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 unrecoverable. One of the top reasons is death: crypto holders that passed 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. We want to solve these problems by introducing a decentralized inheritance protocol.
The idea is as follows: anyone can create a will by deploying the inheritance protocol contract. After that he can deposit coins, tokens and assets, as well as define beneficiaries or heirs by adding their wallet addresses. For each beneficiary, the owner can define a payout amount as percentage of the total deposited assets.
Furthermore, deposited assets are invested using [TODO add vesting protocol]. This allows the balance to grow instead of laying dry.
The owner has to check in at least every 90 days to verify that he's still alive. As long as these check-ins occur, there will be no payout. In case of death, trusted oracles are used to verify the death via death certificates, before initiating payout.

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 "./mocks/IDeathOracle.sol";
import {MockAavePool} from "./mocks/MockAavePool.sol";

contract InheritanceProtocol is Ownable, ReentrancyGuard {

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

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

  Beneficiary[10] private _beneficiaries;

  State private _currentState;

  uint256 private _lastCheckIn;
  bool private _called = false;
```

```
32
33          uint256 private constant NOT_FOUND = type(uint256).max;
34          uint256 private constant MAX_BENEFICIARIES = 10;
35          uint256 private constant MAX_PERCENTAGE = 100;
36          uint256 private constant CHECK_IN_PERIOD = 90 * 1 days;
37          uint256 private constant GRACE_PERIOD = 30 * 1 days;
38
39          event BeneficiaryAdded(address indexed payoutAddress, uint256 amount,
    uint256 index);
40          event BeneficiaryRemoved(address indexed payoutAddress, uint256 index);
41          event Deposited(uint256 amount);
42          event Withdrawn(uint256 amount);
43          event CheckedIn(uint256 timestamp);
44          event StateChanged(uint256 timestamp, State from, State to);
45          event PayoutMade(uint256 amount, address payoutAddress);
46          event TestEvent(string s);
47          event TestEventNum(uint s);
48
49          /**
50          * Initializes a new InheritanceProtocol.
51          * @param _usdcAddress address of the currency used (non-zero).
52          */
53          constructor(address _usdcAddress, address _deathOracleAddress, address
    _notaryAddress, address _aavePoolAddress) Ownable(msg.sender) {
54            require(_usdcAddress != address(0), "USDC address zero");
55            require(_deathOracleAddress != address(0), "Death Oracle address zero");
56            usdc = IERC20(_usdcAddress);
57            deathOracle = IDeathOracle(_deathOracleAddress);
58            notaryAddress = _notaryAddress;
59            aavePool = MockAavePool(_aavePoolAddress);
60            _currentState = State.ACTIVE;
61            _lastCheckIn = block.timestamp;
62          }
63
64          /// ---------- MODIFIERS ----------
65
66          /**
67          * This modifier requires the function call to be made before distribution.
68          */
69          modifier onlyPreDistribution() {
70            require(_currentState < State.DISTRIBUTION, "Cannot modify funds post-
    distribution");
71            _;
72          }
73
74          /**
75          * This modifier requires the function call to be made in the ACTIVE or
    WARNING phase
76          */
77          modifier onlyActiveWarning() {
78            require(_currentState < State.VERIFICATION, "Cannot make administrative
    changes without Owner check-In");
79            _;
80          }
81
82          /**
83          * This modifier requires the function call to be made in the DISTRIBUTION
    phase
84          */
85          modifier onlyDistribution() {
86            require(_currentState == State.DISTRIBUTION, "Can only make payouts in
    distribution phase");
87            _;
```

```
88          }
89
90          /**
91          * This modifier requires the function call to be made by the notary
92          */
93          modifier onlyNotary() {
94            require(msg.sender == notaryAddress, "Only notary can call this function")
    ;
95            _;
96          }
97
98          /// ---------- STATE MACHINE & CHECK-INS ----------
99
100         /**
101         * Defines the state of the contract.
102         *  - Active: mutable state, owner check-ins required.
103         *  - Warning: Missed check-in, notification sent at 90 days,
104         *    verification phase starts at 120 days.
105         *  - Verification: submission of death certificate (30 days).
106         *  - Distribution: distribute assets based on defined conditions.
107         */
108         enum State { ACTIVE, WARNING, VERIFICATION, DISTRIBUTION }
109
110         /**
111         * Updates the State in the State-Machine
112         * Should always be possible and accessible by anyone
113         * @return currentState after execution
114         */
115         function updateState() public returns (State) {
116           uint256 elapsed = uint256(block.timestamp) - _lastCheckIn;
117           State oldState = _currentState;
118
119           // --- Phase transitions in logical order ---
120
121           // If in ACTIVE and check-in expired  WARNING
122           if (_currentState == State.ACTIVE && elapsed > CHECK_IN_PERIOD) {
123             _currentState = State.WARNING;
124           }
125
126           // If in WARNING and grace period expired  VERIFICATION
127           if (_currentState == State.WARNING && elapsed > CHECK_IN_PERIOD +
    GRACE_PERIOD) {
128             _currentState = State.VERIFICATION;
129           }
130
131           // If in VERIFICATION and death confirmed  DISTRIBUTION
132           if (_currentState == State.VERIFICATION && deathOracle.isDeceased(owner())
    ) {
133             _currentState = State.DISTRIBUTION;
134           }
135
136           emit StateChanged(block.timestamp, oldState, _currentState);
137
138           // Trigger payout if we reached DISTRIBUTION
139           if (_currentState == State.DISTRIBUTION) {
140             distributePayout();
141           }
142
143           return _currentState;
144         }
145
146
147         /**
```

