Protocol Audit Report

Table of Contents

Protocol Summary

Inheriting crypto funds in cold or hot wallets is still an issue until this day, the Inheritance Manager contract implements a time-locked inheritance management system, enabling secure distribution of assets to designated beneficiaries. It uses time-based locks to ensure that assets are only accessible after a specified period. The contract maintains a list of beneficiaries, automating the allocation of inheritance based on predefined conditions. This system offers a trustless and transparent way to manage estate planning, ensuring assets are distributed as intended without the need for intermediaries.

Inheritance Manager can also be used as a backup for your wallet.

An extra gimmick is, that the owner of this contract is able to mint NFTs representing real life assets in an extremely simple way. We are aware that these NFTs have no legal value, we just integrated them, in case the beneficiaries agree to settle claims towards those assets on-chain within the contract. The really important part for us is, that the inheritance of funds works flawless after the wallet has been inactive for more than 90 days (standard configuration).

Disclaimer

I make 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

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

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

Audit Details

Scope

Roles

Executive Summary

Issues found

Severity	Number of issues found
High	7
Medium	1
Low	1
Info	0
Total	9

Findings

High

[H-1] Contract can't receive ether.

Description: The InheritanceManager contract does not implement a receive() or a payable fallback() function. As a result, it cannot accept direct ETH transfers via standard methods.

Impact: Without a payable fallback or receive function, the contract cannot accept ETH sent via typical transfers. This may limit its ability to receive funds from users or interact properly with other contracts that expect to transfer ETH.

Recommended Mitigation: Add a payable receive() function to allow the contract to accept ETH in `InheritanceManager`

```
receive() external payable {}
```

[H-2] Faulty Reentrancy Guard Implementation

Description: The reentrancy guard in `InheritanceManager::nonReentrant` incorrectly reads from transient storage slot 1 instead of slot 0. Since tload(1) always returns zero, the reentrancy check is effectively disabled.

Impact: Critical functions such as `InheritanceManager::sendETH`, `InheritanceManager::contractInteractions`, `InheritanceManager::withdrawInheritedFunds` are vulnerable to reentrancy attacks, potentially allowing an attacker to drain funds from the contract.

Recommended Mitigation: Modify the reentrancy guard to read from slot 0 instead:

```
modifier nonReentrant() {
    assembly {
-           if tload(1) { revert(0, 0) }
+           if tload(0) { revert(0, 0) }
        tstore(0, 1)
    }
    _;
    assembly {
        tstore(0, 0)
    }
    }
```

[H-3] Incorrect Removal of Beneficiaries from Array

Description: In `InheritanceManager::removeBeneficiary` using the 'delete' keyword sets the element to the zero address rather than removing it from the array. This leaves a "hole" in the array.

Impact: This bug interferes with subsequent logic in functions like `InheritanceManager::buyOutEstateNFT` and `InheritanceManager::withdrawInheritedFunds` , potentially causing incorrect calculations and distribution errors. For instance, the zero address may be inadvertently included in the beneficiary count, resulting in misallocated funds.

Proof of Concept: Include the following tests in the `InheritanceManagerTest.t.sol` file:

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

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

    vm.warp(1);
    vm.deal(address(im), 10e10);
    vm.warp(1 + 90 days);

    vm.startPrank(user1);
    im.inherit();
    vm.stopPrank();
```

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

    function testWithdrawInheritedFundsAfterDeleteBeneficiary() public {
        address user2 = makeAddr("user2");
        address user3 = makeAddr("user3");

        im.addBeneficiery(user1);
        im.addBeneficiery(user2);
        im.addBeneficiery(user3);
        im.removeBeneficiary(user3);

        vm.warp(1);

        uint256 value = 3e18;
        vm.deal(address(im), value);

        vm.warp(1 + 90 days);

        vm.startPrank(user1);
        im.inherit();
        im.withdrawInheritedFunds(address(0));
        vm.stopPrank();

        uint256 beneficiariesAmount = 2;
        uint256 sharePerUser = value / beneficiariesAmount;

        assertLt(user1.balance, sharePerUser);
        assertLt(user2.balance, sharePerUser);
    }
```

Recommended Mitigation: Swap the element to remove with the last element and then use the pop operation to remove it from the array. This approach is gas efficient and avoids leaving a zero address in the array.

```
function removeBeneficiary(address _beneficiary) external onlyOwner {
    uint256 indexToRemove = _getBeneficiaryIndex(_beneficiary);
-   delete beneficiaries[indexToRemove];
+   require(indexToRemove < beneficiaries.length, "Beneficiary not found");
+   beneficiaries[indexToRemove] = beneficiaries[beneficiaries.length - 1];
+   beneficiaries.pop();
}
```

[H-4] Missing msg.sender Check Allows Unauthorized Ownership Transfer

Description: `InheritanceManager::inherit` lacks a proper check to ensure that only legitimate beneficiaries can call it. As a result, any caller can trigger the line 'owner = msg.sender'.

