MYKEY（Tron）

Smart Contract Security Audit

2020-08-17

Abstract

This report provides a comprehensive view of the security audit results for the smart contract code of the MYKEY(Tron) project. The task of SlowMist is to review and point out security issues in the audited smart contract code.

Disclaimer

Summary

In this report, SlowMist audited the smart contract code for the MYKEY(Tron) project. The audit results showed that no issues with critical severity or high severity were found. However, some issues with medium severity and low severity were discovered. After communication and feedback from both parties, the issues have been fixed.

Project Overview

Description

File name：mykey-tron-slowmist-review-20200817.zip

SHA256: c35709a807ef4b4e92e598505c5eeec4ab91995dc411385667a4b38aa1c5cad3

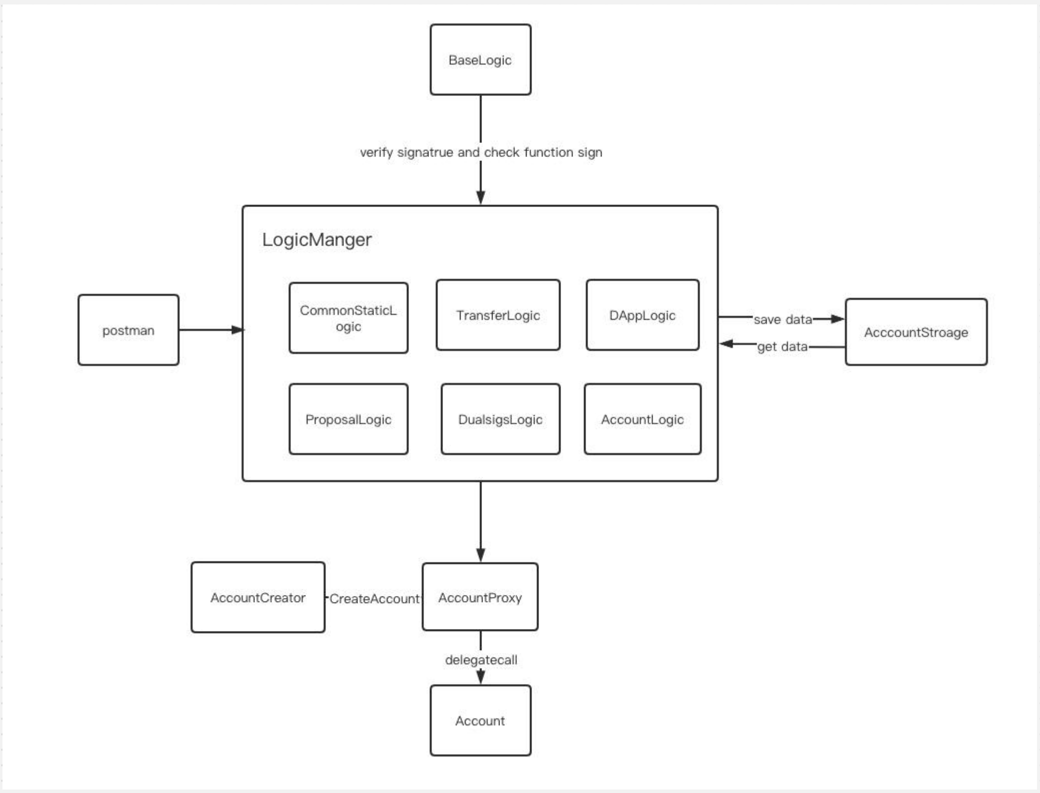
Project Structure

The project includes the following smart contract files:

```
./contracts
    contracts
    ├── Account.sol
    ├── AccountCreator.sol
    ├── AccountProxy.sol
    ├── AccountStorage.sol
    ├── LogicManager.sol
    ├── Migrations.sol
    ├── logics
    │   ├── AccountLogic.sol
    │   ├── CommonStaticLogic.sol
    │   ├── DappLogic.sol
    │   ├── DualsigsLogic.sol
    │   ├── ProposalLogic.sol
    │   ├── TransferLogic.sol
    │   └── base
    │       ├── AccountBaseLogic.sol
    │       └── BaseLogic.sol
    ├── testUtils
    │   ├── MyNft.sol
    │   └── MyToken.sol
    └── utils
        ├── MultiOwned.sol
        ├── Owned.sol
        ├── RLPReader.sol
    └── SafeMath.sol
```

Contracts Structure

MYKEY adopts the strategy of creating a contract for each account. AccountCreator contract is responsible for the account creation, and each account in MYKEY system is a contract. The account contract only stores a few data and loads the contract logic by delegatecall Account contract to reduce the user deployment cost. The account data is stored in AccountStorage contract. The MYKEY system has six modules now, which are TransferLogic, DAppLogic, DualsigsLogic, AccountLogic, CommonStaticLogic, and ProposalLogic and used for the transfer, interacting with DApp, multi-sign and account information modification. All these modules are managed by the LogicManger contract. Every ETH address (postman) who has the account signature can initiate an operation for the account. The overall structure of the contract is shown below:



Audit Methodology

The security audit process for smart contracts consists of the following two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using public and in-house automated analysis tools.
- Manually audit the security of the code. Discover the potential security issues in the code by manually analyzing the contract code.

