Smart Contract Source Code Audit sol-mass-payouts

Prepared for Bloq • January 2021

v210121

1. Table Of Contents

2. Executive Summary

In **September 2020**, Bloq engaged Coinspect to perform a source code review of the smart contracts in the git repository at https://github.com/bloq/sol-mass-payouts. The objective of the project was to evaluate the security of the smart contracts and write tests.

The assessment was conducted on the contracts from the `master` branch of the git repository at https://github.com/bloq/sol-mass-payouts as of commit `cdab5ed9ca18227bd5cd47eb785115c83aebf14d` of **September 25th, 2020**.

The following issues were identified during the assessment:

High Risk	Medium Risk	Low Risk	Zero Risk
1	1	3	-

BMP-004 is due to a confusion with how the `transferFrom` mechanism works, and it's considered high severity because **the funds end up locked in the contract and lost**.

As for the medium risk finding BMP-003, it presents a **DoS attack that prevents the creation of any claims group**.

And among the low severity problems, BMP-002 is about **calls to `require` without specifying a reason string**, BMP-001 is an integer overflow when adding funds, and BMP-005 is about an incorrect return value when a claims group doesn't exist.

After the first assessment results were presented to Bloq, all issues were properly addressed following Coinspect recommendations. Then Coinspect reviewed the proposed changes, wrote more tests and suggested more improvements as well as fixes for some newly introduced issues. As of commit `682fcb87923deb15d7c0b0af00dbd8712050a065` of **January 11th, 2021**, **all issues have been fixed and no further problems have been found**.

Regarding the use of the `MerkleBox` contract, it is worth mentioning that, unless the creator of the Merkle tree publishes the full list of recipients and amounts that were used to build the Merkle tree, there are no guarantees that recipients with a valid receipt will be able to claim their payment, not only because the creator could have backdoored the Merkle tree in order to be able to make arbitrary withdrawals at any time bypassing the time lock, but also because there's no way to know that a claims group is not underfunded. So, when a claims group is created, and if it is desired to provide guarantees, **the full list of recipients and amounts used for creating the Merkle tree should be made available off-chain for independent verification**.

3. Introduction

The audit started on October 19th and was conducted on the `master` branch of the git repository at https://github.com/bloq/sol-mass-payouts as of commit `cdab5ed9ca18227bd5cd47eb785115c83aebf14d` of **September 25th, 2020**.

The scope of the audit was limited to the following Solidity source files, shown here with their sha256sum hash:

```
7e5a9212eb2163ce99c040c3a18c0dc3946405cc0372ede69e966d778360988b  MerkleBox.sol
b0045ed87fb560f0877530e5a9424b8fe76b9148d115fd5f04c37ad957cb2522  Migrations.sol
7e0a0e2874d7422cfab8beacebed36536accf9170d215c559de9967a458059a8  MultiTransfer.sol
2dd044466d2caacc03ce0f9a3d1a346e712d90f63c78b0092e41287a408bcc6a  interfaces/IMerkleBox.sol
e4aed9026b5d47ccfcfff93510e486b3c6c64fa9df73ec0161bca954eefa73cd  interfaces/IMultiTransfer.sol
```

The final verification of fixes was performed on the source code as of commit `682fcb87923deb15d7c0b0af00dbd8712050a065` of **January 11th, 2021,** corresponding to the following Solidity source files and sha256sum hashes:

```
512c1c1430ecd496cb1071541fb2f8bdb9f3c378e20e462003e190820becc9a3  MerkleBox.sol
b0045ed87fb560f0877530e5a9424b8fe76b9148d115fd5f04c37ad957cb2522  Migrations.sol
2b6fdd2b21e0d098429b9ee75d1b5228ce303d8ea8fa7984fc89604292f1d085  MultiTransfer.sol
e2897ce1dd98888c636c15f8f0e9939744fcbb48a8e14999465531d27710cb1b  interfaces/IERC20WithPermit.sol
e79ba5665329ce0cc33c5db984dbfe1f541a0b9731f46dc48ca48b7717884937  interfaces/IMerkleBox.sol
2a1e30175aa4309e7a8383e1344a3ff5e0aa2a53eb566dd7cfca86a39f5d7a02  interfaces/IMultiTransfer.sol
e4f193b22d91f7af1ee632d11b91a5545221dc9d7a716c9c0f12c1910d727690  mocks/ERC20Mock.sol
760ad9259c3e0afef8774a9564ca60b106c0c0074b5e399930d73b9013588dd0  mocks/ERC20WithPermitMock.sol
```

4. Assessment

The contracts are specified to be compiled with Solidity version 0.6.6. In general it is recommended to keep up to date with the latest Solidity version (for example 0.6.12 or 0.7.2).

