First Flight N28 GivingThanks

Prepared by: 0xKoiner

First Flight N28 GivingThanks Audit Report

Prepared by: 0xKoiner Lead Auditors: 0xKoiner

- [0xKoiner] (#https://twitter.com/0xKoiner)

Assisting Auditors:

- None

Table of contents

▶ Details
See table

About OxKoiner

Hi, I'm OxKoiner, a developer and smart contract auditor based in sunny Barcelona. My journey in the crypto industry started back in 2016, and it's been an exciting ride ever since. With a background in Computer Science from the Technion (Israel Institute of Technology), I've built up a versatile skill set that spans Python development and smart contract security.

My Experience Python Development I have a solid background in Python, focusing on backend development, software automation, testing, and Web3 libraries. I enjoy building tools and solutions that help simplify complex problems, and I'm always looking for new ways to streamline workflows and processes.

Solidity & Smart Contract Development I'm a junior Solidity developer with a growing passion for low-level aspects of Ethereum development, including OpCodes, Huff, and Assembly. I love diving into the details of smart contract design, aiming to write clean, efficient, and secure code. I'm still learning every day, but I enjoy the challenge of constantly improving.

Auditing & Security I've completed several courses focused on smart contract audits and enjoy the process of analyzing code for potential vulnerabilities. My toolkit includes:

Foundry & HardHat: For testing and development. Slither & Aderyn: For static analysis and finding common issues. Certora: For formal verification to ensure contract safety. I take a careful, methodical approach when auditing, trying to catch even the smallest issues that could become big problems. I believe every audit is a chance to learn and help make the Web3 space a bit safer.

My Approach I'm motivated by a genuine interest in blockchain technology and its potential to bring about positive change. I'm not an expert in everything, but I'm always eager to learn and collaborate with others. If you're working on an interesting project or need a fresh set of eyes for an audit, I'd love to connect and see how I can help.

Let's build something great together!

Disclaimer

The OxKoiner team has made every effort to identify potential vulnerabilities within the time allocated for this audit. However, we do not assume responsibility for the findings or any issues that may arise after the audit. This security audit is not an endorsement of the project’s business model, product, or team. The audit was time-limited and focused exclusively on assessing the security of the Solidity code implementation. It is strongly recommended that additional testing and security measures be conducted by the project team.

Risk Classification

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

Scope

```
All Contracts in `src` are in scope.

├── src
│   ├── CharityRegistry.sol
│   └── GivingThanks.sol
```

Protocol Summary

GivingThanks is a decentralized platform that embodies the spirit of Thanksgiving by enabling donors to contribute Ether to registered and verified charitable causes. Charities can register themselves, and upon verification by the trusted admin, they become eligible to receive donations from generous participants. When donors make a donation, they receive a unique NFT as a donation receipt, commemorating their contribution. The NFT's metadata includes the donor's address, the date of the donation, and the amount donated.

Roles

- Admin (Trusted) - Can verify registered charities.
- Charities - Can register to receive donations once verified.
- Donors - Can donate Ether to verified charities and receive a donation receipt NFT.

Executive Summary

Issues found

Severity	Number of issues found

Severity	Number of issues found
High	4
Medium	5
Low	7
Total	16

Findings

High

[H-1] Constructor Conflict in GivingThanks.sol and CharityRegistry.sol

_Submitted by 0xKoiner

Summary

A deployment conflict occurs due to a mismatch in the expected address for the `CharityRegistry` contract instance in `GivingThanks.sol`. This prevents the `GivingThanks` contract from deploying correctly, as the `_registry` parameter is not utilized as intended in the constructor.

Vulnerability Details

In `GivingThanks.sol`, the constructor receives an address parameter `_registry` intended to reference an existing `CharityRegistry` contract instance. However, the `_registry` address is not correctly utilized in the contract, which results in a failure during deployment. This makes the `GivingThanks` contract unusable as it cannot be deployed to the blockchain without addressing this issue.

Impact

High Impact: The deployment failure blocks the `GivingThanks` contract from functioning, preventing any interactions on-chain. This issue effectively halts the contract's deployment and intended operation.

Tools Used

- Manual code review
- **Foundry** (for compilation and testing)

##POC

```
forge compile
```

Compiler Output

``` ## POC 2

```
Compiler run successful with warnings:
Warning (5667): Unused function parameter. Remove or comment out the
variable name to silence this warning.
--> src/GivingThanks.sol:15:17:
   |
15 | constructor(address _registry) ERC721("DonationReceipt", "DRC") {
   | ^^^^^^^^^^^^^^^^
```

### Recommendations

To resolve the issue, modify the constructor in GivingThanks.sol to properly utilize the _registry parameter, as shown below:

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

This change will correctly assign registry to an instance of the CharityRegistry contract, allowing the GivingThankscontract to interact with it as intended.

[H-2] updateRegistry() Function in GivingThanks.sol Allows Unrestricted Access to Change CharityRegistry Address

_Submitted by 0xKoiner

Summary

The updateRegistry function in **GivingThanks.sol** does not have any access control, allowing anyone (including unauthorized users) to change the registry contract address. This opens up the possibility for an attacker to replace the legitimate CharityRegistry contract with a malicious contract, potentially bypassing the charity verification logic and affecting the contract's integrity. The test fails because the attacker is able to update the registry contract address successfully.

Vulnerability Details

Affected Contract:

- GivingThanks.sol
```

Affected Function:

- `updateRegistry(address _registry) public`

Issue: In the current implementation of the `updateRegistry` function, there is no restriction on who can call this function. As a result, an attacker can change the registry address to any address of their choice, potentially a malicious contract that manipulates the charity verification logic.

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

Vulnerability Details

Severity: High

The lack of access control in `updateRegistry` means that:

- **Unauthorized Users Can Modify Critical Logic:** Any user can replace the `CharityRegistry` contract with another one, including malicious ones.
- **Bypassing Charity Verification:** If an attacker sets the registry address to a malicious contract, subsequent donations or verifications could be processed by the attacker's contract, effectively bypassing any verification checks or altering contract state.
- **Potential Malicious Exploitation:** The attacker could alter the registry logic, enabling them to manipulate donations or other key processes in the GivingThanks contract, leading to potential financial loss or unauthorized actions

Proof of Code: **Test Code Showing Unrestricted Access:**

```
function testGivingThanksUpdateRegistryNotProtected() public {
    address oldRegistryAddress = address(charityContract.registry());

    address attackerAddress = makeAddr("attackerAddress");
    vm.startPrank(attackerAddress);
    charityContract.updateRegistry(attackerAddress); // Attacker updates
the registry
    vm.stopPrank();

    address updatedRegistryAddress = address(charityContract.registry());

    assertNotEq(oldRegistryAddress, updatedRegistryAddress); // Test fails
as the registry is updated by the attacker
}
```

In this test case:

- The attacker successfully updates the registry address, which is not expected since the test assumes the registry should remain unchanged.
- This happens because the `updateRegistry` function lacks any access control to restrict who can call it.

Tools Used

- Manual code review
- Foundry

Recommendations

To secure the contract and prevent unauthorized users from updating the registry contract, you should add an access control modifier to the `updateRegistry` function. For example, you could restrict access to the contract's owner or a trusted entity using the `onlyOwner` modifier (or any other role-based access control mechanism depending on your requirements):

```
modifier onlyOwner() {
    require(msg.sender == owner, "Not contract owner");
    _;
}

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

- **`onlyOwner`**: This modifier ensures that only the owner of the contract (or an authorized address) can update the registry contract address. The contract will no longer be vulnerable to unauthorized changes.

By adding access control, only trusted parties (like the contract owner) can modify the registry address, securing the GivingThanks contract from potential exploitation.

[H-3] Reentrancy Vulnerability in donate Function of GivingThanks.sol

_Submitted by 0xKoiner

Summary

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.

Vulnerability Details

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).

