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 conflict between the 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 depositing coins, tokens and assets, as well as defining beneficiaries or heirs by adding their wallet addresses, is quick and easy with function calls to the contract. For each beneficiary, the owner can define a payout amount as percentage of the total deposited assets.
Furthermore, deposited assets are invested using Aave[1]. 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 (in most cases a notary) are used to verify the death via death certificates, before initiating payout.

2 Appendix

Appendices

```
1
2         // SPDX-License-Identifier: MIT
3         pragma solidity ^0.8.28;
4
5         import "@openzeppelin/contracts/access/Ownable.sol";
6         import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
7         import "@openzeppelin/contracts/utils/ReentrancyGuard.sol";
8         import {IDeathOracle} from "./mocks/IDeathOracle.sol";
9         import {MockAavePool} from "./mocks/MockAavePool.sol";
10
11        contract InheritanceProtocol is Ownable, ReentrancyGuard {
12
13            IERC20 public immutable usdc;
14            IDeathOracle public immutable deathOracle;
15            address private notaryAddress;
16            MockAavePool public aavePool;
17
18            // address for donations (underdetermined payout)
19            address private ourAddress;
20
21            /**
22             * Stores address and payout percentage amount (0-100) of
                  a beneficiary.
23             */
24            struct Beneficiary {
25                address payoutAddress;
```