```
148         * Changes the state of the contract to a given state.
149         * @param to the state to change to.
150         */
151         function changeState (State to) public {
152           require(to != _currentState, "Already in requested state");
153           emit StateChanged(block.timestamp, _currentState, to);
154           _currentState = to;
155         }
156
157         /**
158         * The owner checks in to verify that he's alive.
159         * Should be possible in active and warning state.
160         */
161         function checkIn() public onlyOwner {
162           require(_currentState == State.ACTIVE || _currentState == State.WARNING,
    "Need to be in active or warning state");
163           emit CheckedIn(block.timestamp);
164           _lastCheckIn = block.timestamp;
165         }
166
167         /// ---------- BENEFICIARY HANDLING ----------
168
169
170         /**
171         * Finds the index of a beneficiary in the beneficiaries list.
172         * @param _address the address whose index to find.
173         * @return the index if the address is in the list, 'NOT_FOUND' otherwise.
174         */
175         function findBeneficiaryIndex(address _address) public view returns (
    uint256) {
176           if (_address == address(0)) {
177             return NOT_FOUND;
178           }
179           for (uint256 i = 0; i < MAX_BENEFICIARIES; i++) {
180             if (_beneficiaries[i].payoutAddress == _address) {
181               return i;
182             }
183           }
184           return NOT_FOUND;
185         }
186
187         /**
188         * Removes a beneficiary with a given address.
189         * Only the owner can perform this action.
190         * @param _address the address to remove.
191         * Fails if the provided address is zero OR not in the list of beneficiaries.

192         * @return true if the deletion was successful, false otherwise.
193         */
194         function removeBeneficiary(address _address) public onlyOwner
    onlyActiveWarning returns (bool) {
195           checkIn();
196           uint256 index = findBeneficiaryIndex(_address);
197           if (index == NOT_FOUND) {
198             return false;
199           }
200           delete _beneficiaries[index];
201           emit BeneficiaryRemoved(_address, index);
202           return true;
203         }
204
205         /**
206         * Adds a beneficiary to the list.
```

```
207         * Only the owner can perform this action.
208         * Requirements:
209         *  - List not full
210         *  - Payout after adding <= 100
211         * @param _address the address to add to the list.
212         * @param _amount the payout amount related to this address.
213         * @return true if the addition was successful, false otherwise.
214         */
215         function addBeneficiary(address _address, uint256 _amount) public onlyOwner
        onlyActiveWarning returns (bool) {
216           checkIn();
217           require(_address != address(0), "Invalid address");
218           require(_amount > 0 && _amount <= MAX_PERCENTAGE, "Invalid amount");
219
220           // Check for duplicate
221           if (findBeneficiaryIndex(_address) != NOT_FOUND) {
222             return false;
223           }
224
225           uint256 currentSum = getDeterminedPayoutPercentage();
226           if (currentSum + _amount > MAX_PERCENTAGE) {
227             // it should not be possible to payout more than 100%
228             return false;
229           }
230
231           // Find empty slot
232           uint256 emptyIndex = NOT_FOUND;
233           for (uint256 i = 0; i < MAX_BENEFICIARIES; i++) {
234             if (_beneficiaries[i].payoutAddress == address(0)) {
235               emptyIndex = i;
236               break;
237             }
238           }
239
240           if (emptyIndex == NOT_FOUND) {
241             return false; // Max beneficiaries reached
242           }
243
244           _beneficiaries[emptyIndex] = Beneficiary({ payoutAddress: _address,
        amount: _amount });
245           emit BeneficiaryAdded(_address, _amount, emptyIndex);
246           return true;
247         }
248
249
250
251         /// ---------- BALANCE HANDLING ----------
252
253         /**
254         * Deposits a given amount of USDC.
255         * @param _amount the amount to deposit.
256         */
257         function deposit(uint256 _amount) external onlyOwner nonReentrant
        onlyPreDistribution {
258           checkIn();
259           require(_amount > 0, "Amount has to be greater than zero.");
260
261           usdc.transferFrom(msg.sender, address(this), _amount);
262
263           usdc.approve(address(aavePool), _amount);
264
265           aavePool.supply(address(usdc), _amount, address(this));
266
```