```
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;
}
```

Tools Used

High 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.

Tools Used

- Manual code review
- **Slither** (for vulnerability detection)

```
INFO:Detectors:
Reentrancy in GivingThanks.donate(address) (src/GivingThanks.sol#21-37):
        External calls:
        - (sent) = charity.call{value: msg.value}()
(src/GivingThanks.sol#23)
        State variables written after the call(s):
        - _mint(msg.sender,tokenCounter) (src/GivingThanks.sol#26)
                - _balances[from] -= 1 (lib/openzeppelin-
contracts/contracts/token/ERC721/ERC721.sol#256)
                - _balances[to] += 1 (lib/openzeppelin-
contracts/contracts/token/ERC721/ERC721.sol#262)
        - _mint(msg.sender,tokenCounter) (src/GivingThanks.sol#26)
                - _owners[tokenId] = to (lib/openzeppelin-
contracts/contracts/token/ERC721/ERC721.sol#266)
        - _mint(msg.sender,tokenCounter) (src/GivingThanks.sol#26)
                - _tokenApprovals[tokenId] = to (lib/openzeppelin-
contracts/contracts/token/ERC721/ERC721.sol#424)
        - _setTokenURI(tokenCounter,uri) (src/GivingThanks.sol#34)
                - _tokenURIs[tokenId] = _tokenURI (lib/openzeppelin-
contracts/contracts/token/ERC721/extensions/ERC721URIStorage.sol#58)
        - tokenCounter += 1 (src/GivingThanks.sol#36)
Reference: https://github.com/crytic/slither/wiki/Detector-
Documentation#reentrancy-vulnerabilities-2
INFO:Detectors:
Reentrancy in GivingThanks.donate(address) (src/GivingThanks.sol#21-37):
        External calls:
        - (sent) = charity.call{value: msg.value}()
(src/GivingThanks.sol#23)
        Event emitted after the call(s):
        - Approval(owner,to,tokenId) (lib/openzeppelin-
contracts/contracts/token/ERC721/ERC721.sol#420)
                - _mint(msg.sender,tokenCounter) (src/GivingThanks.sol#26)
        - MetadataUpdate(tokenId) (lib/openzeppelin-
contracts/contracts/token/ERC721/extensions/ERC721URIStorage.sol#59)
                - _setTokenURI(tokenCounter,uri) (src/GivingThanks.sol#34)
        - Transfer(from,to,tokenId) (lib/openzeppelin-
contracts/contracts/token/ERC721/ERC721.sol#268)
                - _mint(msg.sender,tokenCounter) (src/GivingThanks.sol#26)
Reference: https://github.com/crytic/slither/wiki/Detector-
Documentation#reentrancy-vulnerabilities-3
```

Recommendations

To mitigate the reentrancy vulnerability, we recommend the following solutions:

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.

Solution: Updated Code:

```
import "@openzeppelin/contracts/security/ReentrancyGuard.sol";

contract GivingThanks is ERC721URIStorage, ReentrancyGuard {
    // ... rest of the code remains unchanged

    function donate(address charity) public payable nonReentrant {
        require(registry.isVerified(charity), "Charity not verified");
        // First, perform all internal state changes
        _mint(msg.sender, tokenCounter);
        string memory uri = _createTokenURI(
            msg.sender,
            block.timestamp,
            msg.value
        );
        _setTokenURI(tokenCounter, uri);
        tokenCounter += 1;

        // After state changes, execute external call
        (bool sent, ) = charity.call{value: msg.value}("");
        require(sent, "Failed to send Ether");
    }
}
```

Explanation of the Fixes

1. **`ReentrancyGuard`**: The `nonReentrant` modifier prevents reentrancy by blocking any re-entry into the `donate`function. This ensures that once the function is entered, no other calls can interrupt it until it completes.
2. **Checks-Effects-Interactions Pattern**: All internal state changes (such as minting the token, setting the token URI, and incrementing the `tokenCounter`) are now done before the external call to `charity`. This prevents any reentrancy issues, as the contract's state is fully updated before the external call is made.

Conclusion

By applying both the `ReentrancyGuard` and the Checks-Effects-Interactions pattern, the reentrancy vulnerability in the `donate` function is mitigated. These changes secure the contract by ensuring that external calls cannot interfere with the contract's state, protecting it from malicious reentrancy attacks.

[H-4] Broken Verification Logic in isVerified Function of CharityRegistry.sol

_Submitted by 0xKoiner

Summary

The `isVerified` function in `CharityRegistry.sol` mistakenly checks the `registeredCharities` mapping instead of the `verifiedCharities` mapping. As a result, the `GivingThanks` contract allows donations to any registered charity regardless of whether it has been verified, bypassing the intended verification requirement.

Vulnerability Details

The function `isVerified` in `CharityRegistry.sol` is designed to validate if a charity is verified before allowing a donation. However, instead of checking the `verifiedCharities` mapping, which stores verification status, it incorrectly references `registeredCharities`, a separate mapping for registration status. This oversight breaks the workflow, permitting donations to any registered charity, verified or not.

Vulnerable Code

The error is found in the `isVerified` function within `CharityRegistry.sol`:

```
function isVerified(address charity) public view returns (bool) {
    return registeredCharities[charity]; // Incorrect mapping used here
}
```

This should reference `verifiedCharities` instead:

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

Impact

This vulnerability allows donations to any charity that is merely registered without being verified. This opens the contract to potential misuse or fraudulent donations to unverified charities, undermining the integrity of the donation process and the intent of the verification system. Given the implications for donor trust and potential misuse of funds, this bug can be categorized as a HIGH-LEVEL SEVERITY ISSUE.

Tools Used

- Manual code inspection
- `forge test` output and trace analysis

Recommendations

To resolve this issue, update the `isVerified` function to check the correct mapping (`verifiedCharities`). This fix aligns the logic with the intended functionality, ensuring donations are only permitted for verified charities.

Solution

In `CharityRegistry.sol`, modify the `isVerified` function as follows:

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

This change enforces the intended verification check, preventing donations to unverified charities and restoring the expected workflow of the contract.

Recommendations

Notice to Command:

During testing, the function `testCannotDonateToUnverifiedCharity` failed as it allowed donations to an unverified charity, indicating that verification was not enforced as intended. Below is the command run and the detailed output:

Command and Test Output:

1. Initial command run:

```
forge test
```

Output:

```
Suite result: FAILED. 2 passed; 1 failed; 0 skipped; finished in 30.75ms
(27.43ms CPU time)
[FAIL: next call did not revert as expected]
testCannotDonateToUnverifiedCharity() (gas: 307608)
```

to gain more detail, the following was run:

```
forge test --mt testCannotDonateToUnverifiedCharity -vvvv
```

Output:

```
[307608] GivingThanksTest::testCannotDonateToUnverifiedCharity()
  ├─ [0] VM::prank(Identity: [0x0000000000000000000000000000000000000004])
  ├─ [22464] CharityRegistry::registerCharity(Identity:
[0x0000000000000000000000000000000000000004])
  ├─ [0] VM::deal(donor: [0xBd4D11D884c3c7bF373b013AC52fd99f9DD86D0A],
10000000000000000000 [1e19])
  ├─ [0] VM::prank(donor: [0xBd4D11D884c3c7bF373b013AC52fd99f9DD86D0A])
  ├─ [0] VM::expectRevert(custom error 0xf4844814)
  ├─ [262040] GivingThanks::donate{value: 1000000000000000000}(Identity:
[0x0000000000000000000000000000000000000004])
  │    ├─ CharityRegistry::isVerified(...) // incorrectly returns true
  │    └─ ← [Stop]
```

From this output, it's clear that `CharityRegistry::isVerified` returned `true` for an unverified charity, allowing the donation. This confirmed that the test failed due to the incorrect mapping reference.

Medium

[M-1]. Missing receive() and fallback() Functions in GivingThanks.sol to Handle Incoming Ether Transfers

_Submitted by 0xKoiner

Summary

The `GivingThanks.sol` contract currently lacks the ability to accept Ether sent directly to the contract. Without a `receive()` or `fallback()` function, any direct transfer of Ether to the contract will revert the transaction. Adding a `receive()` function will allow the contract to accept Ether sent directly, ensuring that transactions don't fail when Ether is mistakenly sent.

Vulnerability Details

Low. Only suggesion for better practice

Impact

The impact is moderate but could affect the usability of the contract:

- **Failure to handle Ether transfers**: Users who send Ether directly to the contract (either accidentally or intentionally) will cause the transaction to revert.
- **Loss of Ether**: If the contract cannot accept Ether, users might lose funds if they mistakenly send Ether to the contract.

Tools Used

Manual review of the contract code

Recommendations

Solution: Add `receive()` and/or `fallback()` function

To address this, we recommend adding a `receive()` function to handle direct Ether transfers. Optionally, you can add a `fallback()` function to handle transfers with data attached.

Changes to `GivingThanks.sol`:

1. **Add a `receive()` function** to accept Ether with no data attached.
2. **Optionally** add a `fallback()` function to handle any transfers that may include data.

Modified Contract Example:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.19;

import { CharityRegistry } from "./CharityRegistry.sol";
import { ERC721URIStorage } from
"@openzeppelin/contracts/token/ERC721/extensions/ERC721URIStorage.sol";
import { Strings } from "@openzeppelin/contracts/utils/Strings.sol";
import { Base64 } from "@openzeppelin/contracts/utils/Base64.sol";

contract GivingThanks is ERC721URIStorage {
    // Receive Ether
    receive() external payable {
        // Optionally, handle incoming Ether with some logic
        // You can log events, forward Ether to other addresses, or simply
accept the Ether
    }

    // Fallback function (optional)
    fallback() external payable {
        // Handle Ether with data if necessary or log the event
    }
}
```