The contracts compile without warnings. However, linting with `'npm run solhint'` produces 186 errors and 112 warnings. It is recommended to address all the linting problems.

Originally the repository didn't contain any tests, and extensive tests were developed during the assessment.

The `MultiTransfer` contract implements only the function `multiTransfer` and anyone can call this function to make batch ERC20 transfers to a list of pairs `(receiver, amount)`. This function was found to not work as expected because of a confusion with the use of `safeTransferFrom` (see BMP-004).

The `MerkleBox` contract implements a pull payment mechanism with Merkle trees. A *funder* can build (off-chain) a Merkle tree with leaves `(receiver, amount)`, then approve the required total amount of tokens for the `MerkleBox` and call the function `newClaimsGroup` to transfer the tokens to the `MerkleBox` and register the Merkle root (the root of the Merkle tree just built) as a claims group. Then, for each `(receiver, amount)` pair, any address can call the function `claim` in `MerkleBox` and the corresponding amount of tokens will be transferred to the receiver.

The `MerkleBox` contract also provides function `addFunds` that allows any address to deposit additional funds in a claims group via `approve/transferFrom`, and `addFundsWithPermit` that uses the ERC2612 *permit* extension to ERC20 with ERC712-signed approvals.

An integer overflow was found in function `addFunds` (see BMP-001).

Also, the functions `withdrawFunds` and `claim` were not working properly and would revert. The problem was that they were calling `safeTransferFrom` instead of `safeTransfer` (see BMP-004).

It is important to mention that, unless the list of receivers and amounts used for creating a Merkle tree is made public, there are no guarantees that recipients with a valid receipt will be able to claim their payment, not only because the funder could withdraw the tokens at any time, but also because there's no way to know that a claims group hasn't been underfunded without reconstructing the Merkle tree from the full list of `(receiver, amount)` pairs that was used to build the tree. Having the full list of `(receiver, amount)` pairs available allows anyone to verify that the deposited amount is at least as much as the sum of all the amounts, and from this it can be concluded that the claims group is not underfunded or backdoored, and if time-locking is enabled (i.e., the funder cannot immediately withdraw funds) all receivers will be able to claim and receive their tokens.

5. Summary of Findings

ID	Description	Risk	Fixed
BMP-001	Integer overflow in `addFunds`	Low	✔
BMP-002	Require with no reason string	Low	✔
BMP-003	DoS in `newClaimsGroup`	Medium	✔
BMP-004	Incorrect use of `transferFrom` mechanism	High	✔
BMP-005	`claimable()` returns true to unknown claims group	Low	✔

6. Findings

BMP-001 Integer overflow in addFunds

Total Risk	Impact	Location
Low	High	MerkleBox.sol
Fixed	**Likelihood**	
✔	Low	

Description

The function `addFunds` in the `MerkleBox` contract can produce an integer overflow when performing the addition of the current balance with the newly deposited funds:

```
function addFunds(bytes32 merkleRoot, uint256 amount) external override {
    // prelim. parameter checks
    require(amount != 0, "Invalid amount");
[...]
    // calculate amount to deposit.  handle deposit-all.
    IERC20 token = IERC20(holding.erc20);
    uint256 balance = token.balanceOf(msg.sender);
    if (amount == uint256(-1)) {
        amount = balance;
    }
    require(amount <= balance, "Insufficient balance");
    require(amount != 0, "Amount cannot be zero");
[...]
    // update holdings record
    holding.balance += amount;
```

Recommendation

Use the SafeMath library, or make sure to detect the integer overflow and revert.

BMP-002 Require with no reason string

Total Risk	Impact	Location
Low	Low	MultiTransfer.sol
Fixed	Likelihood	
✔	High	

Description

Calls to `require` in `MultiTransfer` don't specify reason strings. Although this doesn't affect security or functionality, it is generally considered a bad practice.

Recommendation

Following best practices, it is recommended to always specify a reason string in all calls to `require`. This allows tests to verify that a `revert` happens because of the expected reason.