Impact: This vulnerability enables any arbitrary address to assume ownership of the contract by calling `InheritanceManager::inherit`, representing a critical security risk.

Recommended Mitigation: Introduce a modifier (e.g., 'onlyBeneficiary') to restrict access to beneficiaries only:

```
modifier onlyBeneficiary() {
    bool isBeneficiary = false;
    for (uint256 i = 0; i < beneficiaries.length; i++) {
        if (beneficiaries[i] == msg.sender) {
            isBeneficiary = true;
            break;
        }
    }
    require(isBeneficiary, "Caller is not a beneficiary");
    _;
}
```

Apply this modifier to the `InheritanceManager::inherit`as needed.

[H-5] Precision Loss Due to Order of Operations in Calculations

Description: In `InheritanceManager::buyOutEstateNFT`, the calculation '(value / divisor) * multiplier' performs division before multiplication. Since Solidity uses integer division, truncation can occur if 'value' is not perfectly divisible by 'divisor'.

Impact: Beneficiaries may receive less than their intended share due to precision loss. This can lead to financial discrepancies and disputes among beneficiaries.

Proof of Concept: Include the following test in the `InheritanceManagerTest.t.sol` file:

```
function testBuyOutEstateNFTPrecisionLoss() public {
    address user2 = makeAddr("user2");
    address user3 = makeAddr("user3");
    address user4 = makeAddr("user4");
    address user5 = makeAddr("user5");

    vm.warp(1);

    im.addBeneficiery(user1);
    im.addBeneficiery(user2);
    im.addBeneficiery(user3);
    im.addBeneficiery(user4);
    im.addBeneficiery(user5);

    uint256 nftValue = 3;
    im.createEstateNFT("our beach-house", nftValue, address(usdc));

    uint256 startAmount = 1e6;
    usdc.mint(user5, startAmount);
```

```
    vm.warp(1 + 90 days);

    vm.startPrank(user5);
    usdc.approve(address(im), startAmount);
    im.inherit();
    im.buyOutEstateNFT(1);
    vm.stopPrank();

    uint256 userBalanceAfterBuyOut = usdc.balanceOf(user5);

    assertEq(startAmount, userBalanceAfterBuyOut);
}
```

Recommended Mitigation: Reorder the calculation to perform multiplication before division:

```
function buyOutEstateNFT(uint256 _nftID) external onlyBeneficiaryWithIsInherited {
    uint256 value = nftValue[_nftID];
    uint256 divisor = beneficiaries.length;
    uint256 multiplier = beneficiaries.length - 1;
-   uint256 finalAmount = (value / divisor) * multiplier;
+   uint256 finalAmount = value * multiplier / divisor;
    ...
}
```

[H-6] Faulty Loop Logic in `InheritanceManager::buyOutEstateNFT`

Description: Within `InheritanceManager::buyOutEstateNFT`, the loop that distributes funds exits immediately upon finding the caller, due to a return statement. This prematurely terminates the loop and prevents funds from being distributed to the remaining beneficiaries.

Impact: This bug may result in incomplete distribution of funds, causing some beneficiaries to receive no funds at all during a buyout, leading to potential disputes or financial losses.

Proof of Concept: Include the following test in the `InheritanceManagerTest.t.sol` file:

```
function testBuyOutEstateNFTLoopCheck() public {
    address user2 = makeAddr("user2");
    address user3 = makeAddr("user3");

    vm.warp(1);

    im.addBeneficiery(user1);
    im.addBeneficiery(user2);
    im.addBeneficiery(user3);

    uint256 nftValue = 3e6;
    im.createEstateNFT("our beach-house", nftValue, address(usdc));
```

```
        usdc.mint(user1, 4e6);

        vm.warp(1 + 90 days);

        vm.startPrank(user1);
        usdc.approve(address(im), 4e6);
        im.inherit();
        im.buyOutEstateNFT(1);
        vm.stopPrank();

        uint256 user2Balance = usdc.balanceOf(user2);
        uint256 user3Balance = usdc.balanceOf(user3);

        assertEq(user2Balance, 0);
        assertEq(user3Balance, 0);
    }
```