[M-2] Lack of Zero Address Check in Critical Functions of GivingThanks.sol and CharityRegistry.sol

_Submitted by 0xKoiner

Summary

The functions in both `GivingThanks.sol` and `CharityRegistry.sol` lack checks to prevent the use of `address(0)`. This can lead to critical issues, such as accidentally setting important addresses to `address(0)` or sending Ether to the zero address. It is a best practice to ensure that addresses passed to functions are not `address(0)`.

Vulnerability details

Missing Zero Address Check:
The functions such as `updateRegistry`, `donate`, `changeAdmin`, and `verifyCharity` do not verify if the passed address is `address(0)`. Using `address(0)` could cause unexpected behaviors, such as sending Ether to an invalid address or making an address such as the admin address invalid (`address(0)`).

Impact

Failure to validate against `address(0)` could allow for critical issues:

- If `address(0)` is used for the `admin` address in `CharityRegistry.sol`, this would break the ability to perform administrative actions.
- If `address(0)` is passed as the charity address in `donate`, the contract would attempt to send Ether to the zero address, effectively losing the Ether.
- The same applies to setting the registry address or charity verification, leading to loss of functionality.

Tools Used

Manual Code Review

Recommended mitigation

1. **Create a Modifier to Check for Zero Address:**
   Implement a `nonZeroAddress` modifier to check if the provided address is `address(0)` before executing the function logic.
2. **Apply the Modifier to Critical Functions:**
   The `nonZeroAddress` modifier should be applied to all functions that involve updating addresses, such as `updateRegistry`, `donate`, `changeAdmin`, and any other relevant functions that deal with addresses.
3. **Update Constructors and Functions:**
   Apply the check in the constructor to ensure the provided addresses are not `address(0)` during initialization.

Solution:

1. **Create the `nonZeroAddress` Modifier:**

```
modifier nonZeroAddress(address _addr) {
    require(_addr != address(0), "Address cannot be zero");
    _;
}
```

** Apply the Modifier to Functions:**

Updated `GivingThanks.sol`:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import { CharityRegistry } from "./CharityRegistry.sol";
import { ERC721URIStorage } from
"@openzeppelin/contracts/token/ERC721/extensions/ERC721URIStorage.sol";
import { Ownable } from "@openzeppelin/contracts/access/Ownable.sol";
import { Strings } from "@openzeppelin/contracts/utils/Strings.sol";
import { Base64 } from "@openzeppelin/contracts/utils/Base64.sol";

contract GivingThanks is ERC721URIStorage {
    CharityRegistry public registry;
    uint256 public s_tokenCounter;
    address public immutable i_owner;

    modifier nonZeroAddress(address _addr) {
        require(_addr != address(0), "Address cannot be zero");
        _;
    }

    constructor(address _registry) ERC721("DonationReceipt", "DRC") {
        require(_registry != address(0), "Registry address cannot be
zero");
        registry = CharityRegistry(_registry);
        i_owner = msg.sender;
        s_tokenCounter = 0;
    }

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

        _mint(msg.sender, s_tokenCounter);

        // Create metadata for the tokenURI
        string memory uri = _createTokenURI(
            msg.sender,
            block.timestamp,
            msg.value
        );
        _setTokenURI(s_tokenCounter, uri);

        s_tokenCounter += 1;
    }

    function _createTokenURI(
        address donor,
        uint256 date,
        uint256 amount
    ) internal pure returns (string memory) {
        // Create JSON metadata
        string memory json = string(
            abi.encodePacked(
                '{"donor":"',
                Strings.toHexString(uint160(donor), 20),
                '","date":"',
                Strings.toString(date),
                '","amount":"',
                Strings.toString(amount),
                '"}'
            )
        );

        // Encode in base64 using OpenZeppelin's Base64 library
        string memory base64Json = Base64.encode(bytes(json));

        // Return the data URL
        return
            string(
                abi.encodePacked("data:application/json;base64,",
base64Json)
            );
    }

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

Updated `CharityRegistry.sol`:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract CharityRegistry {
    address public admin;
    mapping(address => bool) public verifiedCharities;
    mapping(address => bool) public registeredCharities;

    modifier nonZeroAddress(address _addr) {
        require(_addr != address(0), "Address cannot be zero");
        _;
    }

    constructor() {
        admin = msg.sender;
    }

    function changeAdmin(address newAdmin) public nonZeroAddress(newAdmin)
{
        require(msg.sender == admin, "Only admin can change admin");
        admin = newAdmin;
    }

    function verifyCharity(address charity) public nonZeroAddress(charity)
{
        require(msg.sender == admin, "Only admin can verify");
        require(registeredCharities[charity], "Charity not registered");
        verifiedCharities[charity] = true;
    }

    function registerCharity(address charity) public
nonZeroAddress(charity) {
        registeredCharities[charity] = true;
    }
}
```

[M-3] Incorrect Import Path in GivingThanks.sol or foundry.toml incorrect remmaping

_Submitted by 0xKoiner

Summary

In the `GivingThanks.sol` file, there is an incorrect import path for the @openzeppelin contract from OpenZeppelin. The import path in the contract should align with the one used in the `foundry.toml` configuration, which remaps the OpenZeppelin contracts directory. This discrepancy could lead to import errors or compilation issues.

Vulnerability Details

The current import statement in `GivingThanks.sol`:

```
import "@openzeppelin/contracts/access/Ownable.sol";
import
"@openzeppelin/contracts/token/ERC721/extensions/ERC721URIStorage.sol";
import "@openzeppelin/contracts/utils/Strings.sol";
import "@openzeppelin/contracts/utils/Base64.sol";
```

does not align with the remapping configuration in `foundry.toml`:

```
remappings = [
  "openzeppelin-contracts/=lib/openzeppelin-contracts/",
  "forge-std/=lib/forge-std/src/",
];
```

As per the remappings, OpenZeppelin contracts should be imported as:

```
import "@openzeppelin-contracts/contracts/access/Ownable.sol";
import "@penzeppelin-
contracts/contracts/token/ERC721/extensions/ERC721URIStorage.sol";
import "@penzeppelin-contracts/contracts/utils/Strings.sol";
import "@penzeppelin-contracts/contracts/utils/Base64.sol";
```

This mismatch between the remapping configuration in `foundry.toml` and the import statement in the contract can cause issues during compilation. Specifically, Foundry may not be able to locate the correct file without the proper remapping.

Impact

- **Compilation failure**: The incorrect import path could lead to an error during the compilation of the contract if the remapping is not properly resolved.
- **Inconsistency**: This creates an inconsistency between the remapping configuration and the actual import path used in the code. It could confuse developers and lead to unexpected behaviors in different environments.
- **Gas costs and execution failures**: If the remapping is ignored and the contract doesn't find the right library, this could cause the contract to fail execution or result in a mismatch of expected behavior.

Tools Used

Manual Review, Foundry

Recommendations

1. **Correct the import path**: Update the import statements in `GivingThanks.sol` to match the remapped paths defined in `foundry.toml`.

   Correct the import in `GivingThanks.sol` from:

