Report: Decentralized Inheritance Protocol

Noah Klaholz, Vincent Schall, Max Mendes Carvalho

November 2025

Contents

1 Introduction

1.1 Motivation

No one can escape death - but what happens to your crypto when you die? According to [2], 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.

1.2 The 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 a 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. When a check in is missed there is a 30-day grace period during which a check in can be made again TODO

In case of death, trusted oracles (in most cases a notary) are used to verify the death via death certificates, before initiating the payout.

2 Smart Contract architecture

2.1 General Design and Flow

The inheritance protocol is implemented in a single smart contract and composes well-known primitives from *OpenZeppelin* for access control and safety [6]. In particular, the contract inherits from `Ownable` to grant the will's creator administrative privileges, uses `ReentrancyGuard` to protect sensitive functions, and interacts with funds through the `IERC20` interface of the ERC-20 standard [3]. For external integration, the contract talks to a lending pool (Aave-compatible mock) to invest idle balances and to a death oracle for verification.

The constructor wires all dependencies (token, death oracle, notary, pool) and initializes the state machine that models the life cycle of a will. The state machine transitions among four phases:

```
enum State { ACTIVE, WARNING, VERIFICATION, DISTRIBUTION }
```

Listing 1: Contract state machine

ACTIVE is the normal operating phase where the owner can manage beneficiaries and funds. If the owner misses a check-in for more than 90 days, the state moves to *WARNING*. After a 30-day grace period without check-in, the state advances to *VERIFICATION*. Once the death oracle confirms the owner's passing, the state becomes *DISTRIBUTION*, which triggers payout.

The `updateState()` function is public so that the notary, family members, or any third party can progress the state machine when the objective conditions are met. This creates an incentive-aligned mechanism: beneficiaries want the state to be up to date to receive their

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

funds and a trusted notary can be instructed to call this function regularily. When the state reaches *DISTRIBUTION*, the contract immediately invokes `distributePayout()` and emits a `StateChanged` event to support off-chain indexing.

2.2 Roles and Access Control

We use `Ownable` for a single privileged owner (the testator), and a dedicated *notary* address for external verification tasks [6]. Access is enforced by modifiers:

- `onlyOwner`: administrative actions (check-in, adding/removing beneficiaries, deposits/withdrawals) are restricted to the owner.
- `onlyNotary`: only the notary can upload death verification proofs.
- `onlyPreDistribution`: prevents fund mutations once the system is in the distribution phase.
- `onlyDistribution`: guards payout functions so they are callable only in the final phase.

Functions that transfer value also use the `nonReentrant` modifier from `ReentrancyGuard` to mitigate reentrancy (SWC-107) [6, 8].

2.3 Beneficiaries and Payout Logic

Beneficiaries are kept in a fixed-size array of at most ten entries to keep gas costs predictable and iteration bounded. Each entry stores a payout address and a percentage amount. The contract enforces that:

- No duplicate beneficiary addresses exist.
- The total determined payout never exceeds 100%.
- Add/remove operations are only allowed before distribution and require a fresh owner check-in.
- All administrative changes can only be made by the contract's owner.

On distribution, the contract retrieves the pool balance from Aave, computes each beneficiary's share by percentage, and transfers tokens accordingly by iterating through the list of beneficiaries. If the sum of percentages is below 100%, the residual is sent to a donation address to prevent funds from being stranded forever.

2.4 Funds Management and Aave Integration

The protocol accepts an ERC-20 token (MockUSDC in our deployment) via `deposit`. The owner first approves the contract to spend tokens, then the contract supplies tokens into an Aave-compatible pool (MockAavePool in our deployment) to accrue yield [3, 1]. Withdrawals reverse the flow: tokens are pulled from the pool and transferred back to the owner. Critical operations are protected with `nonReentrant` and disallowed after distribution. In production, gas efficiency and UX could be improved by supporting `permit` (EIP-2612) to avoid an extra approval transaction [5].

