SMART CONTRACT AUDIT

Conducted for: Humble Donations

Waleed Ahmed
VULSIGHT
18/8/2024

Contents

Executive Summary:

The client contacted our Consulting Firm to conduct a smart contract audit on their Solidity based smart contract. The HumbleDonations smart contract, designed to facilitate charitable donations on the Ethereum blockchain, demonstrates a robust architecture for creating and managing donation projects. The contract supports ETH donations aswell as multi-token contributions.

Our comprehensive audit revealed no critical vulnerabilities. However, we identified areas for improvements. These include some logical flaws that would have had interfered with normal contract operations. Key recommendations include implementing measures against logical vulnerabilities and adding mechanisms to update hard-coded addresses for better long-term flexibility. A retest of the vulnerabilities was done after the remediations by the client. All of the vulnerabilities were found to be fixed.

Audit conducted by: Waleed Ahmed

[illegible]

Project Background:

The smart contracts are written in solidity. The contract is supposed to be deployed on the Ethereum Mainnet or the Arbitrum Mainnet. The donation contract as well as the token contract are currently deployed on the Sepolia testnet.

Audit Scope:

Deployed Addresses	**0x23F02AE5a4331EF595C74Cb86bdb2A99B3940727**
Chain	**Sepolia**
Language	**Solidity**
Audit Completion Date	**18/8/24**

Contract Summary:

- **Contract architecture:** The HumbleDonations contract utilizes the UUPS (Universal Upgradeable Proxy Standard) pattern, allowing for future upgrades while maintaining data persistence.
- **Token standard implementation:** The contract correctly implements the ERC721 standard for non-fungible tokens, representing individual donation projects.

- **Multi-token support:** The contract efficiently handles donations in various ERC20 tokens as well as native ETH. It integrates with Uniswap for token swaps.
- **Access control:** The contract employs OpenZeppelin's Ownable contract to restrict access to critical functions, such as setting tax percentages and withdrawing excess funds, to the contract owner only.
- **Reentrancy protection:** The contract utilizes OpenZeppelin's ReentrancyGuard to protect against reentrancy attacks in critical functions like donate and safeMint.
- **Event emission:** The contract emits appropriate events for important actions such as project creation (ProjectCreated), donations (DonationMade), and project deletion (ProjectDeleted). This enables effective off-chain tracking and indexing of contract activities.
- **Whitelist mechanism:** The contract implements a Merkle tree-based whitelist for controlling which ERC20 tokens can be used for donations. This provides a gas-efficient way to manage a large list of allowed tokens without storing them all on-chain.
- **Slippage protection:** The donate function includes slippage parameters to protect against unfavorable swap rates when converting donations to the internal HDT token. This shows consideration for real-world trading conditions and user protection.
- **Donation Storage:** The smart contract does not itself store any donation funds. The tax is divided into multiple recipients while the donation is sent to the project creator.
- **Clear and auditable code:** The contract code is well-structured, properly commented, and follows Solidity conventions, making it readable and easier to audit. Function names are descriptive, and complex operations are broken down into smaller, manageable steps.

Various tools such as Slither and different LLM scanners were used in the audit. Extensive manual code review was also done to ensure that any vulnerability if present could be detected in the audit.

We found:

- **0 Critical risk finding**
- **1 High risk finding**
- **2 Medium risk findings**
- **1 Low risk finding**

Severity Definitions:

Risk Level	Description
Critical	**Any kind of vulnerability that could lead to direct token or monetary loss**
High	**Any type of vulnerability that could disrupt the proper functioning of smart contract or indirectly lead to token or monetary loss.**
Medium	**Any type of vulnerability that could cause undesired actions but no serious disruption or monetary loss**
Low	**Any type of vulnerability that doesn't have any significant impact on the execution of the proper functioning of contract**

Technical Checks:

Checks	Result
Solidity version not specified	Passed
Solidity version too old	Passed
Integer overflow/underflow	Passed
Function input parameters check bypass	Passed
Insufficient randomness used	Passed
Fallback function misuse	Passed
Race Condition	Passed
Matching Project Specifications	Passed
Logical Vulnerabilities	Passed
Function visibility not explicitly declared	Passed
Use of deprecated keywords/functions	Passed
Out of Gas issue	Passed
Front Running	Passed
Insufficient Decentralization	Passed
Function input parameters lack of check	Passed
Proper function Access Control	Passed
Hardcoded Data	Passed

Code Quality and Documentation:

The audit scope included one smart contract, which was written in Solidity programming language. The code is well indented and clear in nature.

Audit Findings:

Critical findings:

Zero critical findings were found in the smart contract.

High Findings:

One high finding was found in the smart contract.

