# GivingThanks Security Audit Report by MK (14th DEC 2024)

# High Risk Findings

## [H-1] Incorrect Logic in isVerified Function

**Description:** The `isVerified` function in the `CharityRegistry` contract contains incorrect logic that may cause unverified charities to be incorrectly identified as verified. This function currently checks if a charity is *registered*, not if it is *verified*, which could allow unverified charities to appear eligible to receive donations.

**Impact:** Allowing unverified charities to appear verified undermines the donation process by making funds accessible to unapproved entities. This could lead to potential misuse of funds, as unverified charities might receive donations they were not meant to access.

**Proof of Concept:**

1. The `isVerified` function in the `CharityRegistry` contract is intended to check if a charity has been verified by the admin. However, it currently checks the `registeredCharities` mapping instead of the `verifiedCharities` mapping:

```
function isVerified(address charity) public view returns (bool) {
  return registeredCharities[charity];
}
```

2. This logic error means that any registered charity will be considered "verified," even if the admin has not verified it. This could allow unverified charities to be mistakenly recognized as eligible to receive donations, bypassing the verification process.

**Recommended Mitigation:**

1. Modify the `isVerified` function to check the `verifiedCharities` mapping instead of `registeredCharities`, ensuring only charities marked as verified by the admin are eligible for donations.

Suggested Fix:

```
function isVerified(address charity) public view returns (bool) {
  return verifiedCharities[charity];
}
```

2. This change ensures that only charities marked as verified by the admin can be considered eligible for donations, maintaining the integrity of the verification process and the security of donor funds.

## [H-2] Unauthorized Registry Address Update Vulnerability

**Description:** The `GivingThanks::updateRegistry` function currently lacks access control, allowing any user to modify the `registry` address. Without restrictions, a malicious user could replace the `registry` with an unauthorized or malicious address. This could compromise the protocol, as functions such as `donate` depend on verified charity addresses from the `registry`. If the `registry` points to an unauthorized address, critical operations will revert due to unverified charity addresses, rendering the protocol inoperable.

**Impact:** This vulnerability allows unauthorized users to alter the `registry` address, potentially redirecting it to a malicious contract and making the protocol unusable. The integrity and functionality of the protocol are compromised, as legitimate `donate` transactions would fail when the `registry` points to an unverified address.

**Recommended Mitigation:**

1. Implement an access control check in the `updateRegistry` function to restrict updates to the contract owner only

```
    function updateRegistry(address _registry) public {
+       require(msg.sender == owner, "Only owner can update registry");
        registry = CharityRegistry(_registry);
    }
```

# Medium Risk Findings

## [M-1] Incorrect Registry Initialization in GivingThanks Contract Constructor

**Description:** The `GivingThanks` contract's constructor mistakenly assigns the deployer's address (`msg.sender`) to the `registry` variable instead of the provided `_registry` address. This misconfiguration causes the contract to fail when attempting to verify a charity through the `donate` function, resulting in reverted transactions and making the donation functionality unusable.

**Impact:** Without access to the actual `CharityRegistry` contract, the `GivingThanks` contract cannot verify charities, leading to reverted transactions for all donation attempts. This issue effectively breaks the donation feature, preventing donors from contributing to charities.

**Proof of Concept:**

1. In the `GivingThanks` contract's constructor, the `registry` variable is set using `msg.sender` rather than the `_registry` parameter. Consequently, instead of pointing to the deployed `CharityRegistry` contract, `registry` holds the deployer's address.

```
// Original constructor with incorrect registry initialization
constructor(address _registry) ERC721("DonationReceipt", "DRC") {
    registry = CharityRegistry(msg.sender); // Incorrect assignment
    owner = msg.sender;
```

```
    tokenCounter = 0;
}
```

2. This misconfiguration leads to failed charity verification in the `donate` function, as the `isVerified` check calls the incorrect address (the deployer's address) instead of the `CharityRegistry` contract. As a result, all attempts to donate revert, rendering the core functionality of the contract unusable.

**Recommended Mitigation:**

1. Modify the constructor to properly assign the `registry` variable using the `_registry` parameter instead of `msg.sender`:

```
// Corrected constructor
constructor(address _registry) ERC721("DonationReceipt", "DRC") {
    registry = CharityRegistry(_registry); // Correct assignment
    owner = msg.sender;
    tokenCounter = 0;
}
```

2. By using `_registry`, the `GivingThanks` contract correctly references the deployed `CharityRegistry` contract, allowing charity verification to function as intended.

## [M-2] Reentrancy Vulnerability in `GivingThanks::donate`

**Description:** The `donate` function in the `GivingThanks.sol` contract is vulnerable to a reentrancy attack due to the external call to `charity.call{value: msg.value}("")`. This call is made before the state variables are updated, enabling a malicious charity contract to re-enter the function and perform additional malicious actions before the state is fully updated.

**Impact:** The ability for an external contract to re-enter the `donate` function can result in a reentrancy attack. This could allow a malicious contract to repeatedly call `donate`, draining the contract's funds and potentially minting multiple tokens for the same donation. This poses a serious risk to the integrity and security of the contract and its funds.

