

# Report: Decentralized Inheritance Protocol

Noah Klaholz, Vincent Schall, Max Mendes Carvalho

November 2025

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Appendix</b>	<b>2</b>
	<b>Appendices</b>	<b>2</b>
<b>A</b>	<b>References</b>	<b>9</b>

## 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

```
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
38     index);
39     event BeneficiaryRemoved(address indexed payoutAddress, uint256 index);
40     event Deposited(uint256 amount);
41     event Withdrawn(uint256 amount);
42     event CheckedIn(uint256 timestamp);
43     event StateChanged(uint256 timestamp, State from, State to);
44     event PayoutMade(uint256 amount, address payoutAddress);
45     event TestEvent(string s);
46     event TestEventNum(uint s);
47
48     /**
49      * Initializes a new InheritanceProtocol.
50      * @param _usdcAddress address of the currency used (non-zero).
51      */
52     constructor(address _usdcAddress, address _deathOracleAddress, address
53     _notaryAddress) 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         _currentState = State.ACTIVE;
60         _lastCheckIn = block.timestamp;
61     }
62
63     /// ----- MODIFIERS -----
64
65     /**
66      * This modifier requires the function call to be made before distribution.
67      */
68     modifier onlyPreDistribution() {
69         require(_currentState < State.DISTRIBUTION, "Cannot modify funds post-
70         distribution");
71         _;
72     }
73
74     /**
75      * This modifier requires the function call to be made in the ACTIVE or WARNING
76      * phase
77      */
78     modifier onlyActiveWarning() {
79         require(_currentState < State.VERIFICATION, "Cannot make administrative
80         changes without Owner check-In");
81         _;
82     }
83
84     /**
85      * This modifier requires the function call to be made in the DISTRIBUTION 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

```

```

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

```

```

212     function addBeneficiary(address _address, uint256 _amount) public onlyOwner
213     onlyActiveWarning returns (bool) {
214         checkIn();
215         require(_address != address(0), "Invalid address");
216         require(_amount > 0 && _amount <= MAX_PERCENTAGE, "Invalid amount");
217
218         // Check for duplicate
219         if (findBeneficiaryIndex(_address) != NOT_FOUND) {
220             return false;
221         }
222
223         uint256 currentSum = getDeterminedPayoutPercentage();
224         if (currentSum + _amount > MAX_PERCENTAGE) {
225             // it should not be possible to payout more than 100%
226             return false;
227         }
228
229         // Find empty slot
230         uint256 emptyIndex = NOT_FOUND;
231         for (uint256 i = 0; i < MAX_BENEFICIARIES; i++) {
232             if (_beneficiaries[i].payoutAddress == address(0)) {
233                 emptyIndex = i;
234                 break;
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
255     onlyPreDistribution {
256         checkIn();
257         require(_amount > 0, "Amount has to be greater than zero.");
258
259         usdc.transferFrom(msg.sender, address(this), _amount);
260         _balance += _amount;
261
262         //TODO add yield generating here -> Aave or something similar
263         emit Deposited(_amount);
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
271     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)
288 external onlyNotary{
289     deathOracle.setDeathStatus(owner(), _deceased, _proof);
290 }
291
292 /**
293 * Checks if the owner died by calling death certificate oracle.
294 * @return true if the owner died, else otherwise.
295 */
296 function checkIfOwnerDied() public view returns (bool) {
297     return deathOracle.isDeceased(owner());
298 }
299
300 /// ----- DISTRIBUTION METHODS -----
301
302 /**
303 * Distributes the payout based on definitions given by owner.
304 * Is only called in the updateState() Function, after death verification
305 */
306 function distributePayout() public {
307     require(!_called, "Payout can only be called once.");
308     _called = true;
309     uint256 count = getActiveCount();
310     Beneficiary[] memory activeBeneficiaries = getActiveBeneficiaries();
311     uint256 originalBalance = _balance;
312     for (uint256 i=0; i<count; i++) {
313         Beneficiary memory beneficiary = activeBeneficiaries[i];
314         uint256 amount = beneficiary.amount;
315         address payoutAddress = beneficiary.payoutAddress;
316
317         uint actualAmount = (originalBalance * amount) / MAX_PERCENTAGE;
318
319         // decision made: change balance value (should be 0 at the end)
320         // pros: good for checking / testing
321         // cons: just setting it to 0 would be less error-prone
322         _balance -= actualAmount;
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.

```

```

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     */
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 _balance; // If using Aave this might not work anymore
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     */
389     function getActiveBeneficiaries() public view returns (Beneficiary[] memory) {
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             }
397         }

```

```

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