128 {
                return shim.Error("too many signers")
            }
            // pending balance id
            pbID := stub.GetTxID()
            // contract
            ptStr := "0"
            if pendingTime != nil {
                ptStr = params[4]
            }
            doc := []string{"transfer", pbID, sender.GetID(), receiver.GetID(), amount.String(),
fee.String(), memo, ptStr}
            docb, err := json.Marshal(doc)
            if err != nil {
                logger.Debug(err.Error())
                return shim.Error("failed to create a contract")
            }
            con, err := contract.CreateContract(stub, docb, expiry, signers)
            if err != nil {
```

```
            return shim.Error(err.Error())
        }
        // pending balance
        log, err = bb.Deposit(pbID, sBal, con, *amount, fee, memo)
        if err != nil {
            logger.Debug(err.Error())
            return shim.Error("failed to create the pending balance")
        }
    } else { // instant sending
        log, err = bb.Transfer(sBal, rBal, *amount, *fee, memo, pendingTime)
        if err != nil {
            logger.Debug(err.Error())
            return shim.Error("failed to transfer")
        }
    }

    // log is not nil
    data, err := json.Marshal(log)
    if err != nil {
        logger.Debug(err.Error())
        return shim.Error("failed to marshal the log")
    }

    return shim.Success(data)
}
```

Other Audit Issue:

- The use of concurrency is discouraged in chaincode.

no issues found

- The chaincode object should not declare any fields.

no issues found

- Operations on the ledger should not depend on global variables.

no issues found

- The usage of certain libraries can lead to non-determinism.

no issues found

- **Range iterations over map entries is not deterministic.**

Many “range map” found，but they are fine.

account_stub.go

```
        // create account-holder relationship
        for kid := range holders.Map() {
            holder := NewHolder(kid, account)
            holder.CreatedTime = ts
            if err = ab.PutHolder(holder); err != nil {
                return nil, nil, errors.Wrap(err, "failed to create the holder")
            }
        }
```

account_tx.go

```
    // validate & get kid of co-holders
    for addr := range addrs.Map() {
        kids, err := ab.GetSignableIDs(addr)
        if err != nil {
            return responseError(err, "invalid co-holder")
        }
        holders.AppendSlice(kids)
    }
```

token_tx.go

```
        for addr := range addrs.Map() {
            kids, err := ab.GetSignableIDs(addr)
            if err != nil {
                return responseError(err, "invalid co-holder")
            }
            holders.AppendSlice(kids)
        }
```

transfer_tx.go

```
                    for addr := range addrs.Map() {
                        kids, err := ab.GetSignableIDs(addr)
                        if err != nil {
                            return shim.Error(err.Error())
                        }
```

```
                    signers.AppendSlice(kids)
                }
```

contract.go

```
    for signer := range signers.Map() {
        args = append(args, []byte(signer))
    }
```

- The number of arguments should be validated before their use.

no issues found

- Read after write operations to the same variable yield the old value.

no issues found

- Errors should not be ignored.

no issues found

- Results of phantom reads should not be used to manipulate the ledger.

no issues found

- Exchange Issue Summary

1、User CA certificate can be revoked, which is an inherent property of Fabric system and has been publicized.

2、The total amount of PCI is 39,4100,0000.0000,0000，without token lock logic.

3、Fabric Chaincode can be update。

4、Transfer function is used by the user for non-payment transfer business (or withdrawal of the exchange), and the handling fee is borne by the remitter. Pay function is used when the payment is actually made, and the handling fee is borne by the payee. Exchange needs to judge the recharge method to avoid the loss of handling fee.

Official Website
www.slowmist.com

E-mail
team@slowmist.com

Twitter
@SlowMist_Team

WeChat Official Account