Multisender

audit report

14 August 2019

BlockchainLabs.NZ

Table of contents

Background

This audit report was undertaken by **BlockchainLabs.nz** for the purpose of providing feedback to the **MultiSender** team. It has subsequently been shared publicly without any express or implied warranty.

Solidity contracts were provided by the **Multisender** development team at this commit[_multisender_contracts_v2 @ 3a407830d7cde93192d19c27b4411d1fec36edb9] - we would encourage all community members and token holders to make their own assessment of the contracts.

Document structure

The report will include the following sections:

- Security audit
- Test suite analysis
- Observations
- Conclusion

Focus areas

Correctness	No correctness defects uncovered during static analysis? No implemented contract violations uncovered during execution? No other generic incorrect behaviour detected during execution? Adherence to adopted standards such as ERC20?
Testability	Test coverage across all functions and events? Test cases for both expected behaviour and failure modes? Settings for easy testing of a range of parameters? No reliance on nested callback functions or console logs? Avoidance of test scenarios calling other test scenarios?
Security	No presence of known security weaknesses? No funds at risk of malicious attempts to withdraw/transfer? No funds at risk of control fraud? Prevention of Integer Overflow or Underflow?
Best Practice	Explicit labeling for the visibility of functions and state variables? Proper management of gas limits and nested execution? The latest version of the Solidity compiler?

Scope

All smart contracts, test files, migration and deployment scripts from this Github repo:
https://github.com/Multisender-app/_multisender_contracts_v2

Configurational files, documents and other assets **were out** of the scope of this audit.

Primary smart Contracts in scope

- multisender/Claimable.sol
- multisender/Ownable.sol
- multisender/Messages.sol
- multisender/UpgradebleStormSender.sol

Framework contracts

- EternalStorage.sol
- EternalStorageProxyForStormMultisender.sol
- OwnedUpgradeabilityProxy.sol
- OwnedUpgradeabilityStorage.sol
- Proxy.sol
- SafeMath.sol
- UpgradeabilityOwnerStorage.sol
- UpgradeabilityProxy.sol
- UpgradeabilityStorage.sol

Issues found

Severity description

Minor	A defect that does not have a material impact on the contract execution and is likely to be subjective.
Moderate	A defect that could impact the desired outcome of the contract execution in a specific scenario.
Major	A defect that impacts the desired outcome of the contract execution or introduces a weakness that may be exploited.
Critical	A defect that presents a significant security vulnerability or failure of the contract across a range of scenarios.

Minor

Avoid magic numbers (Ether address) 🏷️ *Best practice*

```
329  emit Multisended(_total, 0x000000000000000000000000000000000000bEEF);
...
360  emit Multisended(_total, 0x000000000000000000000000000000000000bEEF);
```

Suggest replace the address 0x000000000000000000000000000000000000bEEF with a constant variable. It improves readability and maintainability.

- ❎ No fix needed

change should be subtracted from the sent amount 🏷️ *Correctness*

```
114  if (change != 0) {
115    erc20token.transfer(msg.sender, change);
116  }
117  emit Multisended(_total, _token);
```

When emitting the event, the real token amount that been sent (_total.sub(change)) should be logged..

- Not fixed

Duplicated code can be replaced by modifier 🏷️ *Best practice*

```
176  require(_contributors.length > 0, "no contributors sent");
177  require(_contributors.length == _balances.length, "different arrays
lengths");
...
228  require(_contributors.length > 0, "no contributors sent");
229  require(_contributors.length == _balances.length, "different arrays
lengths");
```

Line 176-177 and line 228-229 can be replaced by modifier validLists()

- Not fixed

Local variable created and assigned a value, but never used 🏷️ *Best practice*

```
181     (bool success, ) = _token.call(
182         abi.encodeWithSelector(
183             erc20token.transferFrom.selector,
184             tokenHolder, _contributors[i], _balances[i]
185         )
186     );
```

Boolean success is created and assigned a value, but never used.

- Not fixed

Event should log customer's address 🏷️ *Correctness*

```
367  function setAddressToVip(address _address, uint256 _tier) external
onlyOwner {
368     setUnlimAccess(_address, _tier);
369     emit PurchaseVIP(msg.sender, _tier);
370  }
```

At line 369, it should log customer's address, not the msg.sender. Because msg.sender, in this case, is always the owner of the contract.