The following is a list of common vulnerabilities that will be highlighted during the contract code audit:

- Reentrancy attack and other Race Conditions
- Replay attack
- Reordering attack
- Data Storage issue
- Short address attack
- Denial of service attack
- Transaction Ordering Dependence attack
- Conditional Completion attack
- Authority Control attack
- Integer Overflow and Underflow attack
- TimeStamp Dependence attack
- Gas Usage, Gas Limit and Loops
- Redundant fallback function
- Unsafe type Inference
- Explicit visibility of functions state variables
- Business Logic Flaws
- Uninitialized Storage Pointers
- Floating Points and Numerical Precision
- tx.origin Authentication
- "False top-up" Vulnerability
- Event Security

- Compiler version issues
- Call function Security

Audit Result

Critical Vulnerabilities

Critical severity issues can have a major impact on the security of smart contracts, and it is highly recommended to fix critical severity vulnerability.

The audit has shown no critical severity vulnerability.

High Risk Vulnerabilities

High severity issues can affect the normal operation of smart contracts, and it is highly recommended to fix high severity vulnerability.

The audit has shown no high severity vulnerability.

Medium Risk Vulnerabilities

Medium severity issues can affect the operation of a smart contract, and it is recommended to fix medium severity vulnerability.

1. Gas malicious consumption

The TransferLogic and the DAppLogic will finally call the fallback function of the AccountProxy contract, and eventually invoke the external contract through assembly, but in this flow, the contract does not limit the gas and the attacker can deploy a malicious contract to consume and steal the gas of the account and the asset of the project side.

Location: AccountProxy.sol line 19

```
function() external payable {

        if(msg.data.length == 0 && msg.value > 0) {
            emit Received(msg.value, msg.sender, msg.data);
        }
        else {
            // solium-disable-next-line security/no-inline-assembly
            assembly {
                let target := sload(0)
                calldatacopy(0, 0, calldatasize())
                let result := delegatecall(gas, target, 0, calldatasize(), 0, 0)
                returndatacopy(0, 0, returndatasize())
                switch result
                case 0 {revert(0, returndatasize())}
                default {return (0, returndatasize())}
            }
        }
    }
}
```

Fix status: Will limit and monitor the gas on the server-side.

Low Risk Vulnerabilities

Low severity issues can affect smart contracts operation in future versions of code. We recommend the project party to evaluate and consider whether these problems need to be fixed.

1. No restrictions on the number of user public key pairs and backup accounts

When using the Account contract to initialize an account, there is no limit on the number of pass-in user public key pairs and backup accounts. Users can upload any number of public keys, resulting in some functions may not work properly. Such as the function to get the public key of the signature by getKeyIndex.

Location: AccountLogic.sol line 308

```
function getKeyIndex(bytes memory _data) internal pure returns (uint256) {
                uint256 index; //index default value is 0, admin key
                bytes4 methodId = getMethodId(_data);
                if (methodId == ADD_OPERATION_KEY) {
                        index = 2; //adding key
                } else if (methodId == PROPOSE_AS_BACKUP || methodId == APPROVE_PROPOSAL) {
                        index = 4; //assist key
                }
                return index;
        }
```

Fix status: Control the number of public keys on the server-side. When creating the user's private key, at least 5 private keys will be passed in at one time.

2. Expired delay operation can be canceled

In Account Logic, when the cancelDealy function is called, the delay time is not judged, so that the delayed operation can still be canceled.

```
function cancelDelay(address payable _account, bytes4 _actionId) external allowSelfCallsOnly {
                accountStorage.clearDelayData(_account, _actionId);
                emit CancelDelay(_account, _actionId, keyNonce[accountStorage.getKeyData(_account, 0)]);
        }
```

Fix status: After confirming with the project party, cancelDelay method does not check the expiration time. (reason: if the delayed operation cannot be canceled after the delay time expires, the MYKEY backend may not guarantee multi-chain consistency in some critical cases, such as cancel on the EOS chain changeAdminKey succeeded, and the ETH chain did not cancel in time because the block was inserted slowly).

3. No consistency check on OperationKeys

The changeAllOperationKeys function does not check whether the keys are the same, it is recommended to add a check.