```
267           emit Deposited(_amount);
268         }
269
270         /**
271         * Withdraws a given amount of USDC.
272         * @param _amount the amount to withdraw.
273         */
274         function withdraw(uint256 _amount) external onlyOwner nonReentrant
    onlyPreDistribution {
275           checkIn();
276           require(_amount > 0, "Amount has to be greater than zero.");
277           require(getBalance() >= _amount, "Insufficient balance");
278
279           aavePool.withdraw(address(usdc), _amount, address(this));
280
281           usdc.transfer(msg.sender, _amount);
282           emit Withdrawn(_amount);
283         }
284
285         /// ---------- DEATH CERTIFICATION ----------
286
287         /**
288         * Upload the death verification to the chain
289         * Only callable by the notary
290         */
291         function uploadDeathVerification(bool _deceased, bytes calldata _proof)
    external onlyNotary{
292           deathOracle.setDeathStatus(owner(), _deceased, _proof);
293         }
294
295         /**
296         * Checks if the owner died by calling death certificate oracle.
297         * @return true if the owner died, else otherwise.
298         */
299         function checkIfOwnerDied() public view returns (bool) {
300           return deathOracle.isDeceased(owner());
301         }
302
303         /// ---------- DISTRIBUTION METHODS ----------
304
305         /**
306         * Distributes the payout based on definitions given by owner.
307         * Is only called in the updateState() Function, after death verification
308         */
309         function distributePayout() public {
310           require(!_called, "Payout can only be called once.");
311           _called = true;
312           uint256 count = getActiveCount();
313           Beneficiary[] memory activeBeneficiaries = getActiveBeneficiaries();
314           uint256 balanceRemainingInPool = aavePool.getBalance(address(this));
315           uint256 withdrawnAmount = aavePool.withdraw(address(usdc),
    balanceRemainingInPool, address(this));
316           uint256 originalBalance = withdrawnAmount;
317           for (uint256 i=0; i<count; i++) {
318             Beneficiary memory beneficiary = activeBeneficiaries[i];
319             uint256 amount = beneficiary.amount;
320             address payoutAddress = beneficiary.payoutAddress;
321
322             uint actualAmount = (originalBalance * amount) / MAX_PERCENTAGE;
323
324             usdc.transfer( payoutAddress, actualAmount);
325             emit PayoutMade(actualAmount, payoutAddress);
326           }
```

```
327         }
328
329         /// ---------- VIEW METHODS ----------
330
331         /**
332         * Checks if the currently defined payout is fully determined, meaning
333         * 100% of the balance is being spent.
334         * @return true if the full balance will be spent, false otherwise.
335         */
336         function isPayoutFullyDetermined() public view returns (bool) {
337           uint256 sum = getDeterminedPayoutPercentage();
338           return sum == MAX_PERCENTAGE;
339         }
340
341         /**
342         * Calculates the percentage amount of currently determined payout.
343         * @return a number between 0 and 100, equivalent to the combined relative
      payout.
344         */
345         function getDeterminedPayoutPercentage() public view returns (uint256) {
346           uint256 sum;
347           for (uint256 i = 0; i < MAX_BENEFICIARIES; i++) {
348             if (_beneficiaries[i].payoutAddress != address(0)) {
349               sum += _beneficiaries[i].amount;
350             }
351           }
352           return sum;
353         }
354
355         /**
356         * Gets the current balance.
357         * @return the balance of the combined deposited funds.
358         */
359         function getBalance() public view returns (uint256) {
360           return aavePool.getBalance(address(this));
361         }
362
363         /**
364         * Getter for the beneficiaries list.
365         * @return the list of 10 beneficiaries (might contain empty slots).
366         */
367         function getBeneficiaries() public view returns (Beneficiary[10] memory) {
368           return _beneficiaries;
369         }
370
371         /**
372         * Counts the number of active beneficiaries.
373         * @return the number of active beneficiaries.
374         */
375         function getActiveCount() public view returns (uint256) {
376           uint256 count;
377           for (uint256 i = 0; i < MAX_BENEFICIARIES; i++) {
378             if (_beneficiaries[i].payoutAddress != address(0)) {
379               count++;
380             }
381           }
382           return count;
383         }
384
385         /**
386         * Gets only the active beneficiaries.
387         * @return an array of beneficiaries.
388         */
```

```
    function getActiveBeneficiaries() public view returns (Beneficiary[] memory)
{
      uint256 activeCount = getActiveCount();
      Beneficiary[] memory active = new Beneficiary[](activeCount);
      uint256 count = 0;
      for (uint256 i = 0; i < MAX_BENEFICIARIES; i++) {
        if (_beneficiaries[i].payoutAddress != address(0)) {
          active[count] = _beneficiaries[i];
          count++;
        }
      }
      return active;
    }

    /**
    * Gets the current state of the contract.
    * @return the current state.
    */
    function getState() public view returns (State) {
      return _currentState;
    }

    /**
    * Gets the last check-in time.
    * @return the last check-in time.
    */
    function getLastCheckIn() public view returns (uint256) {
      return _lastCheckIn;
    }

    }

```

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.