Smart Contract Security Audit Report

Hydro MultiSigWallet

SECBIT

Dec 20, 2018

1. Introduction

Hydro MultiSigWallet is a multi-signature wallet contract deployed on Ethereum. SECBIT Labs conducted an audit from Nov 16th, 2018 to Nov 19th, 2018, including an analysis of the contract from 3 aspects: **implementation bugs**, **logic flaws** and **risk assessment**. The audit shows that the Hydro MultiSigWallet contract has no critical security issue. Several issues were found in the first round of audit as shown in the following list (check Section 4 for details). SECBIT Labs provided suggestions for fixes and improvements. After the developing team optimized and updated the code, SECBIT Labs re-audited this project from Dec 11th, 2018 to Dec 14th, 2018. The latest version has addressed all found issues and introduced no additional issue, which is ready for release.

Type	Description	Level	Status
Code Optimization	Condition check `count == required` in `isConfirmed()` may halt transaction under certain circumstances	Low	Fixed
Implementation Bug	`getTransactionIds()` lacks of range checks for `from` and `to`, which may be underflow/out-of-bound of the array `_transactionIds` and cause abnormal results	Medium	Fixed
Code Optimization	Modifier `ownerExists` is missed in `executeTransaction()`, which is inconsistent with the contract logic	Low	Fixed

2. Contract Information

This section describes basic contract information and code structure.

2.1 Basic Information

The following list shows the basic information of MultiSigWallet:

Name	**MultiSigWallet**
Lines of Code	394 & 64
File Name	MultiSigWallet.sol, MultiSigWalletWithLock.sol
Source	Hydro
Initial Git Commit	ab0d97dd01683cfadc1e02681c08e6946ef838ac
Final Git Commit	fabed26fdfe250bcb2a4cc0af1cb6b4bfb0098f0
Stage	In development

2.2 Contract List

The following content shows the contracts included in the MultiSigWallet project:

Name	**Lines**	**Description**
MultiSigWallet	394	Multisignature wallet - Allows multiple parties to agree on transactions before execution
MultiSigWalletWithLock	64	Multisignature wallet with lock function

3. Contract Analysis

This section analyzes the MultiSigWallet contract from three aspects: the type of the contract, the account permissions in the contract, and the functionality of the contract.

3.1 Contract Type

Wallet contract.

3.2 Account Permissions

There are 2 types of accounts in MultiSigWallet: common account and owner.

Common Account

- Description

  Wallet observer
- Permissions
  - Deposit ETH into the wallet
  - View the information of wallet administrators, wallet transaction contents and status
- Authentication Methods

  None. Everyone can be the common account.

Owner

- Description

  Wallet administrator
- Permissions
  - All permissions of the common account
  - Submit transaction proposals
  - Vote for and execute transactions
  - Ownership transfer
- Authentication Methods
  - Authorized on the contract creation
  - Authorized via transaction proposals
  - Authorized by the ownership transfer

3.3 Functionality

The primary functionality of MultiSigWallet is providing a multi-signature wallet managed by multiple administrators.

Stage 1: Contract creation and preparation

- On the contract creation, set the owners, and the required number of owners to confirm wallet transactions.
- Owners deposit assets to the wallet, which will be managed by multiple owners together.

Stage 2: Transaction proposal submission

- Any owner can submit transaction proposals, confirm, and wait for confirmations from other owners.

Stage 3: Proposal execution

- For every transaction, after the required number of confirmations are collected, the transaction becomes ready and is locked by a time lock. After the time lock expires, the transaction can be executed.

4. Audit Details

This section introduces the process and the complete content of the audit, and explains the discovered issues and risks in detail.

4.1 Audit Process

The audit strictly followed the audit specification of SECBIT Lab. We analyzed the project from code bug, logical implementation and potential risks. The process consists of four steps:

1. Each audit team reviews the wallet application independently.
2. Each audit team evaluates the vulnerabilities and potential risks independently.
3. Each audit team shares its audit results, and reviews results from other teams.
4. All audit teams coordinate with the audit leader to prepare the final audit report.

In the audit, several security analyzers and scanners were used, including

- SECBIT Labs internal tools: Adelaide checker, SF-checker, badmsg.sender
- 3rd open-source tools: Mythril, Slither, SmartCheck, Securify

The analyzing and scanning results were reviewed manually by audit teams. Besides, audit teams inspected and reviewed the contract line by line.

4.2 Audit Result

A complete list of 20 audit items and the corresponding results are listed below.