```
- import "@openzeppelin/contracts/access/Ownable.sol";
- import
"@openzeppelin/contracts/token/ERC721/extensions/ERC721URIStorage.sol";
- import "@openzeppelin/contracts/utils/Strings.sol";
- import "@openzeppelin/contracts/utils/Base64.sol";
```

to:

```
+ "@openzeppelin/contracts/token/ERC721/extensions/ERC721URIStorage.sol";
+ "@openzeppelin/contracts/access/Ownable.sol";
+ "@openzeppelin/contracts/utils/Strings.sol";
+ "@openzeppelin/contracts/utils/Base64.sol";
```

Verify all import paths: Ensure that all imports in the contract are consistent with the remapping configurations in the `foundry.toml` file.

`foundry.toml`:

```
+remappings = [
+    "@openzeppelin/contracts=lib/openzeppelin-contracts/contracts",
+    "forge-std/=lib/forge-std/src/"
]
```

[M-4] Unsafe NFT Minting Using _mint() Instead of _safeMint() in GivingThanks.sol

_Submitted by 0xKoiner

Summary

The `GivingThanks` contract utilizes the `_mint()` function to mint NFTs directly to the user. This approach does not verify whether the recipient can receive ERC721 tokens, which can lead to NFTs being sent to addresses that do not support ERC721. To avoid this issue, it is recommended to use `_safeMint()` instead, which ensures the recipient address can properly handle ERC721 tokens.

Vulnerability Details

The `donate` function currently uses `_mint()`: tatement in the contract can cause issues during compilation. Specifically, Foundry may not be able to locate the correct file without the proper remapping.

```
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);

    // Create metadata for the tokenURI
    string memory uri = _createTokenURI(
        msg.sender,
        block.timestamp,
        msg.value
    );
    _setTokenURI(tokenCounter, uri);

    tokenCounter += 1;
}
```

Problem: The `_mint()` function directly assigns the NFT to the specified address without checking if the recipient can accept ERC721 tokens. If the recipient is a smart contract that does not implement the `IERC721Receiver` interface, the NFT will be permanently locked in that contract, making it irretrievable.

Impact

- **Permanent Loss of NFTs:** NFTs sent to non-compliant smart contracts may be permanently lost and irretrievable, causing asset loss for users.
- **User Frustration and Financial Loss:** Users could lose valuable NFTs if the recipient address is not compatible with ERC721 tokens.
- **Potential Exploitation:** Malicious actors could exploit this by intentionally providing addresses of contracts that do not support ERC721 tokens to lock up NFTs.

Proof of Concept

The following scenario illustrates the issue:

1. A user donates, and the `donate()` function is called with `_mint()`.
2. The NFT is minted to a contract address that does not implement `IERC721Receiver`.
3. The NFT is permanently locked in the recipient contract and cannot be transferred or recovered.

Tools Used

Manual Review

Recommendations

Replace all instances of `_mint()` with `_safeMint()` in `GivingThanks.sol` to ensure that the recipient address can handle ERC721 tokens properly.

```
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");

    _safeMint(msg.sender, tokenCounter); // Using _safeMint instead of
_mint

    // Create metadata for the tokenURI
    string memory uri = _createTokenURI(
        msg.sender,
        block.timestamp,
        msg.value
    );
    _setTokenURI(tokenCounter, uri);

    tokenCounter += 1;
}
```

[M-5] Lack of Comprehensive Testing Coverage and Additional Tests Added in GivingThanks.t.sol

_Submitted by 0xKoiner

Summary

The **GivingThanks** smart contract previously had limited testing coverage, missing several critical scenarios that could lead to potential issues or unexpected behaviors in production. By adding more comprehensive tests, we improved the contract's robustness and helped identify potential vulnerabilities. This report outlines the importance of these new tests and highlights the improvements made.

Vulnerability Details

The original test suite for `GivingThanks.t.sol` lacked coverage for several critical functions and edge cases, including:

1. **Access Control for updateRegistry**: Tests did not include scenarios for unauthorized access attempts to `updateRegistry`.
2. **Donations to Unverified Charities**: Edge cases were not sufficiently tested for donating to unverified charities.
3. **Token Minting Edge Cases**: The minting process was not tested thoroughly, particularly in cases where the recipient might be a smart contract not supporting ERC721 tokens.
4. **Event Emissions**: There were no tests verifying the correct emission of events, such as `DonateToCharity`.
5. **Address(0) Checks**: Scenarios where invalid addresses (e.g., `address(0)`) were passed to functions were not covered.

Impact

The lack of comprehensive testing could lead to:

- **Unexpected Contract Behavior**: Missed edge cases and incorrect assumptions can cause bugs to surface during deployment or interaction with users.
- **Security Vulnerabilities**: Limited testing increases the risk of undetected issues, including potential exploits related to access control or improper function inputs.
- **Reduced Confidence in Smart Contract**: Incomplete testing reduces the confidence developers and users can have in the reliability of the contract, possibly impacting adoption and usage.

Tools Used

- **Foundry** for running tests (`forge test`)
- **Coverage Analysis** using Foundry's built-in coverage tool (`forge coverage`)

Recommendations

Additional Tests Added Test Breakdown and Explanations:

Detailed Test Functions for `CharityRegistry.sol`

1. `testCharityRegistryChangeAdminWithAdmin`

```
function testCharityRegistryChangeAdminWithAdmin() public {
    address newAdmin = makeAddr("newAdmin");

    vm.startPrank(admin);
    registryContract.changeAdmin(newAdmin);
    vm.stopPrank();
    address updatedAdmin = registryContract.admin();

    assertEq(newAdmin, updatedAdmin);
}
```

- **Purpose**: Verifies that only the current admin can change the admin address.

- **Explanation**:

  - Sets up a new address (`newAdmin`).
  - Impersonates the current admin using `vm.startPrank(admin)`.
  - Calls `changeAdmin` to update the admin address.
  - Asserts that the admin address has been successfully updated to `newAdmin`.

2. `testCharityRegistryChangeAdminWithAdminRevert`

```
function testCharityRegistryChangeAdminWithAdminRevert() public {
    address newAdmin = makeAddr("newAdmin");
    vm.startPrank(donor);
    vm.expectRevert("Only admin can change admin");
    registryContract.changeAdmin(newAdmin);
    vm.stopPrank();
}
```

- **Purpose**: Ensures that a non-admin user cannot change the admin address.
- **Explanation**:
  - Sets up a new address (`newAdmin`).
  - Impersonates a non-admin user (`donor`).
  - Expects the call to `changeAdmin` to revert with the message "Only admin can change admin".

3. `testCharityVerifyCharityAdminWithAdmin`

```
function testCharityVerifyCharityAdminWithAdmin() public {
    address newCharity = makeAddr("newCharity");
    vm.startPrank(newCharity);
    registryContract.registerCharity(newCharity);
    vm.stopPrank();

    vm.startPrank(admin);
    registryContract.verifyCharity(newCharity);
    vm.stopPrank();
    assert(registryContract.isVerified(newCharity));
}
```

- **Purpose**: Confirms that only the admin can verify a registered charity.
- **Explanation**:
  - Registers a new charity (`newCharity`).
  - Impersonates the admin and calls `verifyCharity` to verify the charity.
  - Asserts that the charity is now verified (`isVerified(newCharity)` returns `true`).

4. `testCharityVerifyCharityAdminWithAdminRevert`

```
function testCharityVerifyCharityAdminWithAdminRevert() public {
    address newCharity = makeAddr("newCharity");
    vm.startPrank(newCharity);
    registryContract.registerCharity(newCharity);
    vm.stopPrank();

    vm.startPrank(donor);
    vm.expectRevert("Only admin can verify");
    registryContract.verifyCharity(newCharity);
    vm.stopPrank();
}
```

- **Purpose**: Ensures that a non-admin user cannot verify a charity.
- **Explanation**:
  - Registers a new charity (`newCharity`).
  - Impersonates a non-admin user (`donor`) and attempts to call `verifyCharity`.
  - Expects the call to revert with the message "Only admin can verify".

5. `testCharityVerifyCharityAdminWithAdminRevertNotRegisteredCharities`

