Inheritance Manager Audit Report

Version 1.0

Jason Suárez

March 13, 2025

Inheritance Manager Audit Report

Jason Suárez

March 13, 2025

Prepared by: Jason Suárez Lead Auditors:

- Jason Suárez

Table of Contents

    * [M-1] Duplicate beneficiaries in `InheritanceManager::addBeneficiary` leads to uneven inheritance distribution
    * [M-1] Passing a non-existent address to `InheritanceManager::removeBeneficiary` silently removes the first beneficiary
    * [M-1] Lack of zero address validation in beneficiary management can lead to permanent fund loss
  - Informational
    * [QA-1] Wrong spelling of `beneficiary` in `InheritanceManager::addBeneficiery` and throughout the codebase
  - Gas
    * [GAS-1] Inefficient storage array access in loops across multiple functions

Protocol Summary

The Inheritance Manager is a smart contract wallet that implements a time-locked inheritance management system. It enables secure distribution of assets to designated beneficiaries based on predefined conditions, primarily a 90-day inactivity period. The contract maintains a list of beneficiaries and automates the allocation of inheritance when triggered.

Key features include:

- Time-based locks to ensure assets are only accessible after specified inactivity periods
- Trustless distribution of assets without intermediaries
- Backup wallet functionality
- Simple NFT minting to represent real-life assets (without legal enforceability)

Disclaimer

The Cura team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

Scope

The scope of this audit included the following smart contracts:

- `InheritanceManager.sol`
- `NFTFactory.sol`
- `modules/Trustee.sol`

The audit primarily focused on the security of the inheritance mechanism, fund management functions, and beneficiary management. As stated in the documentation, issues with the NFT functionality were only considered relevant if they could lead to loss of funds.

Roles

- `Owner`: The owner of the smart contract wallet
- `Beneficiary`: Any address set by the owner to inherit the smart contract and all its balances
- `Trustee`: An optional role that can be appointed by beneficiaries to reevaluate estate values or change payout assets

Audit Methodology

The audit followed a systematic approach:

1. **Manual Code Review**: Line-by-line review of the smart contracts to identify potential security vulnerabilities
2. **Test Suite Analysis**: Review of existing test cases and development of additional tests to validate findings
3. **Invariant Checking**: Verification of whether the documented core assumptions and invariants were properly implemented
4. **Static Analysis**: Use of automated tools to identify common vulnerabilities
5. **Gas Optimization Analysis**: Identification of potential gas savings opportunities

Executive Summary

Issues found

Severity	Number of issues found
High	1
Medium	3
Low	0
Info	1
Gas Optimizations	1
Total	6

Recommendations Summary

Based on the findings, we recommend the following key improvements:

1. Ensure consistent application of the deadline reset mechanism across all owner functions
2. Add proper validation for all user inputs, particularly for beneficiary addresses
3. Implement mechanisms to prevent duplicate beneficiaries
4. Use swap-and-pop pattern instead of `delete` for array elements
5. Add zero address validation to prevent fund loss
6. Fix spelling inconsistencies and optimize gas usage in loops

Most of the identified issues are related to the beneficiary management system and the inheritance triggering mechanism. These systems form the core security model of the contract and should be strengthened to ensure proper function under all scenarios.

Findings

High

[H-1] Missing deadline updates in owner functions breaks inheritance timelock protection

Description: Multiple owner-callable functions don't reset the inheritance timelock deadline, violating the core requirement that "EVERY transaction the owner does with this contract must reset the 90 days timer." The inconsistency is evident in the following functions:

```
+ InheritanceManager::sendERC20
+ InheritanceManager::sendETH
+ InheritanceManager::addBeneficiery
- InheritanceManager::contractInteractions
- InheritanceManager::createEstateNFT
- InheritanceManager::removeBeneficiary
```

Impact: The impact is severe as it undermines the core security model of the contract:

- Beneficiaries can gain access to funds while the owner is still active
- The owner could lose funds despite regular contract interaction
- This breaks the stated invariant that all owner activity should reset the timelock

If the owner only calls the functions that do not update the deadline like `InheritanceManager::contractInteractions` or `InheritanceManager::createEstateNFT`, beneficiaries could inherit funds prematurely, despite recent owner interactions.