2.5 Death Verification and Oracles

Death verification is abstracted behind the `IDeathOracle` interface. The notary calls `uploadDeathVerification` with a boolean and opaque proof bytes; the oracle persists the attestation. The state machine polls the oracle by calling `isDeceased(owner())` and, if true, transitions to *DISTRIBUTION*. In our test setup we use a mock oracle to enable deterministic unit tests. For a production deployment, this component could be backed by a notarized registry, a government API gateway, or decentralized oracle networks. Additionally, `updateState()` could be automated using off-chain keepers (e.g., Chainlink Automation) to guarantee timely transitions without relying on manual calls [4].

2.6 Security Considerations

Our design follows standard Solidity best practices [7, 6]:

- Reentrancy protection on functions that transfer tokens (SWC-107) [8].
- Access control via explicit roles and clear phase guards (`onlyPreDistribution`, `onlyDistribution`).
- Use of `immutable` and `constant` for critical configuration to reduce runtime risk and gas.
- Bounded iteration over at most ten beneficiaries to avoid unbounded gas usage.
- Overflow/underflow safety from the Solidity 0.8.x checked arithmetic [7].

Threats and mitigations:

- **Oracle risk**: a compromised notary/oracle could wrongfully trigger distribution. This is mitigated organizationally (trusted notaries) and could be strengthened with multi-sig attestations or time delays.
- **Griefing/liveness**: anyone can call `updateState()`, but transitions are conditional and idempotent; no value is at risk.
- **Allowance management**: deposits rely on prior ERC-20 approvals; supporting `permit` reduces approval risks [5].
- **External calls**: interactions with the pool and token are performed after state updates and protected by `nonReentrant`. The donation transfer happens last to simplify reasoning.

2.7 Gas and Scalability

The fixed-size array avoids storage resizes and bounds loops to a maximum of ten iterations. Getter functions such as `getActiveBeneficiaries()` build a compact memory array for off-chain consumers, trading a small amount of gas for simpler client logic. State checks in `updateState()` are constant time. While the design targets personal wills (low on-chain scale), it remains economical for typical usage.

2.8 Known Limitations and Future Work

- **Single-asset support**: the current implementation handles one ERC-20 token instance. Extending to multiple assets would require per-asset accounting and distribution.
- **Maximum of ten beneficiaries**: chosen for simplicity and predictable gas; a dynamic structure with pagination could be introduced.
- **Oracle centralization**: production setups should consider decentralized attestations or multi-party notaries.

- **Automation**: integrating keepers would remove the need for manual `updateState()` calls [4].
- **UX improvements**: support for EIP-2612 `permit` and richer events to make indexing easier [5].

3 Decisions

//TODO rename this, but should be about why we did certain things, why abandon vesting for example etc.