```
function
testCharityVerifyCharityAdminWithAdminRevertNotRegisteredCharities()
public {
    address newCharity = makeAddr("newCharity");

    vm.startPrank(admin);
    vm.expectRevert("Charity not registered");
    registryContract.verifyCharity(newCharity);
    vm.stopPrank();
}
```

- **Purpose**: Tests that a charity cannot be verified if it was not registered first.
- **Explanation**:
  - Creates a new charity address (`newCharity`) without registering it.
  - Impersonates the admin and attempts to verify the charity.
  - Expects the call to revert with the message "Charity not registered".

Detailed Test Functions for `GivingThanks.sol`

1. `testGivingThanksUpdateRegistryNotProtected`

```
function testGivingThanksUpdateRegistryNotProtected() public {
    address oldRegistryAddress = address(charityContract.registry());

    address attackerAddress = makeAddr("attackerAddress");
    vm.startPrank(attackerAddress);
    charityContract.updateRegistry(attackerAddress);
    vm.stopPrank();

    address updatedRegistryAddress = address(charityContract.registry());

    assertNotEq(oldRegistryAddress, updatedRegistryAddress);
}
```

- **Purpose**: Checks if the `updateRegistry` function can be called by anyone, demonstrating the lack of access control.
- **Explanation**:
  - Stores the current registry address.
  - Impersonates an attacker and calls `updateRegistry` with a new address.
  - Asserts that the registry address has changed, indicating the function is unprotected.

2. `testGivingThanksCreateTokenURIReturnCorrectValues`

```
function testGivingThanksCreateTokenURIReturnCorrectValues() public {
    uint256 donationAmount = 1 ether;
    uint256 initialTokenCounter = charityContract.tokenCounter();

    vm.deal(donor, 10 ether);
    vm.prank(donor);

    charityContract.donate{value: donationAmount}(charity);

    uint256 newTokenCounter = charityContract.tokenCounter();
    assertEq(newTokenCounter, initialTokenCounter + 1);

    address ownerOfToken = charityContract.ownerOf(initialTokenCounter);
    assertEq(ownerOfToken, donor);

    uint256 charityBalance = charity.balance;
    assertEq(charityBalance, donationAmount);

    console.log(block.timestamp);
    console.log(donor);
    string memory resFromFunction = charityContract._createTokenURI(
        donor,
        block.timestamp,
        1 ether
    );

    console.log(resFromFunction);

    string memory checkURI = charityContract.tokenURI(0);
    console.log(checkURI);

    assertEq(checkURI, resFromFunction);
}
```

- **Purpose**: Verifies that the `_createTokenURI` function generates the correct token URI and ensures the donation process works properly.
- **Explanation**:

- Sets up a donation amount and retrieves the initial token counter.
- Funds the `donor` and impersonates them for the donation.
- Asserts that the token counter has incremented by 1 and checks the owner of the new token.
- Verifies the charity's balance to ensure the donation was received.
- Calls `_createTokenURI` and compares the returned URI with the stored value (`tokenURI(0)`).

Conclusion: `forge coverage` before adding new test functions:

```
Ran 3 tests for test/GivingThanks.t.sol:GivingThanksTest
[PASS] testCannotDonateToUnverifiedCharity() (gas: 53483)
[PASS] testDonate() (gas: 303480)
[PASS] testFuzzDonate(uint96) (runs: 257, μ: 306495, ~: 305270)
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 162.65ms
(145.66ms CPU time)

Ran 1 test suite in 166.51ms (162.65ms CPU time): 3 tests passed, 0
failed, 0 skipped (3 total tests)
| File                      | % Lines         | % Statements    | % Branches
| % Funcs       |
|--------------------------|----------------|----------------|-------------
--|---------------|
| src/CharityRegistry.sol | 75.00% (6/8)   | 75.00% (6/8)   | 33.33% (2/6)
| 80.00% (4/5) |
| src/GivingThanks.sol    | 92.86% (13/14) | 94.12% (16/17) | 75.00% (3/4)
| 75.00% (3/4) |
| Total                    | 86.36% (19/22) | 88.00% (22/25) | 50.00%
(5/10) | 77.78% (7/9) |
```

`forge coverage` after adding new test functions:

```
Ran 10 tests for test/GivingThanks.t.sol:GivingThanksTest
[PASS] testCannotDonateToUnverifiedCharity() (gas: 53571)
[PASS] testCharityRegistryChangeAdminWithAdmin() (gas: 20954)
[PASS] testCharityRegistryChangeAdminWithAdminRevert() (gas: 16773)
[PASS] testCharityVerifyCharityAdminWithAdmin() (gas: 64488)
[PASS] testCharityVerifyCharityAdminWithAdminRevert() (gas: 41014)
[PASS]
testCharityVerifyCharityAdminWithAdminRevertNotRegisteredCharities() (gas:
19061)
[PASS] testDonate() (gas: 303370)
[PASS] testFuzzDonate(uint96) (runs: 257, μ: 306479, ~: 305335)
[PASS] testGivingThanksCreateTokenURIReturnCorrectValues() (gas: 349021)
[PASS] testGivingThanksUpdateRegistryNotProtected() (gas: 20592)
Suite result: ok. 10 passed; 0 failed; 0 skipped; finished in 165.03ms
(184.87ms CPU time)

Ran 1 test suite in 165.37ms (165.03ms CPU time): 10 tests passed, 0
failed, 0 skipped (10 total tests)
| File                      | % Lines          | % Statements     | % Branches
| % Funcs         |
|-------------------------|------------------|------------------|-----------
----|----------------|
| src/CharityRegistry.sol | 100.00% (8/8)   | 100.00% (8/8)   | 100.00%
(6/6) | 100.00% (5/5) |
| src/GivingThanks.sol    | 100.00% (14/14) | 100.00% (17/17) | 75.00%
(3/4)  | 100.00% (4/4) |
| Total                   | 100.00% (22/22) | 100.00% (25/25) | 90.00%
(9/10) | 100.00% (9/9) |
```

Low

[L-1] Missing Events for State-Changing Functions in GivingThanks.sol and CharityRegistry.sol

_Submitted by OxKoiner

Summary

The contracts **GivingThanks.sol** and **CharityRegistry.sol** currently do not emit any events when state-changing functions are called. Events are a crucial part of smart contract design as they provide a transparent and traceable log of important actions. In the current state, there is no event emission for key actions like donations and charity verification, making it difficult for users and developers to track important state changes.

Vulnerability Details

No any Vulnerability. Only code optimization.

Impact

The lack of events affects:

- **Transparency:** Users cannot easily trace state changes, making the contract less transparent.
- **dApp Integration:** Off-chain applications that depend on event logs for real-time updates will not receive notifications.
- **Auditability:** Without events, it is harder for developers and auditors to review the contract's behavior.

Tools Used

Manual review

Recommendations

Add event definitions and emit them in the relevant functions to ensure proper logging of important state changes. **Solution:**

1. In **GivingThanks.sol**, add the following event definition and emit the event in the `donate()` function:

```
// Event Definition
event DonateToCharity(address indexed _charity, address _donor);

// Updated Function
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);

    // Emit Event
    emit DonateToCharity(charity, msg.sender);

    // Create metadata for the tokenURI
    string memory uri = _createTokenURI(
        msg.sender,
        block.timestamp,
        msg.value
    );
    _setTokenURI(tokenCounter, uri);

    tokenCounter += 1;
}
```

2.In **CharityRegistry.sol**, add the following event definition and emit the event in the `verifyCharity()` function:

```
// Event Definition
event VerifiedCharity(address _charity);

// Updated Function
function verifyCharity(address charity) public {
    require(msg.sender == admin, "Only admin can verify");
    require(registeredCharities[charity], "Charity not registered");
    verifiedCharities[charity] = true;