- Not fixed

Don't trust outside contract 🏷️ *Correctness*

```
94  function multisendToken(
95      address _token,
96      address[] calldata _contributors,
97      uint256[] calldata _balances,
98      uint256 _total
99  ) external validLists(_contributors.length, _balances.length) hasFee
payable {
100     uint256 change = 0;
101     ERC20 erc20token = ERC20(_token);
102     erc20token.transferFrom(msg.sender, address(this), _total);
103     for (uint256 i = 0; i < _contributors.length; i++) {
104         (bool success, ) = _token.call(
105             abi.encodeWithSelector(
106                 erc20token.transfer.selector,
107                 _contributors[i], _balances[i]
108              )
109         );
110         if(!success) {
111             change += _balances[i];
112         }
113      }
114      if (change != 0) {
115          erc20token.transfer(msg.sender, change);
116      }
117      emit Multisended(_total, _token);
118  }
```

When our contract is calling functions in outside contracts, we need to verify if the expected behaviours are really being executed.
At line 102, it's calling `erc20token.transferFrom()`. We expect the token contract to transfer the tokens to the multisender contract address by calling this function. But a dummy token contract can be designed not to do the transaction.
It should fail and revert when it does not have tokens to send to the `_contributors`. So it would not cause any loss in this case.
Suggest checking the multisender contract's token balance after this `transferFrom()` for confirming receiving the right amount tokens.

Moderate

- None found

Major

- None found

Critical

- None found

Test Suite Analysis

Overview

The test suite is is written using the Truffle testing framework and makes use of the truffle-assert library. It is nicely organised and includes a set of helper functions to handle common testing patterns.

The test runner is a bash script which ensures that the test environment uses the same batch of user addresses with the same balances which means there will be a consistent testing environment. This script is the same one used by OpenZeppelin (test.sh)

Generally speaking, the tests are well written and we are happy with the overall test coverage.

Unexpected Input Testing

Over half of the tests in the test suite test error conditions and make sure that the functions fail when they are expected to. This is a sign of a healthy test framework and we were pleased to see the focus on error handling.

Suggestions for improvement

Creating a perfect test suite can be a bit of an art, you need to find a balance between following every best practice, and the amount of time you can realistically spend writing tests. For smart contracts the test suite should generally be above average compared to other projects. We believe the following criteria describe a healthy test suite.
Automation is an invaluable tool to be paired with a test suite, any changes that want to enter the code base should only be allowed after all tests have passed. Using a tool such as Travis CI which can integrate with Github can help here.
Tests should be **repeatable**, every time you run a test suite it should produce the same results. Using a test script like this project is a good example of a consistent test environment.
Readability is another important aspect of a test suite, the code should be just as professional and readable as the main codebase. In general these tests were well written and easy to understand.

Observations

This section contains notes about issues that likely do not require any action to be addressed, but are not so trivial that we won't mention them.

_total should be sum of _balances

```
94  function multisendToken(
95     address _token,
96     address[] calldata _contributors,
97     uint256[] calldata _balances,
98     uint256 _total
99  ) external validLists(_contributors.length, _balances.length) hasFee
payable {
```

_total at line 98 is the sum of _balances. It can be a local variable calculated in the function. If _total is **less** than the sum of _balances, the transaction should fail at line 104 and increase the change at line 111, then revert at line 115 due to lack of tokens to return. Look below:

```
114   if (change != 0) {
115         erc20token.transfer(msg.sender, change);
116   }
```

If _total is **greater** than the sum of _balances, there will be some token left after sending, and the sender cannot claim them by himself.
However, this scenario only happens when a user wants to call this contract directly without using the multisender website. The reason we put this issue in the observation section is that this contract works with the front end at most of the time.

Prefer .transfer over .send

```
    function _checkFee(address _user, address payable _referral)
internal {
        uint contractFee = currentFee(_user);
        if (contractFee > 0) {
            require(msg.value >= contractFee, "no fee");
            if(_referral != address(0)) {
                _referral.send(referralFee());
            }
        }
    }
```

<address>.transfer is preferred over <address>.send because transfer will throw an exception when a transfer fails, whereas send will just return false. Of course if you want to allow a transfer to be able to fail without making the entire function then this is a valid decision. But it's worth noting for transparencies sake.

Unfinished code

The smart contract is still under development when we are auditing it. We can see an unfinished function in it alongside with newly added features. So far the unfinished function does not affect the safety of these contracts.

Conclusion

The Multisender team demonstrated a strong understanding of Solidity and smart contracts. Overall we were very happy with the responsiveness of their development team and were impressed by their adherence to best practices for smart contract development.

We are satisfied that a user's Tokens or ETH aren't at risk of being hacked or stolen through interacting with these contracts provided that all user input is correct and the contracts are used as they are meant to be. We suggested some changes regarding best practices and but in general we noted that the potential attack surface was very low already and the developers clearly had this in mind.

Overall we consider the resulting contracts following the audit feedback period to be robust and have not identified any critical vulnerabilities. Solidity best practices have been followed and the developers have taken care to provide adequate documentation in the smart contracts.

Disclaimer

Our team uses our current understanding of the best practises for Solidity and Smart Contracts. Development in Solidity and for Blockchain is an emergering area of software engineering which still has a lot of room to grow, hence our current understanding of best practise may not find all of the issues in this code and design.

We have not analysed any of the assembly code generated by the Solidity compiler. We have not verified the deployment process and configurations of the contracts. We have only analysed the code outlined in the scope. We have not verified any of the claims made by any of the organisations behind this code.

Security audits do not warrant bug-free code. We encourage all users interacting with smart contract code to continue to analyse and inform themselves of any risks before interacting with any smart contracts.