

Report: Decentralized Inheritance Protocol

Noah Klaholz, Vincent Schall, Max Mendes Carvalho

November 2025

Contents

1	Introduction	2
2	Appendix	2
	Appendices	2
A	References	9

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 unrecivable. 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.

2 Appendix

Appendices

Listing 1: smart contract

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.28;
3
4 import "@openzeppelin/contracts/access/Ownable.sol";
5 import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
6 import "@openzeppelin/contracts/utils/ReentrancyGuard.sol";
7 import {IDeathOracle} from "./IDeathOracle.sol";
8
9 contract InheritanceProtocol is Ownable, ReentrancyGuard {
10
11     IERC20 public immutable usdc;
12     IDeathOracle public immutable deathOracle;
13     address private notaryAddress;
14
15     /**
16      * Stores address and payout percentage amount (0-100) of a beneficiary.
17      */
18     struct Beneficiary {
19         address payoutAddress;
20         uint256 amount;
21     }
22
23     Beneficiary[10] private _beneficiaries;
24
25     uint256 private _balance;
26     State private _currentState;
27
28     uint256 private _lastCheckIn;
29     bool private _called = false;
30
31     uint256 private constant NOT_FOUND = type(uint256).max;
32     uint256 private constant MAX_BENEFICIARIES = 10;
33     uint256 private constant MAX_PERCENTAGE = 100;
34     uint256 private constant CHECK_IN_PERIOD = 90 * 1 days;
35     uint256 private constant GRACE_PERIOD = 30 * 1 days;
36
37     event BeneficiaryAdded(address indexed payoutAddress, uint256 amount, uint256 index);
38     event BeneficiaryRemoved(address indexed payoutAddress, uint256 index);
39     event Deposited(uint256 amount);
40     event Withdrawn(uint256 amount);
41     event CheckedIn(uint256 timestamp);
42     event StateChanged(uint256 timestamp, State from, State to);
43     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
52     _notaryAddress) Ownable(msg.sender) {
53         require(_usdcAddress != address(0), "USDC address zero");
54         require(_deathOracleAddress != address(0), "Death Oracle address zero");
55         usdc = IERC20(_usdcAddress);
56         deathOracle = IDeathOracle(_deathOracleAddress);
57         notaryAddress = _notaryAddress;
58         _currentState = State.ACTIVE;
59         _lastCheckIn = block.timestamp;
60     }
61
62     /// ----- MODIFIERS -----
63
64     /**
65      * This modifier requires the function call to be made before distribution.
66      */
67     modifier onlyPreDistribution() {
68         require(_currentState < State.DISTRIBUTION, "Cannot modify funds post-
69         distribution");
70         _;
71     }
72
73     /**
74      * This modifier requires the function call to be made in the ACTIVE or
75      * WARNING phase
76      */
77     modifier onlyActiveWarning() {
78         require(_currentState < State.VERIFICATION, "Cannot make administrative
79         changes without Owner check-In");
80         _;
81     }
82
83     /**
84      * This modifier requires the function call to be made in the DISTRIBUTION
85      * phase
86      */
87     modifier onlyDistribution() {
88         require(_currentState == State.DISTRIBUTION, "Can only make payouts in
89         distribution phase");
90         _;
91     }
92
93     /**
94      * This modifier requires the function call to be made by the notary
95      */
96     modifier onlyNotary() {
97         require(msg.sender == notaryAddress, "Only notary can call this function");
98         _;
99     }
100
101    /// ----- STATE MACHINE & CHECK-INS -----
102
103    /**
104     * Defines the state of the contract.
105     * - Active: mutable state, owner check-ins required.
106     * - Warning: Missed check-in, notification sent at 90 days,
107     *   verification phase starts at 120 days.
108     * - 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 +
125             GRACE_PERIOD) {
126             _currentState = State.VERIFICATION;
127         }
128
129         // If in VERIFICATION and death confirmed DISTRIBUTION
130         if (_currentState == State.VERIFICATION && deathOracle.isDeceased(owner()))
131         {
132             _currentState = State.DISTRIBUTION;
133         }
134
135         emit StateChanged(block.timestamp, oldState, _currentState);
136
137         // Trigger payout if we reached DISTRIBUTION
138         if (_currentState == State.DISTRIBUTION) {
139             distributePayout();
140         }
141
142
143     /**
144     * Changes the state of the contract to a given state.
145     * @param to the state to change to.
146     */
147     function changeState (State to) public {
148         require(to != _currentState, "Already in requested state");
149         emit StateChanged(block.timestamp, _currentState, to);
150         _currentState = to;
151     }
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, "
160             Need to be in active or warning state");
161         emit CheckedIn(block.timestamp);
162         _lastCheckIn = block.timestamp;
163     }

```