**Proof of Concept:**

1. The vulnerability arises because the contract first sends Ether to the `charity` address (which could be a contract) and then performs state-changing actions such as minting a token and updating the metadata. If the `charity` contract is malicious and contains a fallback function, it could re-enter the `donate` function before the state changes, allowing an attacker to exploit the situation (e.g., by calling the `donate` function repeatedly and draining Ether).

Here is the vulnerable part of the code:

```
function donate(address charity) public payable {
    require(registry.isVerified(charity), "Charity not verified");

    // External call to charity before state changes
```

```
    (bool sent, ) = charity.call{value: msg.value}("");
    require(sent, "Failed to send Ether");

    // State changes occur after the external call
    _mint(msg.sender, tokenCounter);
    string memory uri = _createTokenURI(
        msg.sender,
        block.timestamp,
        msg.value
    );
    _setTokenURI(tokenCounter, uri);
    tokenCounter += 1;
}
```

**Recommended Mitigation:**

1. Use the Checks-Effects-Interactions Pattern: Move all state changes (such as minting the token, setting the URI, and updating `tokenCounter`) before the external call to ensure that the contract state is updated before the external contract can interfere.
2. Use a `ReentrancyGuard` Modifier: Implement the `nonReentrant` modifier from OpenZeppelin's `ReentrancyGuard`to prevent multiple calls to the `donate` function during execution.

## [M-3] Missing Registry Removal Functionality in `CharityRegistry.sol`

**Description:** The contract lacks functionality to remove registered or verified charities, making it impossible to handle compromised or defunct charities.

**Impact:**

1. No way to remove malicious charities
2. Cannot update registry for defunct organizations
3. Permanent storage bloat
4. No ability to handle compromised charity addresses

**Recommended Mitigation:**

1. Add functions to remove charities with appropriate access controls.

```
event CharityRemoved(address indexed charity);

function removeCharity(address charity) public onlyAdmin {
    require(registeredCharities[charity], "Charity not registered");
    registeredCharities[charity] = false;
    verifiedCharities[charity] = false;
    emit CharityRemoved(charity);
}
```

# Low Risk Findings

## [L-1] Users can get NFTs sending zero transactions

**Description:** The donate function lacks a `check` for msg.value, allowing users to obtain an NFT by sending a transaction with `zero` ether.

**Impact:** Without a check for `msg.value`, users can call the function `donate` with zero ether and still `receive` an NFT.

**Proof of Concept:**

1. Add this `code` to tests , but as there is a bug in the constructor of the `GivingThanks`contract , fix it before running tests .

```
 constructor(address _registry) ERC721("DonationReceipt", "DRC") {
+       registry = CharityRegistry(_registry);
-       registry = CharityRegistry(_msg.sender);
        owner = msg.sender;
        tokenCounter = 0;
    }
```

```
function testZeroDonate() public {
        // Check initial token counter
        uint256 initialTokenCounter = charityContract.tokenCounter();

        // Donor donates to the charity zero Ether
        vm.prank(donor);
        charityContract.donate{value: 0}(charity);

        // Check that the NFT was minted
        uint256 newTokenCounter = charityContract.tokenCounter();
        assertEq(newTokenCounter, initialTokenCounter + 1);

        // Verify ownership of the NFT
        address ownerOfToken = charityContract.ownerOf(initialTokenCounter);
        assertEq(ownerOfToken, donor);

        // Verify that the zero donation was sent to the charity
        uint256 charityBalance = charity.balance;
        assertEq(charityBalance, 0);
    }
```

**Recommended Mitigation:**

1. To address this vulnerability, you should add a check for msg.value in the donate function.

```
function donate(address charity) public payable {
  require(msg.value > 0,"Amount is 0");
```

# [L-2] GivingThanks.donate(): Minting unlimited number of NFT tokens by charity

**Description:** A verified charity can use their donated funds to donate to themselves and receive `DonationReceipt` tokens in an almost unlimited quantity - limited by the sum of their gas costs vs balance.

**Impact:** The purpose of `DonationReceipt` tokens is to identify donors. However, allowing charities to make donations themselves makes the `DonationReceipt` obsolete. You don't really know if the address on the receipt actually donated eth to a charity.

**Proof of Concept:**

1. There is no check that prevents a charity from donating to itself. Therefore, each charity can mint a `DonationReceipt` token as many times as it wants.

```
function testCharityCanMintNFT() public {
        vm.deal(charity, 20 ether);

        uint256 balanceBefore = charity.balance;
        uint256 initialTokenCounter = charityContract.tokenCounter();

        assertEq(initialTokenCounter, 0);

        vm.prank(charity);
        charityContract.donate{value: balanceBefore}(charity);

        uint256 balanceAfter = charity.balance;

        //gas neglected
        assert(balanceBefore == balanceAfter);
        assertEq(charityContract.tokenCounter(), initialTokenCounter + 1);
        assertEq(charityContract.ownerOf(initialTokenCounter), charity);
    }
```

**Recommended Mitigation:**

1. Restrict any registered/verified charity from making donations
2. Restrict any verified charity from making donations to itself.