4 Tool usage / tech stack

5 Appendix

Appendices

```

// 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;

    // address for donations (underdetermined payout)
    address private ourAddress;

    /**
     * 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;
```

```
    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() {
        require(_currentState < State.DISTRIBUTION, "Cannot
            modify funds post-distribution");
        _;
    }

    /**
     * This modifier requires the function call to be made in
         the ACTIVE or WARNING phase
     */
```

```
    modifier onlyActiveWarning() {
        require(_currentState < State.VERIFICATION, "Cannot make administrative changes without Owner check-In");
        _;
    }

    /**
     * This modifier requires the function call to be made in the DISTRIBUTION phase
     */
    modifier onlyDistribution() {
        require(_currentState == State.DISTRIBUTION, "Can only make payouts in distribution phase");
        _;
    }

    /**
     * This modifier requires the function call to be made by the notary
     */
    modifier onlyNotary() {
        require(msg.sender == notaryAddress, "Only notary can call this function");
        _;
    }

    /// ---------- STATE MACHINE & CHECK-INS ----------

    /**
     * Defines the state of the contract.
     *   - Active: mutable state, owner check-ins required.
     *   - Warning: Missed check-in, notification sent at 90 days,
     *     verification phase starts at 120 days.
     *   - Verification: submission of death certificate (30 days).
     *   - Distribution: distribute assets based on defined conditions.
     */
    enum State { ACTIVE, WARNING, VERIFICATION, DISTRIBUTION }

    /**
     * Updates the State in the State-Machine
     * Should always be possible and accessible by anyone
     * @return currentState after execution
     */
    function updateState() public returns (State) {
        uint256 elapsed = uint256(block.timestamp) - _lastCheckIn;
        State oldState = _currentState;

        // --- Phase transitions in logical order ---

        // If in ACTIVE and check-in expired      WARNING
        if (_currentState == State.ACTIVE && elapsed > CHECK_IN_PERIOD) {
            _currentState = State.WARNING;
```

```
        }

        // If in WARNING and grace period expired VERIFICATION
        if (_currentState == State.WARNING && elapsed > CHECK_IN_PERIOD + GRACE_PERIOD) {
            _currentState = State.VERIFICATION;
        }

        // If in VERIFICATION and death confirmed DISTRIBUTION
        if (_currentState == State.VERIFICATION && deathOracle.isDeceased(owner())) {
            _currentState = State.DISTRIBUTION;
        }

        emit StateChanged(block.timestamp, oldState, _currentState);

        // Trigger payout if we reached DISTRIBUTION
        if (_currentState == State.DISTRIBUTION) {
            distributePayout();
        }

        return _currentState;
    }

    /**
     * Changes the state of the contract to a given state.
     * @param to the state to change to.
     */
    function changeState (State to) public {
        require(to != _currentState, "Already in requested state");
        emit StateChanged(block.timestamp, _currentState, to);
        _currentState = to;
    }

    /**
     * The owner checks in to verify that he's alive.
     * Should be possible in active and warning state.
     */
    function checkIn() public onlyOwner {
        require(_currentState == State.ACTIVE || _currentState == State.WARNING, "Need to be in active or warning state");
        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;
        }
```

```
        uint256 currentSum = getDeterminedPayoutPercentage();
        if (currentSum + _amount > MAX_PERCENTAGE) {
            // it should not be possible to payout more than
                100%
            return false;
        }

        // Find empty slot
        uint256 emptyIndex = NOT_FOUND;
        for (uint256 i = 0; i < MAX_BENEFICIARIES; i++) {
            if (_beneficiaries[i].payoutAddress == address(0))
                {
                emptyIndex = i;
                break;
            }
        }

        if (emptyIndex == NOT_FOUND) {
            return false; // Max beneficiaries reached
        }

        _beneficiaries[emptyIndex] = Beneficiary({
            payoutAddress: _address, amount: _amount });
        emit BeneficiaryAdded(_address, _amount, emptyIndex);
        return true;
    }

    /// ---------- BALANCE HANDLING ----------

    /**
     * Deposits a given amount of USDC.
     * @param _amount the amount to deposit.
     */
    function deposit(uint256 _amount) external onlyOwner
        nonReentrant onlyPreDistribution {
        checkIn();
        require(_amount > 0, "Amount has to be greater than
            zero.");

        usdc.transferFrom(msg.sender, address(this), _amount);

        usdc.approve(address(aavePool), _amount);

        aavePool.supply(address(usdc), _amount, address(this));

        emit Deposited(_amount);
    }

    /**
     * Withdraws a given amount of USDC.
     * @param _amount the amount to withdraw.
     */
    function withdraw(uint256 _amount) external onlyOwner
        nonReentrant onlyPreDistribution {
        checkIn();
        require(_amount > 0, "Amount has to be greater than
            zero.");
```

```
        require(getBalance() >= _amount, "Insufficient balance");

        aavePool.withdraw(address(usdc), _amount, address(this));

        usdc.transfer(msg.sender, _amount);
        emit Withdrawn(_amount);
    }

    /// ---------- DEATH CERTIFICATION ----------

    /**
     * Upload the death verification to the chain
     * Only callable by the notary
     */
    function uploadDeathVerification(bool _deceased, bytes calldata _proof) external onlyNotary{
        deathOracle.setDeathStatus(owner(), _deceased, _proof);
    }

    /**
     * Checks if the owner died by calling death certificate oracle.
     * @return true if the owner died, else otherwise.
     */
    function checkIfOwnerDied() public view returns (bool) {
        return deathOracle.isDeceased(owner());
    }

    /// ---------- DISTRIBUTION METHODS ----------

    /**
     * Distributes the payout based on definitions given by owner.
     * Is only called in the updateState() Function, after death verification
     */
    function distributePayout() public {
        require(!_called, "Payout can only be called once.");
        _called = true;
        bool donation = !isPayoutFullyDetermined();
        uint256 count = getActiveCount();
        Beneficiary[] memory activeBeneficiaries = getActiveBeneficiaries();
        uint256 balanceRemainingInPool = getBalance();
        uint256 originalBalance = aavePool.withdraw(address(usdc), balanceRemainingInPool, address(this));
        for (uint256 i=0; i<count; i++) {
            Beneficiary memory beneficiary = activeBeneficiaries[i];
            uint256 amount = beneficiary.amount;
            address payoutAddress = beneficiary.payoutAddress;

            uint actualAmount = (originalBalance * amount) / MAX_PERCENTAGE;