Recommended Mitigation: Replace the 'return' statement with 'continue' so that the loop processes all beneficiaries:

```
function buyOutEstateNFT(uint256 _nftID) external onlyBeneficiaryWithIsInherited {
    uint256 value = nftValue[_nftID];
    uint256 divisor = beneficiaries.length;
    uint256 multiplier = beneficiaries.length - 1;
    uint256 finalAmount = (value / divisor) * multiplier;
    IERC20(assetToPay).safeTransferFrom(msg.sender, address(this), finalAmount);
    for (uint256 i = 0; i < beneficiaries.length; i++) {
        if (msg.sender == beneficiaries[i]) {
-           return;
+           continue;
        } else {
            IERC20(assetToPay).safeTransfer(beneficiaries[i], finalAmount /
divisor);
        }
    }
    nft.burnEstate(_nftID);
}
```

[H-7] Incorrect Calculation Logic in `InheritanceManager::buyOutEstateNFT`

Description: The calculation for fund distribution in `InheritanceManager::buyOutEstateNFT` incorrectly computes each beneficiary's share. Beneficiaries may receive an amount that is lower than expected.

Impact: This error results in beneficiaries receiving less funds than they are entitled to, potentially leading to financial shortfalls and disputes.

Proof of Concept: Include the following test in the `InheritanceManagerTest.t.sol` file:

```
function testBuyOutEstateNFTSharesCheck() public {
    address user2 = makeAddr("user2");
    address user3 = makeAddr("user3");

    vm.warp(1);

    im.addBeneficiery(user1);
    im.addBeneficiery(user2);
    im.addBeneficiery(user3);

    uint256 nftValue = 3e6;
    im.createEstateNFT("our beach-house", nftValue, address(usdc));

    usdc.mint(user3, 4e6);

    vm.warp(1 + 90 days);

    vm.startPrank(user3);
    usdc.approve(address(im), 4e6);
    im.inherit();
    im.buyOutEstateNFT(1);
    vm.stopPrank();

    uint256 user1Balance = usdc.balanceOf(user1);
    uint256 user2Balance = usdc.balanceOf(user2);

    uint256 beneficiariesAmount = 3;
    uint256 sharePerUser = nftValue / beneficiariesAmount;

    assertLt(user1Balance, sharePerUser);
    assertLt(user2Balance, sharePerUser);
}
```

Recommended Mitigation: Adjust the calculation to correctly distribute the funds:

```
function buyOutEstateNFT(uint256 _nftID) external onlyBeneficiaryWithIsInherited {
    uint256 value = nftValue[_nftID];
    uint256 divisor = beneficiaries.length;
    uint256 multiplier = beneficiaries.length - 1;
    uint256 finalAmount = (value / divisor) * multiplier;
    IERC20(assetToPay).safeTransferFrom(msg.sender, address(this), finalAmount);
    for (uint256 i = 0; i < beneficiaries.length; i++) {
        if (msg.sender == beneficiaries[i]) {
            return;
        } else {
-           IERC20(assetToPay).safeTransfer(beneficiaries[i], finalAmount /
divisor);
+           IERC20(assetToPay).safeTransfer(beneficiaries[i], finalAmount /
multiplier);
        }
    }
```

```
        nft.burnEstate(_nftID);
    }
```

Medium

[M-1] Vulnerability in Loop-Based Transfers

Description: `InheritanceManager::withdrawInheritedFunds` uses 'require' statements and 'safeTransfer' calls within a loop. If any single transfer fails—perhaps because a beneficiary is a contract with a rejecting fallback—then the entire transaction reverts.

Impact: A single failing beneficiary transfer can block the distribution process, leaving funds locked in the contract and potentially preventing other beneficiaries from receiving their rightful share.

Recommended Mitigation: Adopt a pull-payment mechanism where each beneficiary withdraws their funds individually. This isolates transfer failures to individual withdrawals rather than impacting the entire transaction.

Low

[L-1] Missing ETH Balance Check in `InheritanceManager::sendETH`

Description: `InheritanceManager::sendETH` does not verify whether the contract holds enough ETH before attempting to send funds.

Impact: If the contract's balance is insufficient, an ETH transfer may fail, leading to transaction reversion or unexpected behavior.

Recommended Mitigation: Add a balance check at the beginning of `InheritanceManager::sendETH`

```
if (address(this).balance < _amount) {
    revert InsufficientBalance();
}
```