Report: Decentralized Inheritance Protocol

Noah Klaholz, Vincent Schall, Max Mendes Carvalho

November 2025

Contents

1 Introduction

No one can escape death - but what happens to your crypto when you die? According to [1], it is estimated that around 3.7 million Bitcoin are lost and unreciverable. One of the top reasons is death: crypto holders that pass away and failed to share access information with heirs will be responsible for inaccessible funds. Traditional inheritance systems are flawed: they take very long, are expensive and more often than not lead to conflicts within heirs.

2 Appendix

Appendices

Listing 1: smart contract

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.28;

import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/utils/ReentrancyGuard.sol";
import {IDeathOracle} from "./IDeathOracle.sol";

contract InheritanceProtocol is Ownable, ReentrancyGuard {

  IERC20 public immutable usdc;
  IDeathOracle public immutable deathOracle;
  address private notaryAddress;

  /**
  * Stores address and payout percentage amount (0-100) of a beneficiary.
  */
  struct Beneficiary {
    address payoutAddress;
    uint256 amount;
  }

  Beneficiary[10] private _beneficiaries;

  uint256 private _balance;
  State private _currentState;

  uint256 private _lastCheckIn;
  bool private _called = false;

  uint256 private constant NOT_FOUND = type(uint256).max;
  uint256 private constant MAX_BENEFICIARIES = 10;
  uint256 private constant MAX_PERCENTAGE = 100;
  uint256 private constant CHECK_IN_PERIOD = 90 * 1 days;
  uint256 private constant GRACE_PERIOD = 30 * 1 days;

  event BeneficiaryAdded(address indexed payoutAddress, uint256 amount, uint256
 index);
  event BeneficiaryRemoved(address indexed payoutAddress, uint256 index);
  event Deposited(uint256 amount);
  event Withdrawn(uint256 amount);
  event CheckedIn(uint256 timestamp);
  event StateChanged(uint256 timestamp, State from, State to);
  event PayoutMade(uint256 amount, address payoutAddress);
```

```
44      event TestEvent(string s);
45      event TestEventNum(uint s);
46
47      /**
48      * Initializes a new InheritanceProtocol.
49      * @param _usdcAddress address of the currency used (non-zero).
50      */
51      constructor(address _usdcAddress, address _deathOracleAddress, address
    _notaryAddress) Ownable(msg.sender) {
52          require(_usdcAddress != address(0), "USDC address zero");
53          require(_deathOracleAddress != address(0), "Death Oracle address zero");
54          usdc = IERC20(_usdcAddress);
55          deathOracle = IDeathOracle(_deathOracleAddress);
56          notaryAddress = _notaryAddress;
57          _currentState = State.ACTIVE;
58          _lastCheckIn = block.timestamp;
59      }
60
61      /// ---------- MODIFIERS ----------
62
63      /**
64      * This modifier requires the function call to be made before distribution.
65      */
66      modifier onlyPreDistribution() {
67          require(_currentState < State.DISTRIBUTION, "Cannot modify funds post-
    distribution");
68          _;
69      }
70
71      /**
72      * This modifier requires the function call to be made in the ACTIVE or
    WARNING phase
73      */
74      modifier onlyActiveWarning() {
75          require(_currentState < State.VERIFICATION, "Cannot make administrative
    changes without Owner check-In");
76          _;
77      }
78
79      /**
80      * This modifier requires the function call to be made in the DISTRIBUTION
    phase
81      */
82      modifier onlyDistribution() {
83          require(_currentState == State.DISTRIBUTION, "Can only make payouts in
    distribution phase");
84          _;
85      }
86
87      /**
88      * This modifier requires the function call to be made by the notary
89      */
90      modifier onlyNotary() {
91          require(msg.sender == notaryAddress, "Only notary can call this function");
92          _;
93      }
94
95      /// ---------- STATE MACHINE & CHECK-INS ----------
96
97      /**
98      * Defines the state of the contract.
99      *  - Active: mutable state, owner check-ins required.
100     *  - Warning: Missed check-in, notification sent at 90 days,
```