    // Emit Event
    emit VerifiedCharity(charity);
}
```

[L-2] Unused Ownable Import in GivingThanks.sol

_Submitted by 0xKoiner

Summary

The `Ownable` contract from OpenZeppelin is imported in `GivingThanks.sol`, but it is not used anywhere in the contract. This import adds unnecessary complexity and increases the contract size, which could lead to higher gas costs and maintenance overhead. If `Ownable` is not needed, it should be removed to simplify the contract.

Vulnerability Details

No Effect. Only Gas Report

Impact

Low Impact: Since the `Ownable` contract is not being used, it does not affect the functionality or security of the contract directly. However, leaving unused imports in the contract increases the deployment size, making it inefficient and unnecessarily complex. This can result in higher deployment and interaction costs.

Tools Used

Manual code review

Recommendations

To improve the contract's efficiency and reduce unnecessary complexity, it is recommended to:

1. **Remove the `Ownable` import**: If the `Ownable` functionality is not required, remove the import statement for `Ownable` to streamline the contract and reduce deployment costs.
2. **Use `Ownable` if needed**: If there are plans to implement ownership-related functions (such as administrative control over certain functions), consider implementing `Ownable` properly.

Solution: Updated Code

1. If `Ownable` is not required:

Simply remove the import statement as follows:

```diff
- import "@openzeppelin/contracts/access/Ownable.sol";
```

2. If `Ownable` is required (for example, for future features such as administrative control):

If you plan to use `Ownable` for administrative functions (e.g., adding, removing, or verifying charities), you can implement it as follows:

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

In this case, you would be able to use the `onlyOwner` modifier on functions that only the contract owner should have access to.

Conclusion

The unused import of `Ownable` in `GivingThanks.sol` should either be removed if not needed, or fully implemented if future administrative functions require it. Removing unnecessary imports reduces contract size and improves efficiency, while adding ownership functionality could offer more control over the contract's operations.

[L-3] Optimize owner with immutable for Gas Savings in GivingThanks.sol

_Submitted by OxKoiner

Summary

The `owner` variable in `GivingThanks.sol` is declared as a `public` state variable. Since this value is set only once during contract deployment and is not intended to change thereafter, it can be marked as `immutable` to save gas costs associated with storage. By using `immutable`, the value is set only during the constructor, reducing gas costs for accessing the `owner`variable.

Vulnerability Details

No Vulnerability. Gas Report

Impact

Low Impact: The impact is primarily related to gas optimization. By making the `owner` variable `immutable`, gas costs for reading the `owner` address will be reduced. This is especially useful in contracts with frequent interactions, where saving on gas can lead to overall cost reductions. The functionality of the contract is not affected, as the `owner` is not intended to be changed after deployment.

Tools Used

Manual Review

Recommendations

It is recommended to mark the `owner` variable as `immutable` since it is only set once during contract deployment and does not change afterward. This can be achieved by using the `immutable` keyword.

```
address public immutable owner; // Changed to immutable
```

[L-4] Use Stable Solidity Version (0.8.18, 0.8.19 , 0.8.20 ....) Instead of Floating (^0.8.0)

_Submitted by OxKoiner

Summary

The current contract files, `GivingThanks.sol` and `CharityRegistry.sol`, use the floating pragma version `^0.8.0`, which allows for any version of Solidity starting from `0.8.0` to the next breaking change (e.g., `0.9.0`). This can lead to unexpected issues if a new Solidity version introduces breaking changes that are not backward-compatible with the current contract. It is recommended to use a specific stable version, such as `0.8.18, 0.8.19 , 0.8.20 ...............`, to ensure compatibility and avoid potential issues caused by unexpected updates.

Vulnerability Details

The caret (`^`) symbol means "compatible with version `0.8.0` and any newer version until the next breaking change" (e.g., `0.9.0`). This introduces risks because:

- Solidity may introduce breaking changes that are not backward compatible, causing unexpected behavior or vulnerabilities in the contract.
- It can result in unpredictable results during contract deployment, especially when the compiler is updated or changes between patch or minor versions.

Impact

The impact of leaving the floating version is mainly related to ensuring contract stability and predictability. By locking the version to `0.8.19`, the contract will always compile with the same version, ensuring that no future compiler changes cause issues.

Tools Used

Manual Review

Recommendations

It is highly recommended to use a fixed Solidity version to avoid unexpected issues related to compiler changes.

[L-5] Code Design and Optimizations in GivingThanks.sol

_Submitted by 0xKoiner

Summary

The `GivingThanks.sol` contract needs some improvements to enhance code readability, structure, naming conventions, and performance. The following changes are proposed:

- Rename variables to follow a consistent naming convention.
- Reorganize functions for better readability and maintainability.
- Add Natspec documentation for better understanding and standardization.
- Improve imports and code layout for better modularity.

Vulnerability Details

The contract currently lacks standardized variable naming, clear function ordering, and missing Natspec documentation. Additionally, the imports are not modular, and there is a lack of clarity in how the contract functions are arranged.

Impact

Low Impact: The changes do not introduce any new vulnerabilities or risks but aim to enhance the readability, maintainability, and standardization of the contract code. Adopting a more modular design will make the code easier to understand, and ensuring Natspec documentation will help external developers or auditors understand the functionality.

Tools Used

Manual Review

Recommendations

1. **Renaming Variables**:
    - `tokenCounter` should be renamed to `s_tokenCounter` for consistency with naming conventions.
    - `owner` should be renamed to `i_owner` to follow the convention for immutable state variables.
2. **Reordering Functions**: Functions should be ordered as per best practices:
    - `external` functions first
    - `public` functions next
    - `internal` functions after that
    - `private` functions last
    - View and pure functions should be grouped separately
3. **Adding Natspec**: Natspec comments should be added to all functions to enhance code documentation and make the contract more understandable.
4. **Improved Imports**: Use modular import statements to improve clarity and separation of concerns.

Solution: Updated Code:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.19;

// Module Imports
import {CharityRegistry} from "./CharityRegistry.sol";
import {ERC721URIStorage} from
"@openzeppelin/contracts/token/ERC721/extensions/ERC721URIStorage.sol";
import {Ownable} from "@openzeppelin/contracts/access/Ownable.sol";
import {Strings} from "@openzeppelin/contracts/utils/Strings.sol";
import {Base64} from "@openzeppelin/contracts/utils/Base64.sol";

/**
 * @title GivingThanks
 * @dev This contract allows users to donate to verified charities and
receive a token of thanks (NFT) as proof of donation.
 */
contract GivingThanks is ERC721URIStorage {
    // Immutable state variable for the contract owner
    address public immutable i_owner;
    // State variable to track the token ID counter
    uint256 public s_tokenCounter;

    // CharityRegistry contract instance
    CharityRegistry public registry;

    /**
     * @dev Constructor function to initialize the contract
     * @param _registry The address of the CharityRegistry contract.
     */
    constructor(address _registry) ERC721("DonationReceipt", "DRC") {
        registry = CharityRegistry(_registry);
        i_owner = msg.sender;
        s_tokenCounter = 0;
    }

    // External Functions

    /**
     * @dev Allows a user to donate to a verified charity and mint an NFT
as proof of donation.
     * @param charity The address of the charity to donate to.
     */
    function donate(address charity) external payable {
        require(registry.isVerified(charity), "Charity not verified");

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

        // Mint an NFT for the donor
        _mint(msg.sender, s_tokenCounter);

        // Create metadata for the token URI
        string memory uri = _createTokenURI(
            msg.sender,
            block.timestamp,
            msg.value
        );

        // Set the token URI
        _setTokenURI(s_tokenCounter, uri);

        // Increment token counter
        s_tokenCounter += 1;
    }

    // Public Functions

    /**
     * @dev Allows the owner to update the registry address.
     * @param _registry The new address of the CharityRegistry contract.
     */
    function updateRegistry(address _registry) public {
        registry = CharityRegistry(_registry);
    }

    // Internal Functions

    /**
     * @dev Internal function to create the token URI metadata.
     * @param donor The address of the donor.
     * @param date The timestamp of the donation.
     * @param amount The amount donated.
     * @return A string representing the token URI (base64 encoded JSON).
     */
    function _createTokenURI(
        address donor,
        uint256 date,
        uint256 amount
    ) internal pure returns (string memory) {
        // Create JSON metadata
        string memory json = string(
            abi.encodePacked(
                '{"donor":"',
                Strings.toHexString(uint160(donor), 20),
                '","date":"',
                Strings.toString(date),
                '","amount":"',
                Strings.toString(amount),
                '"}'
            )
        );

        // Encode in base64 using OpenZeppelin's Base64 library
        string memory base64Json = Base64.encode(bytes(json));

        // Return the data URL
        return
            string(
                abi.encodePacked("data:application/json;base64,",
base64Json)
            );
    }

    // Private Functions

    // No private functions in this contract
}
```

2. **Explanation of Changes:**

Renaming Variables:

- `tokenCounter` is renamed to `s_tokenCounter` to align with the standard for state variables that are not immutable.
- `owner` is renamed to `i_owner` to reflect its immutable nature.