```

163
164     /// ----- BENEFICIARY HANDLING -----
165
166
167     /**
168      * Finds the index of a beneficiary in the beneficiaries list.
169      * @param _address the address whose index to find.
170      * @return the index if the address is in the list, 'NOT_FOUND' otherwise.
171      */
172     function findBeneficiaryIndex(address _address) public view returns (uint256)
173     {
174         if (_address == address(0)) {
175             return NOT_FOUND;
176         }
177         for (uint256 i = 0; i < MAX_BENEFICIARIES; i++) {
178             if (_beneficiaries[i].payoutAddress == _address) {
179                 return i;
180             }
181         }
182         return NOT_FOUND;
183     }
184
185     /**
186      * Removes a beneficiary with a given address.
187      * Only the owner can perform this action.
188      * @param _address the address to remove.
189      * Fails if the provided address is zero OR not in the list of beneficiaries.
190      * @return true if the deletion was successful, false otherwise.
191      */
192     function removeBeneficiary(address _address) public onlyOwner
193     onlyActiveWarning returns (bool) {
194         checkIn();
195         uint256 index = findBeneficiaryIndex(_address);
196         if (index == NOT_FOUND) {
197             return false;
198         }
199         delete _beneficiaries[index];
200         emit BeneficiaryRemoved(_address, index);
201         return true;
202     }
203
204     /**
205      * Adds a beneficiary to the list.
206      * Only the owner can perform this action.
207      * Requirements:
208      * - List not full
209      * - Payout after adding <= 100
210      * @param _address the address to add to the list.
211      * @param _amount the payout amount related to this address.
212      * @return true if the addition was successful, false otherwise.
213      */
214     function addBeneficiary(address _address, uint256 _amount) public onlyOwner
215     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();

```

```

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:
242     _amount });
243     emit BeneficiaryAdded(_address, _amount, emptyIndex);
244     return true;
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 * Withdraws a given amount of USDC.
267 * @param _amount the amount to withdraw.
268 */
269 function withdraw(uint256 _amount) external onlyOwner nonReentrant
onlyPreDistribution {
270     checkIn();
271     require(_amount > 0, "Amount has to be greater than zero.");
272     require(_balance >= _amount, "Insufficient balance");
273
274     _balance -= _amount;
275
276     usdc.transfer(msg.sender, _amount);
277     emit Withdrawn(_amount);
278 }
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)
288     external onlyNotary{
289         deathOracle.setDeathStatus(owner(), _deceased, _proof);
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
343      * payout.
344     */

```

```

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
390         uint256 activeCount = getActiveCount();
391         Beneficiary[] memory active = new Beneficiary[](activeCount);
392         uint256 count = 0;
393         for (uint256 i = 0; i < MAX_BENEFICIARIES; i++) {
394             if (_beneficiaries[i].payoutAddress != address(0)) {
395                 active[count] = _beneficiaries[i];
396                 count++;
397             }
398         }
399         return active;
400     }
401
402     /**
403      * Gets the current state of the contract.
404      * @return the current state.
405      */
406     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] Bitget. *How Many Bitcoin Have Been Lost?* Accessed 2025-11-06. 2025. URL: <https://www.bitget.com/wiki/how-many-bitcoin-have-been-lost>.