BMP-003 DoS in newClaimsGroup

Total Risk	Impact	Location
Medium	High	MerkleBox.sol
Fixed ✔	Likelihood Low	

Description

The funder creates a new claims group for an ERC20 token and a set of pairs `(recipient, amount)` by building a Merkle tree with the pairs `(recipient, amount)` as leafs, then approving the desired amount of tokens to the address of the `MerkleBox` instance, and calling the function `newClaimsGroup` with the root of the Merkle tree as argument (among other arguments). The root of the Merkle tree is used as a key to identify the claims group in a mapping, and duplicates are not allowed: if the provided Merkle root has already been used for a claims group, it reverts.

An attacker could look for transactions to a `MerkleBox` and front run calls to `newClaimsGroup`. The attacker would simply send a call to `newClaimsGroup` with a minimal amount (or a dummy ERC20 token of himself) and the same Merkle root that the funder submitted. After this, when the funder's call is finally executed, the function `newClaimsGroup` reverts with *"`Holding already exists`"* because the Merkle root is already registered.

Recommendation

There are several ways to address this problem. One way is to make the `MerkleBox` contract `Ownable`, and add to `newClaimsGroup` the modifier `onlyOwner`. Another way is to add a sequential id to each claim group (an integer increasing with each new claim group, for example) and identify claim groups by id instead of the Merkle root.

BMP-004 Incorrect use of transferFrom mechanism

Total Risk	Impact	Location
High	High	MerkleBox.sol, MultiTransfer.sol
Fixed ✔	Likelihood: High	

Description

The functions `withdrawFunds` and `claim` in the `MerkleBox` contract fail with a revert, and any deposited funds are lost. This is because these functions call `safeTransferFrom` but should call `safeTransfer` instead.

And in the `MultiTransfer` contract, the function `multiTransfer` first transfers the full amount of tokens to itself and then attempts to make each individual transfer from itself to the receiver (and this fails because no allowance is made before calling `safeTransferFrom` the second time):

```
// receive total amount
IERC20 token = IERC20(erc20);
token.safeTransferFrom(msg.sender, address(this), amountIn);

uint256 totalOut = 0;

// output amount to recipients
for (uint i = 0; i < bits.length; i++) {
    address a = address(bits[i] >> 96);
    uint amount = bits[i] & ((1 << 96) - 1);
    token.safeTransferFrom(address(this), a, amount);

    totalOut += amount;
}
```

Recommendation

Replace the calls to `safeTransferFrom` in functions `withdrawFunds` and `claim` with `safeTransfer`.

In the case of `MultiTransfer`, a better approach that uses less gas is to avoid the first transfer altogether, and do this instead:

```
IERC20 token = IERC20(erc20);

uint256 totalOut = 0;

// output amount to recipients
for (uint i = 0; i < bits.length; i++) {
    address a = address(bits[i] >> 96);
    uint amount = bits[i] & ((1 << 96) - 1);
```

```
        token.safeTransferFrom(msg.sender, a, amount);

        totalOut += amount;
    }
```

BMP-005 claimable() returns true to unknown claims group

Total Risk	Impact	Location
Low	Low	MerkleBox.sol
Fixed	Likelihood	
✔	High	

Description

The functions `claimable` is intended to return `true` when the sender can use a given Merkle proof to claim a given amount of tokens from the claim group identified by a given Merkle tree root:

```
function claimable(bytes32 merkleRoot, uint256 amount, bytes32[] memory proof)
external override view returns (bool) {
    bytes32 leaf = _leafHash(amount);
    if (leafClaimed[merkleRoot][leaf] == true) {
        return false;
    }
    return MerkleProof.verify(proof, merkleRoot, leaf);
}
```

However, as long as the proof is valid and it hasn't yet been claimed, the function claimable always returns true, regardless whether the provided `merkleRoot` has been registered and funded by calling `newClaimGroup` (`claim` reverts with *"Holding not found"*), and regardless whether the available funds are sufficient (`claim` reverts with *"Claim under-funded by funder."*).

Recommendation

Make sure that the behaviour of the function claimable is consistent and clearly specified in the documentation. It is suggested to change `claimable` to return `false` for unknown claim groups, and to make clear in the documentation that a claim could still fail if it is underfunded.

7. Disclaimer

The present security audit does not cover the endpoint systems and wallets that communicate with the contracts, nor the general operational security of the company whose contracts have been audited. This document should not be read as investment advice or an offering of tokens.