Number	Classification	Result
1	All functions work correctly in normal cases	✓
2	No obvious bug (*e.g.* unhandled integer overflow/underflow)	✓
3	Can be compiled without warnings and errors	✓
4	Pass security analyzers and scanners without issues	✓
5	Compliant with security development standards	✓

6	No risk in low level calls (`call`, `delegatecall`, `callcode`) and inline assembly	✓
7	No deprecated or outdated usage	✓
8	Clear implementation; explicit function visibility, variable types and Solidity version number	✓
9	No redundant code	✓
10	No potential risk from the manipulation of timestamp and network environment	✓
11	No issue in calling external contracts (*e.g.*, token contracts)	✓
12	No high GAS consuming operations in contract	✓
13	Clear and explicit business logic	✓
14	Implementation consistent with comments, whitepaper, *etc*.	✓
15	No logic not mentioned in design	✓
16	No ambiguity in logic and implementation	✓
17	No obvious flaw in mechanism design	✓
18	No fairness issue in business logic *wst*. game theory	✓
19	No risk threatening the project team	✓
20	No risk threatening normal users	✓

4.3 Issues

1. Condition check `count == required` in `isConfirmed()` may halt transaction under certain circumstances
    - Severity: **Low**
    - Type: Code Optimization

- Description:

  `executeTransaction()` is called to execute the transaction after `confirmTransaction()` is called by administrators. If the number of confirmations from different administrators reaches the required value, the transaction will be executed via `call`. However, if the balance of contract is insufficient, `call` will fail. Administrators needs to call `revokeConfirmation()` in order to continue the transaction.

- Impact:

  Transactions in wallet might not get executed normally.

- Suggestion:

  Change the transaction execution condition in `isConfirmed()` to `count >= required`.

- Status:

  Fixed.

2. `getTransactionIds()` lacks of range check of `from` and `to`, which may be underflow/out-of-bound of the array `_transactionIds` and cause abnormal results

   - Severity: **Medium**
   - Type: Implementation Bug
   - Description:

     The function does not check `from` and `to`, which might contain integer underflow bug.

```
  _transactionIds = new uint256[](to - from);
  for (i = from; i < to; i++) {
      _transactionIds[i - from] = transactionIdsTemp[i];
  }
```

   - Impact:

     Unable to query results normally.

   - Suggestion:

     Add checks to verify that `from` & `to` are within the transaction number range and `to` is greater than `from`.

```
  require(from<to && to<=transactionIdsTemp.length, "variable out
of range.");
```

- Status:

  Deleted `getTransactionIds()`.

3. Modifier `ownerExists` is missed in `executeTransaction()`, which is inconsistent with the contract logic
   - Severity: **Low**
   - Type: Code Optimization
   - Description:

     `executeTransaction()` executes transaction proposals, but it can be called by anyone, which is inconsistent with the contract logic.
   - Impact:

     Any user can execute transaction proposals that satisfy all other conditions.
   - Suggestion:

     Use `ownerExists` modifier in `executeTransaction()`.
   - Status:

     Fixed.

4.4 Potential Risks

SECBIT Labs found the following risk after assessing MultiSigWallet contract.

1. The `call` operation in `executeTransaction()` can call arbitrary functions, which may introduce risks
   - Severity: **Mid**
   - Type: Logical Implementation
   - Description:

     `executeTransaction()` uses `call` to execute transactions which are from inputs. The wallet owners should check the content of transactions to avoid potential issues.

```
    function executeTransaction(uint256 transactionId)
        public
        notExecuted(transactionId)
    {
        if (isConfirmed(transactionId)) {
            Transaction storage transaction =
transactions[transactionId];
```

```
            transaction.executed = true;
            if
(transaction.destination.call.value(transaction.value)
(transaction.data))
                emit Execution(transactionId);
            else {
                emit ExecutionFailure(transactionId);
                transaction.executed = false;
            }
        }
    }
```

5. Conclusion

The MultiSigWallet contract implements a multi-signature wallet. In the audit, SECBIT Labs found 1 implementation bug, 2 possible code optimizations and 1 potential risk. The corresponding fixes and improvements were also provided by SECBIT Labs. The latest version of MultiSigWallet has addressed all above issues. Overall, SECBIT Labs think the MultiSigWallet contract is in a high quality.

Disclaimer

The smart contract audit service from SECBIT Labs assesses the contract's correctness, security and performability in code quality, logic design and potential risks. The report is provided "as is", without any warranties about the code practicability, business model, management system's applicability and anything related to the contract adaptation. This audit report is not to be taken as an endorsement of the platform, team, company or investment.

APPENDIX

Vulnerability/Risk Level Classification

Severity	Description
High	Severely damage the contract's integrity and allow attackers to steal ethers and tokens, or lock ethers inside the contract.
Medium	Damage contract's security under given conditions and cause impairment of benefit for stakeholders.
Low	Cause no actual impairment to contract.
Info	Relevant to practice or rationality of the contract, could possibly bring risks.

SECBIT Labs is devoted to construct a common-consensus, reliable and ordered blockchain economic entity.

 http://www.secbit.io

 audit@secbit.io

 @secbit_io