[1] Aave — a decentralized lending protocol: supply crypto to earn interest via liquidity pools. https://aave.com/docs/developers/liquidity-pool

```
        uint256 amount;
    }

    Beneficiary[10] private _beneficiaries;

    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);
    event TestEvent(string s);
    event TestEventNum(uint s);

    /**
     * Initializes a new InheritanceProtocol.
     * @param _usdcAddress address of the currency used
         (non-zero).
     */
    constructor(address _usdcAddress, address
        _deathOracleAddress, address _notaryAddress, address
        _aavePoolAddress) Ownable(msg.sender) {
        require(_usdcAddress != address(0), "USDC address
            zero");
        require(_deathOracleAddress != address(0), "Death
            Oracle address zero");
        ourAddress =
            0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266;
        usdc = IERC20(_usdcAddress);
        deathOracle = IDeathOracle(_deathOracleAddress);
        notaryAddress = _notaryAddress;
        aavePool = MockAavePool(_aavePoolAddress);
        _currentState = State.ACTIVE;
        _lastCheckIn = block.timestamp;
    }

    /// ---------- MODIFIERS ----------

    /**
     * This modifier requires the function call to be made
         before distribution.
     */
    modifier onlyPreDistribution() {
```

```
74                require(_currentState < State.DISTRIBUTION, "Cannot
                      modify funds post-distribution");
75                _;
76            }
77
78            /**
79             * This modifier requires the function call to be made in
                   the ACTIVE or WARNING phase
80             */
81            modifier onlyActiveWarning() {
82                require(_currentState < State.VERIFICATION, "Cannot
                      make administrative changes without Owner
                      check-In");
83                _;
84            }
85
86            /**
87             * This modifier requires the function call to be made in
                   the DISTRIBUTION phase
88             */
89            modifier onlyDistribution() {
90                require(_currentState == State.DISTRIBUTION, "Can only
                      make payouts in distribution phase");
91                _;
92            }
93
94            /**
95             * This modifier requires the function call to be made by
                   the notary
96             */
97            modifier onlyNotary() {
98                require(msg.sender == notaryAddress, "Only notary can
                      call this function");
99                _;
100           }
101
102           /// ---------- STATE MACHINE & CHECK-INS ----------
103
104           /**
105            * Defines the state of the contract.
106            *   - Active: mutable state, owner check-ins required.
107            *   - Warning: Missed check-in, notification sent at 90
                   days,
108            *     verification phase starts at 120 days.
109            *   - Verification: submission of death certificate (30
                   days).
110            *   - Distribution: distribute assets based on defined
                   conditions.
111            */
112           enum State { ACTIVE, WARNING, VERIFICATION, DISTRIBUTION }
113
114           /**
115            * Updates the State in the State-Machine
116            * Should always be possible and accessible by anyone
117            * @return currentState after execution
118            */
119           function updateState() public returns (State) {
```

```
120             uint256 elapsed = uint256(block.timestamp) -
                    _lastCheckIn;
121             State oldState = _currentState;
122
123             // --- Phase transitions in logical order ---
124
125             // If in ACTIVE and check-in expired      WARNING
126             if (_currentState == State.ACTIVE && elapsed >
                    CHECK_IN_PERIOD) {
127                 _currentState = State.WARNING;
128             }
129
130             // If in WARNING and grace period expired
                    VERIFICATION
131             if (_currentState == State.WARNING && elapsed >
                    CHECK_IN_PERIOD + GRACE_PERIOD) {
132                 _currentState = State.VERIFICATION;
133             }
134
135             // If in VERIFICATION and death confirmed
                    DISTRIBUTION
136             if (_currentState == State.VERIFICATION &&
                    deathOracle.isDeceased(owner())) {
137                 _currentState = State.DISTRIBUTION;
138             }
139
140             emit StateChanged(block.timestamp, oldState,
                    _currentState);
141
142             // Trigger payout if we reached DISTRIBUTION
143             if (_currentState == State.DISTRIBUTION) {
144                 distributePayout();
145             }
146
147             return _currentState;
148         }
149
150         /**
151          * Changes the state of the contract to a given state.
152          * @param to the state to change to.
153          */
154         function changeState (State to) public {
155             require(to != _currentState, "Already in requested
                    state");
156             emit StateChanged(block.timestamp, _currentState, to);
157             _currentState = to;
158         }
159
160         /**
161          * The owner checks in to verify that he's alive.
162          * Should be possible in active and warning state.
163          */
164         function checkIn() public onlyOwner {
165             require(_currentState == State.ACTIVE || _currentState
                    == State.WARNING, "Need to be in active or warning
                    state");
166             emit CheckedIn(block.timestamp);
167             _lastCheckIn = block.timestamp;
```

```
168             }
169
170             /// ---------- BENEFICIARY HANDLING ----------
171
172             /**
173              * Finds the index of a beneficiary in the beneficiaries
                    list.
174              * @param _address the address whose index to find.
175              * @return the index if the address is in the list,
                    'NOT_FOUND' otherwise.
176              */
177             function findBeneficiaryIndex(address _address) public
                    view returns (uint256) {
178                 if (_address == address(0)) {
179                     return NOT_FOUND;
180                 }
181                 for (uint256 i = 0; i < MAX_BENEFICIARIES; i++) {
182                     if (_beneficiaries[i].payoutAddress == _address) {
183                         return i;
184                     }
185                 }
186                 return NOT_FOUND;
187             }
188
189             /**
190              * Removes a beneficiary with a given address.
191              * Only the owner can perform this action.
192              * @param _address the address to remove.
193              * Fails if the provided address is zero OR not in the
                    list of beneficiaries.
194              * @return true if the deletion was successful, false
                    otherwise.