```
101       *    verification phase starts at 120 days.
102       *  - Verification: submission of death certificate (30 days).
103       *  - Distribution: distribute assets based on defined conditions.
104       */
105       enum State { ACTIVE, WARNING, VERIFICATION, DISTRIBUTION }
106
107       /**
108       * Updates the State in the State-Machine
109       * Should always be possible and accessible by anyone
110       * @return currentState after execution
111       */
112       function updateState() public returns (State) {
113         uint256 elapsed = uint256(block.timestamp) - _lastCheckIn;
114         State oldState = _currentState;
115
116         // --- Phase transitions in logical order ---
117
118         // If in ACTIVE and check-in expired  WARNING
119         if (_currentState == State.ACTIVE && elapsed > CHECK_IN_PERIOD) {
120           _currentState = State.WARNING;
121         }
122
123         // If in WARNING and grace period expired  VERIFICATION
124         if (_currentState == State.WARNING && elapsed > CHECK_IN_PERIOD +
    GRACE_PERIOD) {
125           _currentState = State.VERIFICATION;
126         }
127
128         // If in VERIFICATION and death confirmed  DISTRIBUTION
129         if (_currentState == State.VERIFICATION && deathOracle.isDeceased(owner()))
     {
130           _currentState = State.DISTRIBUTION;
131         }
132
133         emit StateChanged(block.timestamp, oldState, _currentState);
134
135         // Trigger payout if we reached DISTRIBUTION
136         if (_currentState == State.DISTRIBUTION) {
137           distributePayout();
138         }
139
140         return _currentState;
141       }
142
143
144       /**
145       * Changes the state of the contract to a given state.
146       * @param to the state to change to.
147       */
148       function changeState (State to) public {
149         require(to != _currentState, "Already in requested state");
150         emit StateChanged(block.timestamp, _currentState, to);
151         _currentState = to;
152       }
153
154       /**
155       * The owner checks in to verify that he's alive.
156       * Should be possible in active and warning state.
157       */
158       function checkIn() public onlyOwner {
159         require(_currentState == State.ACTIVE || _currentState == State.WARNING, "
    Need to be in active or warning state");
160         emit CheckedIn(block.timestamp);
```

```
        _lastCheckIn = block.timestamp;
      }

      /// ---------- BENEFICIARY HANDLING ----------


      /**
      * Finds the index of a beneficiary in the beneficiaries list.
      * @param _address the address whose index to find.
      * @return the index if the address is in the list, 'NOT_FOUND' otherwise.
      */
      function findBeneficiaryIndex(address _address) public view returns (uint256) {
        if (_address == address(0)) {
          return NOT_FOUND;
        }
        for (uint256 i = 0; i < MAX_BENEFICIARIES; i++) {
          if (_beneficiaries[i].payoutAddress == _address) {
            return i;
          }
        }
        return NOT_FOUND;
      }

      /**
      * Removes a beneficiary with a given address.
      * Only the owner can perform this action.
      * @param _address the address to remove.
      * Fails if the provided address is zero OR not in the list of beneficiaries.
      * @return true if the deletion was successful, false otherwise.
      */
      function removeBeneficiary(address _address) public onlyOwner onlyActiveWarning returns (bool) {
        checkIn();
        uint256 index = findBeneficiaryIndex(_address);
        if (index == NOT_FOUND) {
          return false;
        }
        delete _beneficiaries[index];
        emit BeneficiaryRemoved(_address, index);
        return true;
      }

      /**
      * Adds a beneficiary to the list.
      * Only the owner can perform this action.
      * Requirements:
      *  - List not full
      *  - Payout after adding <= 100
      * @param _address the address to add to the list.
      * @param _amount the payout amount related to this address.
      * @return true if the addition was successful, false otherwise.
      */
      function addBeneficiary(address _address, uint256 _amount) public onlyOwner onlyActiveWarning returns (bool) {
        checkIn();
        require(_address != address(0), "Invalid address");
        require(_amount > 0 && _amount <= MAX_PERCENTAGE, "Invalid amount");

        // Check for duplicate
        if (findBeneficiaryIndex(_address) != NOT_FOUND) {
          return false;
        }
```

```
221
222         uint256 currentSum = getDeterminedPayoutPercentage();
223         if (currentSum + _amount > MAX_PERCENTAGE) {
224           // it should not be possible to payout more than 100%
225           return false;
226         }
227
228         // Find empty slot
229         uint256 emptyIndex = NOT_FOUND;
230         for (uint256 i = 0; i < MAX_BENEFICIARIES; i++) {
231           if (_beneficiaries[i].payoutAddress == address(0)) {
232             emptyIndex = i;
233             break;
234           }
235         }
236
237         if (emptyIndex == NOT_FOUND) {
238           return false; // Max beneficiaries reached
239         }
240
241         _beneficiaries[emptyIndex] = Beneficiary({ payoutAddress: _address, amount:
      _amount });
242         emit BeneficiaryAdded(_address, _amount, emptyIndex);
243         return true;
244       }
245
246
247
248       /// ---------- BALANCE HANDLING ----------
249
250       /**
251       * Deposits a given amount of USDC.
252       * @param _amount the amount to deposit.
253       */
254       function deposit(uint256 _amount) external onlyOwner nonReentrant
     onlyPreDistribution {
255         checkIn();
256         require(_amount > 0, "Amount has to be greater than zero.");
257
258         usdc.transferFrom(msg.sender, address(this), _amount);
259         _balance += _amount;
260
261         //TODO add yield generating here -> Aave or something similar
262         emit Deposited(_amount);
263
264       }
265
266       /**
267       * Withdraws a given amount of USDC.
268       * @param _amount the amount to withdraw.
269       */
270       function withdraw(uint256 _amount) external onlyOwner nonReentrant
     onlyPreDistribution {
271         checkIn();
272         require(_amount > 0, "Amount has to be greater than zero.");
273         require(_balance >= _amount, "Insufficient balance");
274
275         _balance -= _amount;
276
277         usdc.transfer(msg.sender, _amount);
278         emit Withdrawn(_amount);
279       }
280
```