Location: Account.sol line 115

```
function changeAllOperationKeys(address payable _account, address[] calldata _pks) external allowSelfCallsOnly {
                    uint256 keyCount = accountStorage.getOperationKeyCount(_account);
                    require(_pks.length == keyCount, "invalid number of keys");
                    require(accountStorage.getDelayDataHash(_account, CHANGE_ALL_OPERATION_KEYS) == 0,
"delay data already exists");
                    address pk;
                    for (uint256 i = 0; i < keyCount; i++) {
                              pk = _pks[i];
                              require(pk != address(0), "0x0 is invalid");
                    }
                    bytes32 hash = keccak256(abi.encodePacked('changeAllOperationKeys', _account, _pks));
                    uint256 due = now + DELAY_CHANGE_OPERATION_KEY;
                    accountStorage.setDelayData(_account, CHANGE_ALL_OPERATION_KEYS, hash, due);
                    emit ChangeAllOperationKeys(_account, _pks, keyNonce[accountStorage.getKeyData(_account, 0)],
due);
          }
```

Fix status: After communicating with the project party, it is determined that the contract layer does not take the repeated keys checking. In theory, the client will not generate duplicate keys, and the server will check.

4. No minimum restriction number of Backupkeys

There is no minimum restriction number of BackupKeys, causing the Backup account to do evil without the user's knowledge, change the public key of the main account, and steal user assets. In the case of a single backup account, you can directly initiate a proposal, bypass approvedProposal, and then call executeProposal. After waiting 30 days, you can change the user's master public key (Adminkey).

Location: AccountLogic.sol line 260, ProposalLogic.sol line 90

Attack flow:

(1) The backup account initiate the proposal.

```
function proposeAsBackup(address _backup, address payable _client, bytes calldata _functionData) external
allowSelfCallsOnly {
                        require(getSignerAddress(_functionData) == _client, "invalid _client");

                        bytes4 proposedActionId = getMethodId(_functionData);
                        require(proposedActionId == CHANGE_ADMIN_KEY_BY_BACKUP, "invalid proposal by backup");
                        checkRelation(_client, _backup);
                        bytes32 functionHash = keccak256(_functionData);
                        accountStorage.setProposalData(_client, _backup, proposedActionId, functionHash, _backup);
                        emit ProposeAsBackup(_client, _backup, _functionData,
keyNonce[accountStorage.getKeyData(_backup, 4)]);
            }
```

(2) In the case of BackupKey, the backup account directly initiates the executeProposal operation. After 30 days, the system automatically triggers and changes the account's admin public key.

```
function changeAdminKeyByBackup(address payable _account, address _pkNew) external allowSelfCallsOnly {
                        require(_pkNew != address(0), "0x0 is invalid");
                        address pk = accountStorage.getKeyData(_account, 0);
                        require(pk != _pkNew, "identical admin key exists");
                        require(accountStorage.getDelayDataHash(_account, CHANGE_ADMIN_KEY_BY_BACKUP) == 0,
"delay data already exists");
                        bytes32 hash = keccak256(abi.encodePacked('changeAdminKeyByBackup', _account, _pkNew));
                        uint256 due = now + DELAY_CHANGE_ADMIN_KEY_BY_BACKUP;
                        accountStorage.setDelayData(_account, CHANGE_ADMIN_KEY_BY_BACKUP, hash, due);
                        emit ChangeAdminKeyByBackup(_account, _pkNew, due);
            }

    // called from external
            function triggerChangeAdminKeyByBackup(address payable _account, address _pkNew) external {
                        bytes32 hash = keccak256(abi.encodePacked('changeAdminKeyByBackup', _account, _pkNew));
                        require(hash == accountStorage.getDelayDataHash(_account,
CHANGE_ADMIN_KEY_BY_BACKUP), "delay hash unmatch");

                        uint256 due = accountStorage.getDelayDataDueTime(_account,
CHANGE_ADMIN_KEY_BY_BACKUP);
                        require(due > 0, "delay data not found");
                        require(due <= now, "too early to trigger changeAdminKeyByBackup");
                        accountStorage.setKeyData(_account, 0, _pkNew);
                        //clear any existing related delay data and proposal
                        accountStorage.clearDelayData(_account, CHANGE_ADMIN_KEY_BY_BACKUP);
                        accountStorage.clearDelayData(_account, CHANGE_ADMIN_KEY);
```

```
                    clearRelatedProposalAfterAdminKeyChanged(_account);
                    emit ChangeAdminKeyByBackupTriggered(_account, _pkNew);
          }
```

Fix status: After confirming with the project party, the first default backup account is currently MYKEY official, and any account related operations will be notified to the user.