i) Anyone can call function `donate()` in `GivingThanks.sol` and Mint NFT with zero donations.

Summary

The `donate` function in `GivingThanks.sol` lacks a minimum donation check. Anyone can call `donate()` and mint NFT by donating 0 ETH. This could lead to a situation where a donor mint NFT without making any contributions to a charity.

Vulnerability Details

Issue

In the `donate` function, there is no requirement for a minimum amount of ETH to be sent along with the transaction. This allows any caller to donate 0 ETH and still successfully mint an NFT, as shown in the code snippet:

```
function donate(address charity) public payable {
    require(registry.isVerified(charity), "Charity not verified");
    (bool sent, ) = charity.call{value: msg.value}("");
    require(sent, "Failed to send Ether");

    _mint(msg.sender, tokenCounter);

    string memory uri = _createTokenURI(
        msg.sender,
        block.timestamp,
        msg.value
    );
    _setTokenURI(tokenCounter, uri);

    tokenCounter += 1;
}
```

The function successfully completes without any check to enforce that `msg.value` is greater than zero. As a result, users can call `donate()` with `msg.value = 0` and still receive an NFT.

The absence of a minimum ETH amount check allows the function to be executed with 0 ETH, minting an NFT.

Impact

- Users can mint unlimited NFTs without making any actual donations to verified charities.

Tools Used

- Manual code review.

Recommendations

Add a Minimum Donation Check: Modify the `donate` function to enforce a minimum ETH donation amount by adding a requirement statement:

```
require(msg.value > 0 ether, "Donation should be greater than 0 ETH");
```

ii)Lack of access control in function `updateRegistrty()` in `GivingThanks.sol` allows anyone to update the address of the registry and change its logic by pointing the registry address to an arbitrary contract that steals donor funds

Summary

function `updateRegistry()` has no access controls allowing anyone to change the address of registry and change the logic of `CharityRegistry.sol`. An attacker can deploy an arbitrary contract with different logic, i.e changing function `isVerified` whereby the attacker transfers the donated ether to themselves.

Vulnerability Details

```
function updateRegistry(address _registry) public {
        registry = CharityRegistry(_registry);
    }
```

This function allows anyone to call it and update `registry` to an arbitrary contract which could literally do anything like stealing donors funds.

Impact

Stealing of donors funds.

Tools Used

Manual review.

Recommendations

Add access control mechanism(s) to `function updateRegistry()`, e.g a modifier.

```
modifier onlyOwner(){
        require(msg.sender == owner, "Error!");
        _;
    }
function updateRegistry(address _registry) public onlyOwner{
        registry = CharityRegistry(_registry);
    }
```

iii)function `donate()` in `GivingThanks.sol` reads the wrong mapping allowing non-verified charities to receive donations

Summary

The `donate` function in `GivingThanks.sol` fails to properly check the charity's verification status before allowing a donation. Instead of ensuring that the charity is verified, it mistakenly checks only if the charity is registered in the `CharityRegistry.sol` contract. As a result, unverified charities can receive donations when they shouldn't.

Vulnerability Details

Issue

In `GivingThanks.sol`, the `donate` function has the following code:

```
require(registry.isVerified(charity), "Charity not verified");
```

However, the `isVerified` function in the `CharityRegistry` contract returns the status of `registeredCharities[charity]`, not `verifiedCharities[charity]`. This allows any registered charity to receive donations, regardless of verification status. The intended functionality was for `isVerified` to check the `verifiedCharities` mapping, which actually confirms a charity's verification.

Root Cause

The `isVerified` function currently only checks if a charity is registered by returning `registeredCharities[charity]`. It does not verify whether the charity is also verified by checking `verifiedCharities[charity]`.

Impact

- Unverified charities can bypass the verification process and receive donations directly.

Tools Used

- Manual code review.

Recommendations

1. **Update the `isVerified` function** in `CharityRegistry` to check the `verifiedCharities` mapping directly, ensuring only verified charities receive donations:

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

iv)Unsafe low-level call in function `donate()` in `GivingThanks.sol` no checks that the externally called charity address does exist.

Summary

The `donate` function in `GivingThanks.sol` performs a low-level `.call` to transfer funds to charity, without checking whether the address is associated with a deployed contract. According to the Solidity documentation, the EVM considers a low-level call to a non-existing contract to not revert but instead return `true`, even though the funds may be sent to an invalid address. This can lead to donations being lost or misdirected if the charity address is not properly validated.

