WalletSplitter

Smart Contract Audit Report

Apr 8, 2022

By Andrey L

1. Document Revisions

0.1	Initial Report	Apr 8, 2022

2. Overview

This document presents my findings of security issues and code practice in the reviewed contracts.

2.1 Executive Summary

Game 7 Labs requested to conduct an audit of smart contract “WalletSplitter”.
The audit took place on Apr 7, 2022.

2.2 Audit Methodology

Contract is audited with the following methods.

a. Manual code review
   The contract code is reviewed line by line for common vulnerabilities, best practices and architectures.
b. Local deployment, unit testing and attacking
   The contract code is tested by unit tests, and attacked by mock contracts and scripts, using the Hardhat framework.

2.3 Disclaimer

I’ve put my best effort to find all vulnerabilities in the contract, however my findings shouldn’t be considered as a complete list of all existing issues. The statements made in this document should not be interpreted as investment or legal advice, nor should its author be held accountable for decisions made based on them.

3. System or Protocol Overview

It includes only one smart contract, “WalletSplitter”. The “WalletSplitter” is a smart contract that splits and withdraws the deposited ETH into 4 owners. These 4 owners will be appointed initially while deploying the contract into Blockchain, and each owner can appoint a new owner for future withdrawals.

4. Finding classification

Each finding contains an *Impact* and *Likelihood* ratings.
The full definitions are as follows:

Impact

- High
  Code that activates the issue will lead to undefined or catastrophic consequences for the system.
- Medium
  Code that activates the issue will result in consequences of serious substance.
- Low
  Code that activates the issue will have outcomes on the system that are either recoverable or don’t jeopardize its regular functioning.
- Warning

The issue cannot be exploited given the current code and/or configuration (such as deployment scripts, compiler configuration, use of multisignature wallets for owners, etc.), but could be a security vulnerability if these were to change slightly. If we haven't found a way to exploit the issue given the time constraints, it might be marked as "Warning" or higher, based on my best estimate of whether it is currently exploitable.

- Informational
  The issue is on the border-line between code quality and security. Examples include insufficient logging for critical operations. Another example is that the issue would be security-related if code or configuration (see above) was to change.

Likelihood

- High
  The issue is exploitable by virtually anyone under virtually any circumstance.
- Medium
  Exploiting the issue currently requires non-trivial preconditions.
- Low
  Exploiting the issue requires strict preconditions.

5. Findings

This section contains the list of discovered findings.

Summary of Findings

	Type or Category	Impact	Likelihood
ETH can be locked forever in the wallet	DoS with (Unexpected) revert	High	High
ERC20 can be locked forever in the wallet	DoS with (Unexpected) revert	High	High
ETH funds can be lost forever into black hole, null address	Data Validation	Medium	High
Inclusion of unnecessary library, SafeMath in solidity 0.8	Optimization	Informational	N/A
Practice of function modifiers, public / external	Optimization	Informational	N/A
All address variables are defined as payable	Coding Convention	Informational	N/A

5.1 Problem: DoS with (Unexpected) revert. ETH can be locked forever in the wallet.

If a contract address that has no *fallback()* or *receive()*, or reverts always is inputted as one of the 4 owners, this will lead to a denial of service, and nobody can receive ETH from the wallet, and it will be locked forever.
In withdraw() function, it is designed to split and withdraw all ETH into 4 owners at once. So all transfers should be executed one by one linearly.
In Solidity language, *.transfer()* is trying to call *fallback()* or *receive()* with limited gas for a contract account, and if there is no *fallback()* or *receive()*, then it will revert. And also malicious contracts can intentionally revert in the *fallback()* or *receive()*. One failure of four transfer() with this issue will bring the revert of the entire transaction, and ETH can not be withdrawn.
It is highly possible by selecting the contract address as one of the 4 owners, while deploying the contract or by the execution of *updateOwner()*.

Solution: There is not any validation for the owner addresses, and I recommend adding strict validation for the owners in the *constructor()*, and *updateOwner()*.
This will be mentioned again later in section 5.3.