```

```
                usdc.transfer( payoutAddress, actualAmount);
                emit PayoutMade(actualAmount, payoutAddress);
            }
            if (donation) {
                // If the payout is not fully determined, the rest
                    of the balance will be sent to the developer
                    team.
                // For now this is hardcoded as the first address
                    generated by hardhat when running a local node.
                uint256 donatedAmount =
                    aavePool.withdraw(address(usdc), getBalance(),
                    address(this));
                usdc.transfer(ourAddress, donatedAmount);
                emit PayoutMade(donatedAmount, ourAddress);
            }
        }

        /// ---------- VIEW METHODS ----------

        /**
         * Checks if the currently defined payout is fully
            determined, meaning
         * 100% of the balance is being spent.
         * @return true if the full balance will be spent, false
            otherwise.
         */
        function isPayoutFullyDetermined() public view returns
            (bool) {
            uint256 sum = getDeterminedPayoutPercentage();
            return sum == MAX_PERCENTAGE;
        }

        /**
         * Calculates the percentage amount of currently
            determined payout.
         * @return a number between 0 and 100, equivalent to the
            combined relative payout.
         */
        function getDeterminedPayoutPercentage() public view
            returns (uint256) {
            uint256 sum;
            for (uint256 i = 0; i < MAX_BENEFICIARIES; i++) {
                if (_beneficiaries[i].payoutAddress != address(0))
                    {
                    sum += _beneficiaries[i].amount;
                }
            }
            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;
        }

        /**
         * 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;
            }

        }
```

Listing 2: smart contract

A References

References

[1] *Aave V3: Pool Contract and Supplying Liquidity*. Accessed 2025-11-16. 2025. URL: https://docs.aave.com/developers/core-contracts/pool.

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

[3] V. Buterin and F. Vogelsteller. *ERC-20: Token Standard*. Accessed 2025-11-16. 2015. URL: https://eips.ethereum.org/EIPS/eip-20.

[4] *Chainlink Automation Documentation*. Accessed 2025-11-16. 2025. URL: https://docs.chain.link/chainlink-automation/introduction.

[5] *EIP-2612: Permit — 712-signed approvals*. Accessed 2025-11-16. 2020. URL: https://eips.ethereum.org/EIPS/eip-2612.

[6] *OpenZeppelin Contracts Documentation*. Accessed 2025-11-16. 2025. URL: https://docs.openzeppelin.com/contracts/5.x/.

[7] *Solidity Documentation v0.8.28*. Accessed 2025-11-16. 2025. URL: https://docs.soliditylang.org/en/v0.8.28/.

[8] *SWC-107: Reentrancy*. Accessed 2025-11-16. 2025. URL: https://swcregistry.io/docs/SWC-107.