Reorganizing Functions:

- The functions are now grouped as follows:
    - **External Functions**: Functions that are intended to be called from outside the contract, such as `donate`.
    - **Public Functions**: Functions that are callable externally but also from within the contract (e.g., `updateRegistry`).
    - **Internal Functions**: Functions that can only be called within the contract, such as `_createTokenURI`.
    - **Private Functions**: No private functions were required in this case, so the section is omitted.

Adding Natspec:

- Natspec comments have been added to all functions, explaining their purpose, parameters, and expected behavior. This improves code clarity and helps with external audits and understanding of the contract.

Improved Imports:

- The import statements have been updated to use modular imports:
    - `import {CharityRegistry} from "./CharityRegistry.sol";`
    - `import {ERC721URIStorage} from "@openzeppelin/contracts/token/ERC721/extensions/ERC721URIStorage.sol";`
    - `import {Ownable} from "@openzeppelin/contracts/access/Ownable.sol";`
    - `import {Strings} from "@openzeppelin/contracts/utils/Strings.sol";`
    - `import {Base64} from "@openzeppelin/contracts/utils/Base64.sol";`

This improves clarity and ensures each contract is imported properly without any unnecessary dependencies.

[L-6] Code Design and Optimizations in CharityRegistry.sol

_Submitted by 0xKoiner

Summary

The contract `CharityRegistry.sol` contains multiple design issues that can be optimized for gas efficiency, readability, and clarity. Additionally, the code lacks proper Natspec documentation. The following optimizations and changes are recommended:

- Use more efficient variable naming with a clear naming convention.
- Reorganize functions for better clarity and code layout.
- Introduce proper Natspec comments for functions to improve documentation and understanding of the contract's functionality.

Vulnerability Details

No any Vulnerability. Only optimization for code design

Impact

Low Impact. These issues don’t necessarily pose security risks or major vulnerabilities. However, optimizing the code design and adding Natspec comments significantly improve code readability, maintainability, and reduce potential future errors.

Tools Used

Manual Review

Recommendations

1. **Update Variable Naming:**
   Rename `admin`, `verifiedCharities`,
   and `registeredCharities` to `s_admin`, `s_verifiedCharities`,
   and `s_registeredCharities` for consistency and readability.

2. **Reorganize Functions:**
   Functions should be organized as follows:

   - External functions first.
   - Public functions next.
   - Internal and private functions next.
   - View and pure functions should be placed accordingly at the bottom.

3. **Add Natspec Documentation:**
   Add proper Natspec comments to all functions to explain the purpose, parameters, and returns of each function.

4. **Check for Address(0):**
   Consider adding checks for `address(0)` in functions where addresses are updated or used to avoid issues with sending Ether to the zero address.

Recommendations**😗*

1. **Updated CharityRegistry.sol Contract:**

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

/**
 * @title CharityRegistry Contract
 * @dev This contract manages charity registration and verification.
 * The contract allows only the admin to perform administrative actions,
such as verifying charities and changing the admin.
 */
```

```
contract CharityRegistry {

    // --- State Variables ---
    address public immutable s_admin;
    mapping(address => bool) public s_verifiedCharities;
    mapping(address => bool) public s_registeredCharities;

    // --- Modifiers ---
    modifier onlyAdmin() {
        require(msg.sender == s_admin, "Only admin can perform this
action");
        _;
    }

    modifier nonZeroAddress(address _addr) {
        require(_addr != address(0), "Address cannot be zero");
        _;
    }

    // --- Constructor ---
    /**
     * @dev Constructor sets the initial admin of the contract.
     * @param _admin The address of the admin
     */
    constructor(address _admin) nonZeroAddress(_admin) {
        s_admin = _admin;
    }

    // --- External Functions ---
    /**
     * @dev Changes the admin of the contract to a new address.
     * @param newAdmin The address of the new admin
     */
    function changeAdmin(address newAdmin) external onlyAdmin
nonZeroAddress(newAdmin) {
        s_admin = newAdmin;
    }

    /**
     * @dev Verifies a charity as legitimate.
     * @param charity The address of the charity to verify
     */
    function verifyCharity(address charity) external onlyAdmin
nonZeroAddress(charity) {
        require(s_registeredCharities[charity], "Charity not registered");
        s_verifiedCharities[charity] = true;
    }

    /**
     * @dev Registers a new charity. Only admin can register charities.
     * @param charity The address of the charity to register
     */
    function registerCharity(address charity) external onlyAdmin
nonZeroAddress(charity) {
```

```
            s_registeredCharities[charity] = true;
        }

        // --- Public View Functions ---
        /**
         * @dev Checks whether a charity is verified.
         * @param charity The address of the charity to check
         * @return True if the charity is verified, otherwise false
         */
        function isVerified(address charity) public view returns (bool) {
            return s_verifiedCharities[charity];
        }

        /**
         * @dev Checks whether a charity is registered.
         * @param charity The address of the charity to check
         * @return True if the charity is registered, otherwise false
         */
        function isRegistered(address charity) public view returns (bool) {
            return s_registeredCharities[charity];
        }
    }
```

[L-7] Custom Errors for Gas Optimization in GivingThanks.sol and CharityRegistry.sol

_Submitted by 0xKoiner

Summary

The contracts in both `GivingThanks.sol` and `CharityRegistry.sol` use `require` statements with string error messages, which are less gas-efficient compared to custom errors. Switching to custom errors will reduce gas costs and improve contract efficiency.

Vulnerability Details

In both contracts, `require` statements are used with string messages, which consume more gas due to the storage of the error message strings. Custom errors introduced in Solidity 0.8.4 are more gas-efficient because they only consume gas when the error is actually triggered and don't store strings.

Impact

The current approach with string-based `require` messages leads to unnecessary gas consumption when errors are thrown. Replacing these with custom errors will optimize the contract, reducing transaction costs and improving performance.

Tools Used

Manual Review

Recommendations

1. Use custom errors instead of string-based error messages.
2. Change the `require` statements to use `revert` with custom error messages.
3. Modify the Solidity code to implement the custom errors

Solution: GivingThanks.sol:

Code Changes: Replace the string-based error messages with custom errors:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "./CharityRegistry.sol";
import
"@openzeppelin/contracts/token/ERC721/extensions/ERC721URIStorage.sol";
import {Strings}  from "@openzeppelin/contracts/utils/Strings.sol";
import {Base64} from "@openzeppelin/contracts/utils/Base64.sol";

// Custom errors
error GivingThanks__CharityNotVerified();
error GivingThanks__FailedToSendEther();

contract GivingThanks is ERC721URIStorage {
    CharityRegistry public registry;
    uint256 public tokenCounter;
    address public immutable i_owner;

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

    function donate(address charity) public payable {
        if (!registry.isVerified(charity)) revert
GivingThanks__CharityNotVerified();

        (bool sent, ) = charity.call{value: msg.value}("");
        if (!sent) revert GivingThanks__FailedToSendEther();

        _mint(msg.sender, tokenCounter);

        // Create metadata for the tokenURI
        string memory uri = _createTokenURI(
            msg.sender,
            block.timestamp,
            msg.value
        );
        _setTokenURI(tokenCounter, uri);

        tokenCounter += 1;
    }
    function _createTokenURI(
        address donor,
        uint256 date,
        uint256 amount
    ) internal pure returns (string memory) {
        // Create JSON metadata
        string memory json = string(
            abi.encodePacked(
                '{"donor":"',
                Strings.toHexString(uint160(donor), 20),
                '","date":"',
                Strings.toString(date),
                '","amount":"',
                Strings.toString(amount),
                '"}'
            )
        );

        // Encode in base64 using OpenZeppelin's Base64 library
        string memory base64Json = Base64.encode(bytes(json));

        // Return the data URL
        return
            string(
                abi.encodePacked("data:application/json;base64,",
base64Json)
            );
    }

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

Solution: CharityRegistry.sol

Code Changes

Replace the string-based error messages with custom errors:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

// Custom errors
error CharityRegistry__OnlyAdminCanVerify();
error CharityRegistry__CharityNotRegistered();
error CharityRegistry__OnlyAdminCanChangeAdmin();

contract CharityRegistry {
    address public immutable i_admin;
    mapping(address => bool) public s_verifiedCharities;
    mapping(address => bool) public s_registeredCharities;

    constructor() {
        i_admin = msg.sender;
    }

    function registerCharity(address charity) public {
        s_registeredCharities[charity] = true;
    }

    function verifyCharity(address charity) public {
        if (msg.sender != i_admin) revert
CharityRegistry__OnlyAdminCanVerify();
        if (!s_registeredCharities[charity]) revert
CharityRegistry__CharityNotRegistered();
        s_verifiedCharities[charity] = true;
    }

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

    function changeAdmin(address newAdmin) public {
        if (msg.sender != i_admin) revert
CharityRegistry__OnlyAdminCanChangeAdmin();
        i_admin = newAdmin;
    }
}
```