Vulnerability Details

Issue

The `donate` function uses a low-level call to transfer Ether:

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

According to the Solidity documentation, when using low-level calls, the EVM does not automatically verify that the target address is associated with a contract. This can lead to silent failures where calls to non-existing contracts succeed even though no code exists at the target address. Normally, Solidity uses `extcodesize` to ensure the called address has code, but this check is bypassed when using low-level calls. Here, function `donate()` checks mapping `registeredCharities[charity]` by calling function `isVerified()` in `CharityRegistry.sol`, which does not guarantee the presence of an actual deployed contract.

Solidity Documentation Reference:External Function Calls

In this context, low-level calls bypass this validation, meaning that donations could be sent to non-existing address.

Impact

- **Loss of Funds**: If a non-existing contract charity address is used, funds may be sent to an unintended address without error, leading to permanent loss.

Tools Used

- Solidity documentation and manual code review.

Recommendations

If charity is not an Externally Owned Address implement contract existence checks before transferring funds, use Solidity's `Address.isContract` function from the OpenZeppelin library (or a similar method) to verify that the `charity` address has deployed code.

```
require(Address.isContract(charity), "Invalid charity address");
```

v)Improper Initialization of External Contract `CharityRegistry.sol` in Constructor of `GivingThanks.sol`

Summary

The constructor of the `GivingThanks.sol` contract initializes the `registry` variable, intended to reference an external `CharityRegistry` contract, to `msg.sender` instead of the `_registry` parameter. This creates a critical flaw in which `GivingThanks.sol` contract cannot access the actual `CharityRegistry` instance for operations such as verifying donations. Instead, the contract mistakenly sets the `registry` to the deployer of the

`GivingThanks` contract, leading to erroneous interactions.

Vulnerability Details

Issue

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

- In the above constructor of `GivingThanks.sol`, the line `registry = CharityRegistry(msg.sender);` assigns `msg.sender` (the address deploying `GivingThanks`) to the `registry` variable rather than the intended `_registry` parameter, which is supposed to be the address of a valid `CharityRegistry` contract instance.
- The `GivingThanks` contract will treat the deployer's address as the `CharityRegistry` instance, causing functions such as `donate` to fail, as the deployer's address cannot validate charity addresses or provide any registry functionality.
- It breaks the purpose of the `GivingThanks` contract, which relies on `CharityRegistry` contract to ensure only registered and verified charities receive donations.

Impact

- Donations will fail because the contract cannot verify charity addresses without a proper `CharityRegistry` instance, leading to failed transactions in the `donate` function.

Tools Used

- Manual Code Review

Recommendations

- Update the constructor to use the `_registry` parameter:

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

ISSUES YET TO ESCALATE

Difference in `block.timestamp` in different EVM-Compatible chains

Description

"block.timestamp" method

- Difference: There are differences in the results obtained based on block.timestamp on Ethereum and Arbitrum. On Ethereum, it returns the timestamp of the current block, while on Arbitrum, it retrieves the timestamp recorded by the Sequencer. Security concerns: If the Sequencer reads the timestamp too frequently on Arbitrum, it may result in different blocks having the same timestamp. This can lead to the following security issues: Front-Running Attacks: Blocks with the same timestamp can cause uncertainty in the order of transactions. Attackers can exploit this uncertainty to anticipate the results of other transactions and execute operations that are advantageous to them. Time-sensitive Contract Issues: If smart contracts rely on timestamps to perform certain operations, such as restricting access or calculating time-sensitive rewards, blocks with the same timestamp can cause contract behavior to deviate from expectations. Timestamp Dependency Vulnerabilities: If certain contracts or systems use timestamps for calculations or decisions, blocks with the same timestamp can cause errors in these calculations or decisions, leading to other security issues. Example: In an Arbitrum smart contract, there is code that relies on block.timestamp and block.number to generate random numbers. Due to the unchanging block number and timestamp within a minute, the pseudo-random numbers generated during this time period will be the same.

```
function getRandomNumber() public view returns (uint) {
    return uint(keccak256(abi.encodePacked(block.timestamp,
block.number)));
}
```

- **Suggestion: Using Chainlink VRF to obtain secure random numbers.**

Reference: Salus Gitbool