5.2 Problem: DoS with (Unexpected) revert. ERC20 can be locked forever in the wallet
If one of the 4 owners has a null address, then this will lead to a denial of service, and nobody can receive the token from the wallet, and it will be locked forever.
In the withdrawToken() function, it is designed to split and withdraw all balances into 4 owners at once. So all transfers should be executed one by one linearly. In ERC20, transfer() and transferFrom() reverts if the source or destination is a null address (0x000…0). One failure of 4 transfer() with this issue will bring the revert of the entire transaction, and the token can not be withdrawn.
It is highly possible by selecting the null address as one of the 4 owners, while deploying the contract or by the execution of *updateOwner()*.

Solution: There is not any validation for the owner addresses, and I recommend adding strict validation for the owners in the *constructor()*, and *updateOwner()*.
This will be mentioned again later in section 5.3.

5.3 Problem: Data Validation. ETH funds can be lost forever into black hole, null address
There is not any validation for the owner addresses, so if one of them is a null address, then ETH will be transferred, and lost into black hole.

Solution:

```
////////////////////////////////////////////////////////

// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

import "@openzeppelin/contracts/utils/math/SafeMath.sol";
import "@openzeppelin/contracts/security/ReentrancyGuard.sol";
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/utils/Address.sol";
```

```
contract WalletSplitter is ReentrancyGuard {
    constructor(
      address payable _owner0,
      address payable _owner1,
      address payable _owner2,
      address payable _owner3
    ) {
        require(_owner0 != address(0) && !Address.isContract(_owner0), "Not allowed contract");
        require(_owner1 != address(1) && !Address.isContract(_owner1), "Not allowed contract");
        require(_owner2 != address(2) && !Address.isContract(_owner2), "Not allowed contract");
        require(_owner3 != address(3) && !Address.isContract(_owner3), "Not allowed contract");

        owner0 = _owner0;
        owner1 = _owner1;
        owner2 = _owner2;
        owner3 = _owner3;
    }

/* ... */

function updateOwner(address payable _newOwner) public returns(bool) {
        require(_newOwner != address(0) && !Address.isContract(_newOwner), "Not allowed contract");

        if (msg.sender == owner0) {
            owner0 = _newOwner;
            return true;
        }
        if (msg.sender == owner1) {
            owner1 = _newOwner;
            return true;
        }
        if (msg.sender == owner2) {
            owner2 = _newOwner;
            return true;
        }
        if (msg.sender == owner3) {
            owner3 = _newOwner;
            return true;
        }
        return false;
    }
}
```

5.4 Optimization. Inclusion of unnecessary library, SafeMath in Solidity v0.8

In the contract, it used SafeMath library for arithmetic operations, but since Solidity v0.8.0, arithmetic operations revert on underflow and overflow as default.
Here is a note from openzeppelin, regarding the SafeMath library.

```
// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts v4.4.1 (utils/math/SafeMath.sol)

pragma solidity ^0.8.0;

…

/**
```

```
 * @dev Wrappers over Solidity's arithmetic operations.
 *
 * NOTE: `SafeMath` is generally not needed starting with Solidity 0.8, since the compiler
 * now has built in overflow checking.
 */
library SafeMath {
....
```

Recommendation: SafeMath can be removed from the contract.

5.5 Optimization. Practice of function modifiers, public / external
In Solidity language, *external* functions are part of the contract interface, which means they can be called from other contracts and via transactions, and *public* functions are part of the contract interface and can be either called internally or via messages.
In a nutshell, *public* and *external* differs in terms of gas usage. The former use more than the latter when used with large arrays of data. This is due to the fact that Solidity copies arguments to memory on a public function while external read from calldata which is cheaper than memory allocation.
For best practices, we should use *external* if we expect that the function will only ever be called externally, and use *public* if we need to call the function internally.

Recommendation: In the contract, *withdraw()*, *withdrawToken()* and *updateOwner()* are only external interfaces, and will never be called internally, so these functions can be refactored into external.

5.6 Coding Convention. All address variables are defined as payable
The *address* and *address payable* types both store a 160-bit Ethereum address. The concept of payable and non-payable addresses only exists in the Solidity type system at compile-time. The difference between payable and non-payable addresses is gone in the compiled contract code.
.transfer() and *.send()* can be used on *address payable*, but not on *address*, and *.call()* can be used on both *address* and *address payable*. The difference is that the compiler (at compile time) when encountering an *address payable* is ready to allow that address to access primitives useful to manage ethers (namely call, transfer, send).
The splitting of addresses into *address* and *address payable* mainly serves for the purpose to limit and warn programmers incorrect usage of transferring ETH.
In the contract, it used all addresses as *address payable*.