195              */
196             function removeBeneficiary(address _address) public
                    onlyOwner onlyActiveWarning returns (bool) {
197                 checkIn();
198                 uint256 index = findBeneficiaryIndex(_address);
199                 if (index == NOT_FOUND) {
200                     return false;
201                 }
202                 delete _beneficiaries[index];
203                 emit BeneficiaryRemoved(_address, index);
204                 return true;
205             }
206
207             /**
208              * Adds a beneficiary to the list.
209              * Only the owner can perform this action.
210              * Requirements:
211              *  - List not full
212              *  - Payout after adding <= 100
213              * @param _address the address to add to the list.
214              * @param _amount the payout amount related to this
                    address.
215              * @return true if the addition was successful, false
                    otherwise.
216              */
```

```
217             function addBeneficiary(address _address, uint256 _amount)
                    public onlyOwner onlyActiveWarning returns (bool) {
218                 checkIn();
219                 require(_address != address(0), "Invalid address");
220                 require(_amount > 0 && _amount <= MAX_PERCENTAGE,
                        "Invalid amount");
221
222                 // Check for duplicate
223                 if (findBeneficiaryIndex(_address) != NOT_FOUND) {
224                     return false;
225                 }
226
227                 uint256 currentSum = getDeterminedPayoutPercentage();
228                 if (currentSum + _amount > MAX_PERCENTAGE) {
229                     // it should not be possible to payout more than
                            100%
230                     return false;
231                 }
232
233                 // Find empty slot
234                 uint256 emptyIndex = NOT_FOUND;
235                 for (uint256 i = 0; i < MAX_BENEFICIARIES; i++) {
236                     if (_beneficiaries[i].payoutAddress == address(0))
                            {
237                         emptyIndex = i;
238                         break;
239                     }
240                 }
241
242                 if (emptyIndex == NOT_FOUND) {
243                     return false; // Max beneficiaries reached
244                 }
245
246                 _beneficiaries[emptyIndex] = Beneficiary({
                        payoutAddress: _address, amount: _amount });
247                 emit BeneficiaryAdded(_address, _amount, emptyIndex);
248                 return true;
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                 bool donation = !isPayoutFullyDetermined();
313                 uint256 count = getActiveCount();
314                 Beneficiary[] memory activeBeneficiaries =
                        getActiveBeneficiaries();
315                 uint256 balanceRemainingInPool = getBalance();
```

```
316                 uint256 originalBalance =
                        aavePool.withdraw(address(usdc),
                        balanceRemainingInPool, address(this));
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                 if (donation) {
328                     // If the payout is not fully determined, the rest
                            of the balance will be sent to the developer
                            team.
329                     // For now this is hardcoded as the first address
                            generated by hardhat when running a local node.
330                     uint256 donatedAmount =
                            aavePool.withdraw(address(usdc), getBalance(),
                            address(this));
331                     usdc.transfer(ourAddress, donatedAmount);
332                     emit PayoutMade(donatedAmount, ourAddress);
333                 }
334             }
335
336             /// ---------- VIEW METHODS ----------
337
338             /**
339              * Checks if the currently defined payout is fully
                    determined, meaning
340              * 100% of the balance is being spent.
341              * @return true if the full balance will be spent, false
                    otherwise.
342              */
343             function isPayoutFullyDetermined() public view returns
                    (bool) {
344                 uint256 sum = getDeterminedPayoutPercentage();
345                 return sum == MAX_PERCENTAGE;
346             }
347
348             /**
349              * Calculates the percentage amount of currently
                    determined payout.
350              * @return a number between 0 and 100, equivalent to the
                    combined relative payout.
351              */
352             function getDeterminedPayoutPercentage() public view
                    returns (uint256) {
353                 uint256 sum;
354                 for (uint256 i = 0; i < MAX_BENEFICIARIES; i++) {
355                     if (_beneficiaries[i].payoutAddress != address(0))
                            {
356                         sum += _beneficiaries[i].amount;
357                     }
```

```
            }
            return sum;
        }

        /**
         * Gets the current balance.
         * @return the balance of the combined deposited funds.
         */
        function getBalance() public view returns (uint256) {
            return aavePool.getBalance(address(this));
        }

        /**
         * Getter for the beneficiaries list.
         * @return the list of 10 beneficiaries (might contain
             empty slots).
         */
        function getBeneficiaries() public view returns
            (Beneficiary[10] memory) {
            return _beneficiaries;
        }

        /**
         * Counts the number of active beneficiaries.
         * @return the number of active beneficiaries.
         */
        function getActiveCount() public view returns (uint256) {
            uint256 count;
            for (uint256 i = 0; i < MAX_BENEFICIARIES; i++) {
                if (_beneficiaries[i].payoutAddress != address(0))
                    {
                    count++;
                }
            }
            return count;
        }

        /**
         * Gets only the active beneficiaries.
         * @return an array of beneficiaries.
         */
        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;
        }

        /**
```

```
410             * Gets the current state of the contract.
411             * @return the current state.
412             */
413            function getState() public view returns (State) {
414                return _currentState;
415            }
416
417            /**
418             * Gets the last check-in time.
419             * @return the last check-in time.
420             */
421            function getLastCheckIn() public view returns (uint256) {
422                return _lastCheckIn;
423            }
424
425        }
```

Listing 1: smart contract

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.