Proof of Concept:

1. Add the following test to `InheritanceManager.t.sol`

```
function test_inheritanceManagerInteractionsUpdateDeadline() public {
        address user2 = makeAddr("user2");

        // First add beneficiary and roll to 90 days and check the
           deadline
        vm.startPrank(owner);
        im.addBeneficiery(user1);
        im.addBeneficiery(user2);
        vm.stopPrank();

        vm.warp(90 days);
        assertEq(1 + 90 days, im.getDeadline());
        vm.prank(user1);
        vm.expectRevert(InheritanceManager.
            InactivityPeriodNotLongEnough.selector);
        im.inherit();

        // Now call the functions that do not update the deadline
        vm.startPrank(owner);
        vm.warp(1 + 100 days);
        im.contractInteractions(address(0), abi.encode(address(0)), 0,
            false);
        vm.warp(1 + 120 days);
        im.createEstateNFT("our beach-house", 2000000, address(usdc));
        vm.stopPrank();

        // The deadline has not being updated
        assertEq(1 + 90 days, im.getDeadline());
        assertEq(vm.getBlockTimestamp(), 1 + 120 days);

        vm.prank(user1);
        im.inherit();

        assertEq(true, im.getIsInherited());
    }
```

2. run the test

```
forge test --mt test_inheritanceManagerInteractionsUpdateDeadline
```

Recommended Mitigation: Add the `_setDeadline()` function call to all owner-only functions to ensure consistent deadline updates:

```
function contractInteractions(address _target, bytes calldata _payload,
     uint256 _value, bool _storeTarget)
    external
    nonReentrant
    onlyOwner
{
    (bool success, bytes memory data) = _target.call{value: _value}(
        _payload);
    require(success, "interaction failed");
    if (_storeTarget) {
        interactions[_target] = data;
    }
+   _setDeadline();
}

function createEstateNFT(string memory _description, uint256 _value,
    address _asset) external onlyOwner {
    uint256 nftID = nft.createEstate(_description);
    nftValue[nftID] = _value;
```

```
        assetToPay = _asset;
+       _setDeadline();
}

function removeBeneficiary(address _beneficiary) external onlyOwner {
        uint256 indexToRemove = _getBeneficiaryIndex(_beneficiary);
        delete beneficiaries[indexToRemove];
+       _setDeadline();
}
```

Description: There's inconsistency in updating the deadline. Check the following functions: (in green are the functions that update the deadline and in red are the functions that do not)

```
+ InheritanceManager::sendERC20
+ InheritanceManager::sendETH
+ InheritanceManager::addBeneficiery
- InheritanceManager::contractInteractions
- InheritanceManager::createEstateNFT
- InheritanceManager::removeBeneficiary
```

Impact: If the owner only call the functions that do not update the deadline like `InheritanceManager::contractInteractions` or `InheritanceManager::createEstateNFT` it can lead to allowing inheritance despite owner interactions which breaks the core purpose of the contract.

Proof of Concept:

1. Add the following test to `InheritanceManager.t.sol`

```
function test_inheritanceManagerInteractionsUpdateDeadline() public {
        address user2 = makeAddr("user2");

        // First add beneficiary and roll to 90 days and check the
            deadline
        vm.startPrank(owner);
        im.addBeneficiery(user1);
        im.addBeneficiery(user2);
        vm.stopPrank();

        vm.warp(90 days);
        assertEq(1 + 90 days, im.getDeadline());
        vm.prank(user1);
        vm.expectRevert(InheritanceManager.
            InactivityPeriodNotLongEnough.selector);
        im.inherit();

        // Now call the functions that do not update the deadline
        vm.startPrank(owner);
        vm.warp(1 + 100 days);
        im.contractInteractions(address(0), abi.encode(address(0)), 0,
            false);
        vm.warp(1 + 120 days);
        im.createEstateNFT("our beach-house", 2000000, address(usdc));
        vm.stopPrank();

        // The deadline has not being updated
        assertEq(1 + 90 days, im.getDeadline());
        assertEq(vm.getBlockTimestamp(), 1 + 120 days);

        vm.prank(user1);
        im.inherit();

        assertEq(true, im.getIsInherited());
    }
```

2. run the test

```
forge test --mt test_inheritanceManagerInteractionsUpdateDeadline
```

Recommended Mitigation: You should decide which interactions should update the deadline and be consistent with the rest of the functions.

Medium

[M-1] Duplicate beneficiaries in `InheritanceManager::addBeneficiary` leads to uneven inheritance distribution

Description: The `InheritanceManager::addBeneficiery` function does not check if a beneficiary already exists in the array before adding them. This allows the same address to be added multiple times, creating duplicates in the beneficiaries array. When inheritance occurs, funds are distributed based on the number of entries in this array, not unique addresses.

```
function addBeneficiery(address _beneficiary) external onlyOwner {
    // Missing check for existing beneficiary
    beneficiaries.push(_beneficiary);
    _setDeadline();
}
```

Impact: This vulnerability breaks a core assumption of equal fund distribution among beneficiaries. A beneficiary added multiple times will receive a disproportionately larger share of the inheritance:

- If added twice, they'll receive double their intended share
- If added three times, they'll receive triple their intended share
- This reduces the inheritance share of other legitimate beneficiaries
- Could be exploited intentionally to favor certain beneficiaries

Proof of Concept:

1. Add the following test to `InheritanceManager.t.sol` > This test adds `user1` twice. We calculate the proportion of funds by dividing the total funds by the number of beneficiaries (4 entries for 3 unique addresses). > We then assert that `user1` gets twice the proportion while other users get one proportion each.

```
function test_withdrawInheritedFundsEtherDuplicateBeneficiary() public
    {
    address user2 = makeAddr("user2");
    address user3 = makeAddr("user3");
    vm.startPrank(owner);
    im.addBeneficiery(user1);
    im.addBeneficiery(user1);
    im.addBeneficiery(user2);
    im.addBeneficiery(user3);
    vm.stopPrank();
    vm.warp(1);
    vm.deal(address(im), 9e18);
    vm.warp(1 + 90 days);
    vm.startPrank(user1);
    im.inherit();
    im.withdrawInheritedFunds(address(0));
    vm.stopPrank();
    uint256 proportion = 9e18 / 4;
    assertEq(proportion * 2, user1.balance);
    assertEq(proportion, user2.balance);
    assertEq(proportion, user3.balance);
}
```

2. Run the test

```
$ forge test --mt test_withdrawInheritedFundsEtherDuplicateBeneficiary
```

Recommended Mitigation: Implement a check in the `addBeneficiery` function to prevent duplicate entries:

```
function addBeneficiery(address _beneficiary) external onlyOwner {
    require(!_beneficiariesContains(_beneficiary), "InheritanceManager:
        beneficiary already exists");
    beneficiaries.push(_beneficiary);
    _setDeadline();
}

function _beneficiariesContains(address _beneficiary) internal view
    returns (bool) {
    for (uint256 i = 0; i < beneficiaries.length; i++) {
        if (beneficiaries[i] == _beneficiary) {
            return true;
        }
    }
    return false;
}
```

[M-1] Passing a non-existent address to `InheritanceManager::removeBeneficiary` silently removes the first beneficiary

Description: When passing an address that is not in the beneficiaries array to `InheritanceManager::removeBeneficiary`, the function will silently remove the first element in the array. This happens because the `_getBeneficiaryIndex` function returns 0 (the default value for uint256) when the address is not found, causing the function to delete the beneficiary at index 0.

```
function removeBeneficiary(address _beneficiary) external onlyOwner {
    uint256 indexToRemove = _getBeneficiaryIndex(_beneficiary);
    delete beneficiaries[indexToRemove]; // Will remove index 0 if
        _beneficiary is not found
}

function _getBeneficiaryIndex(address _beneficiary) public view returns
     (uint256 _index) {
    for (uint256 i = 0; i < beneficiaries.length; i++) {
        if (_beneficiary == beneficiaries[i]) {
            _index = i;
            break;
        }
    }
    // If address not found, returns 0 (default value for uint256)
}
```

Impact: This vulnerability has severe consequences for inheritance management:

- If the owner is the only beneficiary, they will lose access to the funds without any warning
- If there are multiple beneficiaries, the first beneficiary will be silently removed from the inheritance list
- Funds could be inherited by the wrong parties as a result
- The function doesn't revert or emit an event, so the owner has no way to detect this error

Proof of Concept:

1. Add the following test to `InheritanceManager.t.sol`

```
function test_removeFirstBeneficiaryByPassingInexistentIndex() public {
    address user2 = makeAddr("user2");
    address inexistentUser = makeAddr("inexistentUser");

    vm.startPrank(owner);

    im.addBeneficiery(user1);
    im.addBeneficiery(user2);

    assertEq(user1, im.getBeneficiary(0)); // user1 is the first beneficiary
    assertEq(user2, im.getBeneficiary(1));

    im.removeBeneficiary(inexistentUser); // inexistentUser is not in the beneficiaries array
    vm.stopPrank();

    assert(user1 != im.getBeneficiary(0)); // user1 is not the first beneficiary anymore
}
```

2. Run the test

```
$ forge test --mt test_removeFirstBeneficiaryByPassingInexistentIndex
```

Recommended Mitigation: Modify the `removeBeneficiary` function to check if the address exists in the array and revert if it doesn't:

```
function removeBeneficiary(address _beneficiary) external onlyOwner {
    // Check if beneficiary exists
    bool found = false;
    uint256 indexToRemove = 0;

    for (uint256 i = 0; i < beneficiaries.length; i++) {
        if (beneficiaries[i] == _beneficiary) {
            indexToRemove = i;
            found = true;
            break;
        }
    }

    require(found, "InheritanceManager: beneficiary does not exist");

    // Remove and rearrange for gas efficiency
    beneficiaries[indexToRemove] = beneficiaries[beneficiaries.length - 1];
    beneficiaries.pop();

    // Add deadline update - currently missing
    _setDeadline();
}
```

Additionally, add a helper function and getter to improve contract usability:

```
function isBeneficiary(address _address) public view returns (bool) {
    for (uint256 i = 0; i < beneficiaries.length; i++) {
        if (beneficiaries[i] == _address) {
            return true;
        }
    }
    return false;
}

function getBeneficiariesCount() public view returns (uint256) {
    return beneficiaries.length;
}
```

[M-1] Lack of zero address validation in beneficiary management can lead to permanent fund loss

Description: The `InheritanceManager` contract doesn't validate against zero addresses when adding beneficiaries. Additionally, when removing beneficiaries, it uses `delete` which sets the value to the zero address without removing the entry from the array. This can lead to funds being distributed to the zero address (effectively burning them) when inheritance occurs.

```
function addBeneficiery(address _beneficiary) external onlyOwner {
    // No check for zero address
    beneficiaries.push(_beneficiary);
    _setDeadline();
}

function removeBeneficiary(address _beneficiary) external onlyOwner {
    uint256 indexToRemove = _getBeneficiaryIndex(_beneficiary);
    // This sets the element to address(0) but keeps it in the array
    delete beneficiaries[indexToRemove];
}
```

The `withdrawInheritedFunds` function will then attempt to send funds to all addresses in the array, including zero addresses:

```
function withdrawInheritedFunds(address _asset) external {
    // ...
    for (uint256 i = 0; i < divisor; i++) {
        address payable beneficiary = payable(beneficiaries[i]);
        (bool success,) = beneficiary.call{value: amountPerBeneficiary
            }("");
        require(success, "something went wrong");
    }
    // ...
}
```

Impact: This vulnerability can lead to several adverse outcomes:

- Funds can be permanently lost by being sent to address(0)
- Using `delete` on array elements doesn't reduce the array size, leading to confusion about the actual number of beneficiaries
- The inheritance share calculation becomes inaccurate if the array contains zero addresses
- In some cases, the contract might revert when trying to send ETH to address(0), preventing any beneficiary from receiving their inheritance

Proof of Concept:

1. Add the following test to `InheritanceManager.t.sol`

```
function test_zeroAddressBeneficiaryFundLoss() public {
    address user2 = makeAddr("user2");

    vm.startPrank(owner);
    // Add a valid beneficiary
    im.addBeneficiery(user1);
    // Add the zero address as a beneficiary
    im.addBeneficiery(address(0));
    // Add another valid beneficiary
    im.addBeneficiery(user2);

    // Fund the contract
    vm.deal(address(im), 3 ether);
    vm.stopPrank();

    // Wait for inheritance timelock to expire
    vm.warp(block.timestamp + 91 days);

    // Trigger inheritance
    vm.prank(user1);
    im.inherit();

    // Try to withdraw funds - this will attempt to send 1 ETH to
       address(0)
    vm.prank(user1);
    // This may revert or burn 1 ETH depending on the environment
    try im.withdrawInheritedFunds(address(0)) {
        // If it succeeds, check that 1 ETH was sent to address(0) (
           effectively burned)
        assertEq(address(0).balance, 1 ether);
        assertEq(user1.balance, 1 ether);
        assertEq(user2.balance, 1 ether);
    } catch {
        // If it reverts, no one gets their funds
        assertEq(address(im).balance, 3 ether);
        assertEq(user1.balance, 0);
```

```
            assertEq(user2.balance, 0);
        }
    }
```

2. Run the test

```
$ forge test --mt test_zeroAddressBeneficiaryFundLoss
```

Recommended Mitigation: Add zero address validation to the `addBeneficiery` function:

```
function addBeneficiery(address _beneficiary) external onlyOwner {
    require(_beneficiary != address(0), "InheritanceManager: zero
        address not allowed");

    // Check for duplicates (addressing another issue)
    for (uint256 i = 0; i < beneficiaries.length; i++) {
        if (beneficiaries[i] == _beneficiary) {
            revert("InheritanceManager: beneficiary already exists");
        }
    }

    beneficiaries.push(_beneficiary);
    _setDeadline();
}
```

For `removeBeneficiary`, instead of using `delete`, use the swap-and-pop pattern to actually remove the element from the array:

```
function removeBeneficiary(address _beneficiary) external onlyOwner {
    // Find the beneficiary
    bool found = false;
    uint256 indexToRemove = 0;

    for (uint256 i = 0; i < beneficiaries.length; i++) {
        if (beneficiaries[i] == _beneficiary) {
            indexToRemove = i;
            found = true;
            break;
        }
    }

    require(found, "InheritanceManager: beneficiary not found");

    // Swap with the last element and pop
    beneficiaries[indexToRemove] = beneficiaries[beneficiaries.length -
        1];
    beneficiaries.pop();

    _setDeadline();
}
```

Additionally, add a check in `withdrawInheritedFunds` to ensure no funds are sent to address(0):

```
function withdrawInheritedFunds(address _asset) external {
    // ...
    for (uint256 i = 0; i < divisor; i++) {
        address beneficiary = beneficiaries[i];
        if (beneficiary == address(0)) continue; // Skip zero addresses

        if (_asset == address(0)) {
            (bool success,) = payable(beneficiary).call{value:
                amountPerBeneficiary}("");
            require(success, "transfer failed");
        } else {
            IERC20(_asset).safeTransfer(beneficiary,
                amountPerBeneficiary);
        }
    }
    // ...
}
```

Informational

[QA-1] Wrong spelling of `beneficiary` in `InheritanceManager::addBeneficiery` and throughout the codebase

Description: The function name `addBeneficiery` in the `InheritanceManager` contract uses an incorrect spelling of "beneficiary" (written as "beneficiery"). This misspelling appears consistently in the contract code and tests. The correct spelling should be `addBeneficiary`.

```
// Incorrect spelling
function addBeneficiery(address _beneficiary) external onlyOwner {
    beneficiaries.push(_beneficiary);
    _setDeadline();
}
```

This inconsistency also appears in the test file `InheritanceManagerTest.t.sol` where the misspelled function is called multiple times.

Impact: While this doesn't directly impact the functionality of the contract, it introduces:

- Reduced code readability
- Potential developer confusion
- Inconsistency between function names and parameter names (which use the correct spelling)
- Maintenance challenges as developers might use both spellings in documentation or future code

Proof of Concept: No proof of concept is needed for this issue as it's directly visible in the contract code.

Recommended Mitigation: Rename the function to use the correct spelling:

```
// Correct spelling
function addBeneficiary(address _beneficiary) external onlyOwner {
    beneficiaries.push(_beneficiary);
    _setDeadline();
}
```

Also update all references to this function in tests and related documentation. Consider using a global search and replace to ensure consistency.

Gas

[GAS-1] Inefficient storage array access in loops across multiple functions

Description: Several functions in the InheritanceManager contract repeatedly access the storage array beneficiaries within loops, which is gas-inefficient. This occurs in at least three functions:

1. _getBeneficiaryIndex: Reads from storage in each loop iteration
2. withdrawInheritedFunds: Reads from storage in each loop iteration
3. buyOutEstateNFT: Reads from storage in each loop iteration

```
function _getBeneficiaryIndex(address _beneficiary) public view returns
    (uint256 _index) {
    for (uint256 i = 0; i < beneficiaries.length; i++) {
        if (_beneficiary == beneficiaries[i]) { // Storage read in loop
            _index = i;
            break;
        }
    }
}

function withdrawInheritedFunds(address _asset) external {
    // ...
    for (uint256 i = 0; i < divisor; i++) {
        address payable beneficiary = payable(beneficiaries[i]); //
            Storage read in loop
        // ...
    }
    // ...
}
```

Each SLOAD operation (reading from storage) costs 100 gas, which becomes significant when performed in loops, especially as the number of beneficiaries grows.

Impact:

- Higher gas costs for function calls, especially with many beneficiaries
- Potentially expensive or even prohibitive inheritance operations if there are numerous beneficiaries
- Less efficient execution compared to industry best practices

Proof of Concept: No specific proof of concept test is needed, as this is a standard gas optimization pattern recognized in smart contract development.

Recommended Mitigation: Copy storage arrays to memory before looping through them:

```
function _getBeneficiaryIndex(address _beneficiary) public view returns
     (uint256 _index) {
    address[] memory beneficiariesCache = beneficiaries; // Copy to
        memory once
    for (uint256 i = 0; i < beneficiariesCache.length; i++) {
        if (_beneficiary == beneficiariesCache[i]) { // Memory read (
            cheaper)
            _index = i;
            break;
        }
    }
}

function withdrawInheritedFunds(address _asset) external {
    // ...
    address[] memory beneficiariesCache = beneficiaries; // Copy to
        memory once
    for (uint256 i = 0; i < divisor; i++) {
        address payable beneficiary = payable(beneficiariesCache[i]);
            // Memory read
        // ...
    }
    // ...
}
```

Similarly, update the `buyOutEstateNFT` function to use a memory cache of the beneficiaries array.

Additionally, consider using uint256 instead of uint for loop counters to avoid conversion costs, though Solidity 0.8.x does this automatically in most cases.