```
281       /// ---------- DEATH CERTIFICATION ----------
282
283       /**
284       * Upload the death verification to the chain
285       * Only callable by the notary
286       */
287       function uploadDeathVerification(bool _deceased, bytes calldata _proof)
      external onlyNotary{
288         deathOracle.setDeathStatus(owner(), _deceased, _proof);
289       }
290
291       /**
292       * Checks if the owner died by calling death certificate oracle.
293       * @return true if the owner died, else otherwise.
294       */
295       function checkIfOwnerDied() public view returns (bool) {
296         return deathOracle.isDeceased(owner());
297       }
298
299       /// ---------- DISTRIBUTION METHODS ----------
300
301       /**
302       * Distributes the payout based on definitions given by owner.
303       * Is only called in the updateState() Function, after death verification
304       */
305       function distributePayout() public {
306         require(!_called, "Payout can only be called once.");
307         _called = true;
308         uint256 count = getActiveCount();
309         Beneficiary[] memory activeBeneficiaries = getActiveBeneficiaries();
310         uint256 originalBalance = _balance;
311         for (uint256 i=0; i<count; i++) {
312           Beneficiary memory beneficiary = activeBeneficiaries[i];
313           uint256 amount = beneficiary.amount;
314           address payoutAddress = beneficiary.payoutAddress;
315
316           uint actualAmount = (originalBalance * amount) / MAX_PERCENTAGE;
317
318           // decision made: change balance value (should be 0 at the end)
319           // pros: good for checking / testing
320           // cons: just setting it to 0 would be less error-prone
321           _balance -= actualAmount;
322
323           usdc.transfer( payoutAddress, actualAmount);
324           emit PayoutMade(actualAmount, payoutAddress);
325         }
326       }
327
328       /// ---------- VIEW METHODS ----------
329
330       /**
331       * Checks if the currently defined payout is fully determined, meaning
332       * 100% of the balance is being spent.
333       * @return true if the full balance will be spent, false otherwise.
334       */
335       function isPayoutFullyDetermined() public view returns (bool) {
336         uint256 sum = getDeterminedPayoutPercentage();
337         return sum == MAX_PERCENTAGE;
338       }
339
340       /**
341       * Calculates the percentage amount of currently determined payout.
```

```
342       * @return a number between 0 and 100, equivalent to the combined relative
      payout.
343       */
344       function getDeterminedPayoutPercentage() public view returns (uint256) {
345         uint256 sum;
346         for (uint256 i = 0; i < MAX_BENEFICIARIES; i++) {
347           if (_beneficiaries[i].payoutAddress != address(0)) {
348             sum += _beneficiaries[i].amount;
349           }
350         }
351         return sum;
352       }
353
354       /**
355       * Gets the current balance.
356       * @return the balance of the combined deposited funds.
357       */
358       function getBalance() public view returns (uint256) {
359         return _balance; // If using Aave this might not work anymore
360       }
361
362       /**
363       * Getter for the beneficiaries list.
364       * @return the list of 10 beneficiaries (might contain empty slots).
365       */
366       function getBeneficiaries() public view returns (Beneficiary[10] memory) {
367         return _beneficiaries;
368       }
369
370       /**
371       * Counts the number of active beneficiaries.
372       * @return the number of active beneficiaries.
373       */
374       function getActiveCount() public view returns (uint256) {
375         uint256 count;
376         for (uint256 i = 0; i < MAX_BENEFICIARIES; i++) {
377           if (_beneficiaries[i].payoutAddress != address(0)) {
378             count++;
379           }
380         }
381         return count;
382       }
383
384       /**
385       * Gets only the active beneficiaries.
386       * @return an array of beneficiaries.
387       */
388       function getActiveBeneficiaries() public view returns (Beneficiary[] memory) {

389         uint256 activeCount = getActiveCount();
390         Beneficiary[] memory active = new Beneficiary[](activeCount);
391         uint256 count = 0;
392         for (uint256 i = 0; i < MAX_BENEFICIARIES; i++) {
393           if (_beneficiaries[i].payoutAddress != address(0)) {
394             active[count] = _beneficiaries[i];
395             count++;
396           }
397         }
398         return active;
399       }
400
401       /**
402       * Gets the current state of the contract.
```

```
403       * @return the current state.
404       */
405       function getState() public view returns (State) {
406         return _currentState;
407       }
408
409       /**
410       * Gets the last check-in time.
411       * @return the last check-in time.
412       */
413       function getLastCheckIn() public view returns (uint256) {
414         return _lastCheckIn;
415       }
416
417       }
418
```

A References

References

[1] Bidget. *How Many Bitcoin Have Been Lost?* Accessed 2025-11-06. 2025. URL: https://www.bitget.com/wiki/how-many-bitcoin-have-been-lost.