Recommendation: all owner addresses can be defined as *address* and can be casted into *address payable* only when to transfer ETH in withdraw().

```
...
address private owner0;
address private owner1;
address private owner2;
address private owner3;
```

```
constructor(address _owner0, address _owner1, address _owner2, address _owner3) {
        require(_owner0 != address(0) && !Address.isContract(_owner0), "Not allowed contract");
        require(_owner1 != address(1) && !Address.isContract(_owner1), "Not allowed contract");
        require(_owner2 != address(2) && !Address.isContract(_owner2), "Not allowed contract");
        require(_owner3 != address(3) && !Address.isContract(_owner3), "Not allowed contract");

        owner0 = _owner0;
        owner1 = _owner1;
        owner2 = _owner2;
        owner3 = _owner3;
    }

function withdraw() external payable nonReentrant {
    ...
    payable(owner0).transfer(twoFifth + dust);
    payable(owner1).transfer(oneFifth);
    payable(owner2).transfer(oneFifth);
    payable(owner3).transfer(oneFifth);
}
```

6. Appendix: Final Contract Code

Based on the findings, I've added a new code base here.

```
//////////////////////////////////////////////////////

// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

import "@openzeppelin/contracts/security/ReentrancyGuard.sol";
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/utils/Address.sol";

contract WalletSplitter is ReentrancyGuard {
    address private owner0; // gets 2/5 of fees + dust
    address private owner1; // gets 1/5 of fees
    address private owner2; // gets 1/5 of fees
    address private owner3; // gets 1/5 of fees

    constructor(address _owner0, address _owner1, address _owner2, address _owner3) {
        require(_owner0 != address(0) && !Address.isContract(_owner0), "Not allowed contract");
        require(_owner1 != address(0) && !Address.isContract(_owner1), "Not allowed contract");
        require(_owner2 != address(0) && !Address.isContract(_owner2), "Not allowed contract");
        require(_owner3 != address(0) && !Address.isContract(_owner3), "Not allowed contract");

        owner0 = _owner0;
        owner1 = _owner1;
        owner2 = _owner2;
        owner3 = _owner3;
    }

    // to receive native token
    receive() external payable {}

    function withdraw() external payable nonReentrant {
        // get balance of native token
        uint256 balance = address(this).balance;
        // calculate fees
        uint256 oneFifth = balance / 5;
        uint256 twoFifth = balance / 5 * 2;
        uint256 dust     = balance % 5;
        // owner0 gets 2/5 of fees + dust
        payable(owner0).transfer(twoFifth + dust);
        // owner1,2,3 gets 1/5 fees
        payable(owner1).transfer(oneFifth);
        payable(owner2).transfer(oneFifth);
        payable(owner3).transfer(oneFifth);
    }

    function withdrawToken(address _contract) external nonReentrant{
        IERC20 erc20 = IERC20(_contract);
        // get balance of tokens
        uint256 balance = erc20.balanceOf(address(this));
        // calculate fees
        uint256 oneFifth = balance / 5;
        uint256 twoFifth = balance / 5 * 2;
        uint256 dust     = balance % 5;
        // owner0 gets 2/5 of fees + dust
        erc20.transfer(owner0, (twoFifth + dust));
        // owner1,2,3 gets 1/5 fees
        erc20.transfer(owner1, oneFifth);
        erc20.transfer(owner2, oneFifth);
        erc20.transfer(owner3, oneFifth);
    }

    function updateOwner(address  _newOwner) public returns(bool) {
        require(_newOwner != address(0), "Not allowed contract");

        if (msg.sender == owner0) {
            owner0 = _newOwner;
            return true;
        }
        if (msg.sender == owner1) {
            owner1 = _newOwner;
            return true;
        }
        if (msg.sender == owner2) {
            owner2 = _newOwner;
            return true;
        }
        if (msg.sender == owner3) {
            owner3 = _newOwner;
            return true;
        }
        return false;
    }

}


/////////////////////////////////////////////////////
```

Thank You

Andrey L

rockid332@gmail.com