Coverage

`forge coverage` before adding new test functions:

```
Ran 3 tests for test/GivingThanks.t.sol:GivingThanksTest
[PASS] testCannotDonateToUnverifiedCharity() (gas: 53483)
[PASS] testDonate() (gas: 303480)
[PASS] testFuzzDonate(uint96) (runs: 257, μ: 306495, ~: 305270)
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 162.65ms
(145.66ms CPU time)

Ran 1 test suite in 166.51ms (162.65ms CPU time): 3 tests passed, 0
failed, 0 skipped (3 total tests)
| File                      | % Lines         | % Statements    | % Branches
| % Funcs       |
|---------------------------|-----------------|-----------------|------------
--|---------------|
| src/CharityRegistry.sol | 75.00% (6/8)    | 75.00% (6/8)    | 33.33% (2/6)
| 80.00% (4/5) |
| src/GivingThanks.sol     | 92.86% (13/14) | 94.12% (16/17) | 75.00% (3/4)
| 75.00% (3/4) |
| Total                     | 86.36% (19/22) | 88.00% (22/25) | 50.00%
(5/10) | 77.78% (7/9) |
```

`forge coverage` after adding new test functions:

```
Ran 10 tests for test/GivingThanks.t.sol:GivingThanksTest
[PASS] testCannotDonateToUnverifiedCharity() (gas: 53571)
[PASS] testCharityRegistryChangeAdminWithAdmin() (gas: 20954)
[PASS] testCharityRegistryChangeAdminWithAdminRevert() (gas: 16773)
[PASS] testCharityVerifyCharityAdminWithAdmin() (gas: 64488)
[PASS] testCharityVerifyCharityAdminWithAdminRevert() (gas: 41014)
[PASS]
testCharityVerifyCharityAdminWithAdminRevertNotRegisteredCharities() (gas:
19061)
[PASS] testDonate() (gas: 303370)
[PASS] testFuzzDonate(uint96) (runs: 257, μ: 306479, ~: 305335)
[PASS] testGivingThanksCreateTokenURIReturnCorrectValues() (gas: 349021)
[PASS] testGivingThanksUpdateRegistryNotProtected() (gas: 20592)
Suite result: ok. 10 passed; 0 failed; 0 skipped; finished in 165.03ms
(184.87ms CPU time)

Ran 1 test suite in 165.37ms (165.03ms CPU time): 10 tests passed, 0
failed, 0 skipped (10 total tests)
| File                      | % Lines         | % Statements    | % Branches
| % Funcs        |
|--------------------------|-----------------|-----------------|-----------
----|----------------|
| src/CharityRegistry.sol | 100.00% (8/8)   | 100.00% (8/8)   | 100.00%
(6/6) | 100.00% (5/5) |
| src/GivingThanks.sol    | 100.00% (14/14) | 100.00% (17/17) | 75.00%
(3/4)  | 100.00% (4/4) |
| Total                   | 100.00% (22/22) | 100.00% (25/25) | 90.00%
(9/10) | 100.00% (9/9) |
```

Slither

Summary

- reentrancy-vulnerabilities-2 (1 results) (High)
- allows-old-versions (1 results) (Informational)
- should-be-immutable (1 results) (Low)
- zero-address-validation (1 results) (Informational)

allows-old-versions

Impact: Informational Confidence: Low

- [ ] ID-0 [Pragma version^0.8.0 (src/CharityRegistry.sol#2) allows old versions] solc-0.8.20 is not recommended for deployment `Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity`

should-be-immutable

Impact: Low Confidence: Low/Gas

- [ ] ID-1 Parameter GivingThanks.updateRegistry(address)._registry (src/GivingThanks.sol#67) is not in mixedCase `Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions`

src/protocol/GivingThanks.sol#L13

zero-address-validation

Impact: Informational Confidence: High

- [ ] ID-2 CharityRegistry.changeAdmin(address).newAdmin (src/CharityRegistry.sol#27) lacks a zero-check on :
    - admin = newAdmin (src/CharityRegistry.sol#29)

  `Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#missing-zero-address-validation`

src/CharityRegistry.sol#29

reentrancy-vulnerabilities-2

Impact: High Confidence: Medium

- [ ] ID-3 INFO:Detectors: Reentrancy in GivingThanks.donate(address) (src/GivingThanks.sol#21-37): External calls: - (sent) = charity.call{value: msg.value}() (src/GivingThanks.sol#23) State variables written after the call(s): - _mint(msg.sender,tokenCounter) (src/GivingThanks.sol#26) - _balances[from] -= 1 (lib/openzeppelin-contracts/contracts/token/ERC721/ERC721.sol#256) - _balances[to] += 1 (lib/openzeppelin-contracts/contracts/token/ERC721/ERC721.sol#262) - _mint(msg.sender,tokenCounter) (src/GivingThanks.sol#26) - _owners[tokenId] = to (lib/openzeppelin-contracts/contracts/token/ERC721/ERC721.sol#266) - _mint(msg.sender,tokenCounter) (src/GivingThanks.sol#26) - _tokenApprovals[tokenId] = to (lib/openzeppelin-contracts/contracts/token/ERC721/ERC721.sol#424) - _setTokenURI(tokenCounter,uri) (src/GivingThanks.sol#34) - _tokenURIs[tokenId] = _tokenURI (lib/openzeppelin-contracts/contracts/token/ERC721/extensions/ERC721URIStorage.sol#58) - tokenCounter += 1 (src/GivingThanks.sol#36) `Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-2`

  ```
  INFO:Detectors:
  Reentrancy in GivingThanks.donate(address) (src/GivingThanks.sol#21-
  ```

```
37):
External calls: - (sent) = charity.call{value: msg.value}()
(src/GivingThanks.sol#23)
Event emitted after the call(s): - Approval(owner,to,tokenId)
(lib/openzeppelin-contracts/contracts/token/ERC721/ERC721.sol#420) -
\_mint(msg.sender,tokenCounter) (src/GivingThanks.sol#26) -
MetadataUpdate(tokenId) (lib/openzeppelin-
contracts/contracts/token/ERC721/extensions/ERC721URIStorage.sol#59) -
\_setTokenURI(tokenCounter,uri) (src/GivingThanks.sol#34) -
Transfer(from,to,tokenId) (lib/openzeppelin-
contracts/contracts/token/ERC721/ERC721.sol#268) -
\_mint(msg.sender,tokenCounter) (src/GivingThanks.sol#26)
`Reference: https://github.com/crytic/slither/wiki/Detector-
Documentation#reentrancy-vulnerabilities-3`
```

src/GivingThanks.sol#21-37