Use of transfer Function for ETH Donations:

The donate function in the HumbleDonations contract contains a vulnerability related to the use of the transfer function for sending ETH to the token owner. Specifically, the transfer use in the *donate* function:

```
        if (erc20Token != address(HDT)) {
            total = amount - taxAmount;
            if (erc20Token == address(0)) {
                require(msg.value >= amount, "Incorrect amount of ETH sent");
                swapExactInputSingleETH(taxAmount, slippageHDT);
                payable(tokenOwner).transfer(total);
            }
```

This use of **transfer** can lead to potential donation failures due to the following reasons:

1. **Gas Limit:** The transfer function is hard-coded with a gas stipend of 2300 gas. This gas limit is sufficient for basic ETH transfers but can be inadequate if the receiving address is a contract with a complex fallback function.
2. **Compatibility Issues:** Some smart contract wallets or multi-sig wallets may require more than 2300 gas to process incoming ETH, causing the transfer to fail consistently for these types of recipients.
3. **Future Proofing:** As the Ethereum network evolves, gas costs for operations may change. The fixed gas stipend of transfer does not account for potential future changes in gas costs.
4. **Unexpected Reverts:** If the recipient is a contract that does not accept ETH or has a fallback function that reverts under certain conditions, the entire donation transaction will fail.

These issues can result in donations being unexpectedly rejected, leading to a poor user experience and potential loss of donations. Moreover, it can create a situation where some project owners (those with regular EOA addresses) can receive donations, while others (using smart contract wallets) cannot, introducing unfairness in the platform.

To mitigate this vulnerability, it is recommended to replace the transfer function with a call method, which allows for custom gas limits and better handles the potential complexities of the receiving address:

Current Status:
The finding was found to be fixed by using the call opcode for transfer:

```
(bool success, ) = payable(tokenOwner).call{value: total}("");
require(success, "ETH transfer failed");
```

Medium Findings:

Two medium findings were found in the smart contract.

Bypassing Single Project Ownership Limit via NFT Transfer:

The HumbleDonations contract contains a vulnerability that allows an address to own multiple projects, circumventing the intended limit of one project per address. This vulnerability stems from the contract's failure to enforce the ownership limit during NFT transfers.

The contract aims to limit each address to owning only one project NFT. This is enforced in the *safeMint* function with the check:

```
require(balanceOf(to) == 0, "Address already owns a token");
```

The *safeMint* function prevents an address from directly creating multiple projects, the contract does not prevent an address from receiving additional project NFTs through transfers.

Exploit Scenario:

1. User A creates a project, receiving NFT #1.
2. User B creates a project, receiving NFT #2.
3. User A can then receive NFT #2 from User B through a standard ERC721 transfer.
4. User A now owns two project NFTs, bypassing the intended limit.

These issues can result in a breakdown of the platform's intended equitable structure, leading to potential centralization of donations and reduced trust in the system. Moreover, it creates a situation where users who follow the intended usage (one project per address) are at a disadvantage compared to those who exploit this vulnerability.

Current Status:
The finding was found to be fixed by adding an additional check by overriding the transferFrom function:

```
    function transferFrom(
        address from,
        address to,
        uint256 tokenId
    ) public virtual override(ERC721Upgradeable, IERC721) {
        // Ensure the recipient does not already own a token
        require(balanceOf(to) == 0, "Recipient already owns a project NFT");

        // Proceed with the normal transfer operation
        super.transferFrom(from, to, tokenId);
    }
```

Duplicate Project Title for Token ID 0:

The HumbleDonations contract contains a vulnerability that allows the creation of a duplicate project title for token ID 0, bypassing the intended uniqueness check. This vulnerability stems from the contract's use of 0 as both the starting token ID and as a sentinel value in the *projectTitleToTokenId* mapping.

Specifically, the vulnerability is in the *safeMint* function:

```
    function safeMint(
        address to,
        string memory uri,
        string memory projectTitle
    ) external payable nonReentrant {
        require(msg.value >= fees + mintRate, "Not enough ETH sent");
        require(balanceOf(to) == 0, "Address already owns a token");
        require(
            projectTitleToTokenId[projectTitle] == 0,
            "Project title already exists"
        );
        uint256 tokenId = _tokenIdCounter;
        ………………………………………………………
      }
```

The vulnerability arises due to the following reasons:

- **Token IDs start from 0:** The *_tokenIdCounter* is initialized to 0, meaning the first minted token will have ID 0.
- **Uniqueness check uses 0:** The contract checks for uniqueness of project titles by verifying if *projectTitleToTokenId[projectTitle]* == 0.
- **Conflating meanings of 0:** The value 0 is used both to represent "no token exists with this title" and as a valid token ID.

Exploit Scenario:

- A victim mints the first token (ID 0) with a specific project title.
- The attacker can now mint another token with the same project title

Current Status:
The finding was found to be fixed by initializing the *_tokenIdCounter* to 1:

```
_tokenIdCounter = 1;
```

Low findings:

One low finding was found in the smart contract.

Redundant Variables and Minor Inefficiencies:

The HumbleDonations contract contains instances of redundant variables and minor inefficiencies that, while not critical, could be optimized to improve code clarity and slightly reduce gas costs. Specifically:

1. The contract declares a *fees* state variable:
   ```
   uint256 public fees;
   ```
   However, the value of the variable is never initialized in the contract. The comment itself suggests it may be unnecessary.
2. The *fees* variable even though having a zero Value is referenced in the SafeMint function:
   ```
   require(msg.value >= fees + mintRate, "Not enough ETH sent");
   ```

Current Status:

The finding was found to be fixed by the removal of the redundant variable.

Disclaimer: