Social Recovery Wallet Implementation for the Ethereum Blockchain

Anshuman Misra
amisra7@uic.edu

CS 594: Foundations of Blockchains

Bibliography

[1] Why we need wide adoption of social recovery wallets
Vitalik Buterin
https://vitalik.ca/general/2021/01/11/recovery.html

[2] Ethereum Whitepaper
Vitalik Buterin
A next-generation smart contract and decentralized application platform. White paper 3, no. 37 (2014).

[3] Solidity Documentation
Ethereum Foundation
https://docs.soliditylang.org/en/v0.8.10/

[4] Brownie Documentation
Brownie Project
https://github.com/eth-brownie/brownie

Outline

Introduction

- Wallet security is one of the biggest challenges in cryptocurrency adoption for the end user
- If a user loses their private key, all funds associated with the key become unreachable
- If an attacker steals the user's private key, they can access all of the funds in the user's account

Introduction

- Current solutions addressing wallet security are mnemonics and hardware wallets
- However, these solutions are highly vulnerable
- Social recovery wallets are a solution to the lost private key problem
- Multisig wallets are a solution for the stolen private key problem

Wallet Design

- Point of Failure
- Mental Overhead
- Ease of Use

Social Recovery Wallet

- Smart contract that holds funds for a user
- Only the user that owns the smart contract can access funds
- Smart contract contains a list of guardian accounts
- A majority of guardian accounts can change ownership of the smart contract

Social Recovery Wallet

- The owner cannot add/remove guardian accounts instantly
- Social recovery wallets in conjunction with a vault can solve both the stolen and lost private key problems

Guardians

- Self
- Friends/Family
- Institution

Guardians

- Avoid collusion and attacks by publishing a hash of guardian addresses
- Single purpose addresses for guardians
- Diverse collection of guardians

Technology Stack

- web3.py/Brownie
- Solidity
- Ganache

Decentralized Applications (DApps)

- DApps consist of a backend executed by a decentralized network of nodes
- The frontend can be hosted on a centralized service or on a decentralized service
- Smart contracts are the backend programs that execute on the blockchain
- The frontend executes functions contained in smart contracts

Decentralized Applications (DApps)

- Smart contracts consist of functions and state stored on the blockchain
- Smart contract code cannot be changed
- Smart contacts run as programmed
- A smart contract is a type of ethereum account

Decentralized Applications (DApps)

- Advantages include transparency, security and censorship resistance
- Concerns include cost and scalability

Social Security Wallet Features

The user client software provides the following features to the wallet owner:

- Deploy wallet
- Receiving funds
- Transferring funds
- Change Guardians

Social Security Wallet Features

The guardian client software provides the following features:

- Request for change of ownership
- Agree to request for change of ownership
- Change the ownership of the wallet

System Model

- The system consists of n guardian processes
- Shared memory provided by the blockchain
- No message passing
- Asynchronous communication
- t crash failures, $t < \lceil n/2 \rceil$
- Byzantine behaviour among the guardians is not allowed

The Ownership Change Problem

- An initiator process posts a request for ownership change in shared memory
- At least $\lceil n/2 \rceil$ processes need to agree to the request
- Any process can execute change of ownership after fulfilment of agreement requirement

The Ownership Change Problem

- The initiator (or any other process) asynchronously checks whether a majority of processes have agreed to its request
- Assume processes that do not reply as crash failures

Implementation - Ownership Change

```
10  pragma solidity ^0.8.0;
11
12  contract wallet{
13
14      address private owner; // owner of account
15      uint256 private balance; // Balance of Wallet
16      address[] private guardians;  // set of guardian acounts
17      uint256[] private agreement_array; // vector used to arrive at consensus between guardians for changing the ownership of the wallet
18      uint256 private counter;  // keeps track of the number of times a change of ownership has been requested
19
20      event ChangeRequest(uint256 s_no); // Event used to communicate between guardians
21      address[] private change_list; // Each time the ownership needs to be changed, a new address is pushed into this list
```

Figure 1: State Variables

Implementation - Ownership Change

```
45      function ChangeOwnerRequest(address new_owner) public payable returns(bool){
46
47          bool flag = false;
48
49          for(uint256 i = 0; i < guardians.length; i++){
50
51              if(guardians[i] == msg.sender){
52
53                  flag = true;
54              }
55          }
56
57          if(flag == false){
58
59                  return false;
60          }
61
62
63          counter = counter + 1;
64
65          change_list.push(new_owner);
66
67          agreechangeowner();
68
69          emit ChangeRequest(counter);
70
71          return true;
72
73
74      }
```

Figure 2: Request for Ownership Change

Implementation - Ownership Change

```
77      function agreechangeowner() public payable returns(bool){
78
79          if(counter < 1){
80
81              return false;
82          }
83
84          for(uint256 i = 0; i < guardians.length; i++)
85          {
86              if(guardians[i] ==  msg.sender){
87
88                  if (counter > agreement_array[i]){
89
90                      agreement_array[i] = agreement_array[i] + 1;
91
92                  }
93
94
95
96              }
97          }
98          return true;
99
100     }
```

Figure 3: Agreement

Implementation - Ownership Change

```
103     function changeOwner() public payable returns(bool){
104         bool flag1 = false;
105         bool flag2 = true;
106         uint256 consensus_count = 0;
107         // Check if the request is from a guardian account
108          for(uint256 i = 0; i < guardians.length; i++)
109         {
110             if(guardians[i] ==  msg.sender){
111                 flag1 = true;
112                 break;
113             }
114         }
115         // Check if the guardians have arrived at consensus
116           for(uint256 i = 0; i < guardians.length; i++)
117         {
118             if(agreement_array[i] ==  counter){
119                 consensus_count = consensus_count + 1;
120             }
121         }
122         if(consensus_count <= guardians.length/2){
123             flag2 = false;
124         }
125         if(flag1 == true && flag2 == true){
126 
127             owner = change_list[counter-1];
128 
129         }
130         else{
131             return false;
132             }
133         return true;
134     }
```

Figure 4: Change Ownership

Guardian Client Algorithm

Algorithm 1: Initiator Process Algorithm for Ownership Change

Request(*wallet*, *new_address*) :

1: *req* = *wallet*.*ChangeOwnerRequest*(*new_address*)
2: **if** *req* = *false* **then**
3: return false
4: **end if**
5: **wait** asynchronously for $\lceil n/2 \rceil$ replies
6: *res* = *wallet*.*ChangeOwner*()
7: return *res*

Guardian Client Algorithm

Algorithm 2: Algorithm for accepting ownership change requests

accept(*wallet*) :

1: check shared memory asynchronously for ownership change request
2: **if** a request posted by a guardian process is present **then**
3: *wallet*.*agreechangeowner*()
4: **end if**

Correctness

Theorem 1

A change of ownership request initiated by Algorithm 1 will eventually result in change of the wallet's ownership.

Correctness

Proof.

Let p_i be a guardian node and p_i initiates a ownership change request in line 1.

Since $t < \lceil n/2 \rceil$, *eventually* $\lceil n/2 \rceil$ or more processes will reply to p_i's request, since p_i is a guardian node.

p_i will then execute the *changeOwner*() function in line 6 resulting in a change of ownership.

Correctness

Corollary 1

This social recovery wallet implementation solves the ownership change problem.

Wallet Security

A social recovery wallet is vulnerable to attacks:

1. **Attack 1:** The attacker directly transfers money from the wallet to themselves.
2. **Attack 2:** The ownership of the wallet is changed to a private key controlled by the attacker.
3. **Attack 3:** The attacker prevents agreement on an ownership change request initiated by a guardian process.

Wallet Security

Lemma 1

It is impossible to launch Attack 1 on this social recovery wallet implementation.

Wallet Security

Proof.

There is only one function that can transfer money from the smart contract implementing this social recovery wallet - *send_money*(), as shown in figure 5. Only the owner of the wallet can execute *send_money*(). This is ensured by line 169 where the function will be stopped from executing if the executing party does not have the correct private key.

□

```
165     function send_money(address _receiver, uint256 amount) public payable {
166
167         // Transfer money from wallet. Only owner can initiate this functionality.
168
169         require(msg.sender == owner, "Only owner can access funds!");
170         require(balance >= amount,"Not enough funds!");
171
172         payable(_receiver).transfer(amount);
173         balance -= amount;
174
175     }
```

Figure 5: Money Transfer functionality

Wallet Security

Theorem 2

It is impossible to launch Attack 2 on this social security wallet implementation.

Wallet Security

Proof.

In order to obtain ownership of the wallet, the attacker needs to populate the *change_list* array with an address.

Since *change_list* is a private state variable, it cannot be modified outside the smart contract.

The only function within the smart contract that modifies *change_list* is *ChangeOwnerRequest*().

As seen in figure 2, lines 49-60 ensure that only a guardian process can execute the critical region of *ChangeOwnerRequest*().

Therefore, an attacker cannot change the ownership of the wallet.

Wallet Security

Theorem 3

It is impossible to launch Attack 3 on this social security wallet implementation.

Wallet Security

Proof.

This is a proof by contradiction. Let $\{p_0, p_1, ...p_{n-1}\}$ be the set of guardian processes and p_n be a byzantine process.

Assume that p_i initiates ChangeOwnerRequest() but p_n successfully executes Attack 3.

This means that $\sum_{i=0}^{n-1}(agreement\_array[i] = counter) < \lceil n/2 \rceil$

There are at least $\lceil n/2 \rceil$ correct processes, therefore this is only possible if p_n modified *agreement_array*.

However, *agreement_array* is private and can only be modified when a guardian executes *agreechangeowner*().

Therefore, since p_n is not a guardian process it cannot modify *agreement_array*. This leads to a contradiction. □

Wallet Security

Corollary 2

This social recovery wallet implementation is secure.

Discussion

- The Implementation is geared towards securing funds that do not need to be accessed on a daily basis
- When used for daily transactions, ether will be wasted due to high gas prices
- Layer-2 solutions are required for widespread adoption

Discussion

- The implementation requires guardians to asynchronously read the blockchain log for ownership change requests
- Message passing between guardian nodes can simplify communication
- Only initiator process will have to asynchronously check whether its request has been accepted by a majority of guardians

Future Work

- Add functions to smart contract implementing a vault for large transactions
- Utilize existing agreement mechanism for vault functionality
- Explore off chain solutions for scalability and cost effectiveness
- Add a web user interface

Conclusion

- This social recovery wallet implementation solves the ownership change problem
- This social recovery wallet implementation is secure
- Social recovery wallets are a credible solution for the lost private key problem
- Social recovery wallets can also solve the stolen private key problem
- Cost is a bottleneck towards adoption of social recovery wallets
- Layer-2 optimizations need to be explored

Supporting slides

Gas Calculation

- Gas is a unit of measure used to represent the relative cost of OPCODEs
- Base cost - 21000 Gas
- Gas price is dynamic
- At the time of sending a transaction to the network, user has to specify gas price in ether

Gas Calculation

- Storage is expensive
- Smart Contracts should try to move computations off chain and focus on state changes