KRYOTECH LTD.

Smart Contract Audit

VxC Smart Contract 0.0.2

111 New Union Street, Coventry, CV1 2NT

11-19-2023

1. Introduction

1.1. Purpose of the Report

This report provides a comprehensive security assessment of the "EthTransferWithFee" smart contract. It aims to identify potential vulnerabilities, assess coding practices, and offer mitigation strategies to enhance the contract's security. This document serves as a guide for developers, auditors, and stakeholders involved in the project to understand the contract's security posture and areas for improvement.

1.2. Overview of the "EthTransferWithFee" Smart Contract

The "EthTransferWithFee" is a Solidity smart contract designed for transferring Ethereum with an applied fee structure. It interacts with ERC721 tokens and provides functionalities for single and group transfers, fee management, and owner control. The contract is implemented using Solidity version 0.8.13 and incorporates features like event logging, input validation, and access control to ensure secure and efficient operations. This report dissects the contract's code to unveil its operational mechanics, security measures, and areas where it could be vulnerable to attacks or malfunctions.

2. Coding and Security Standards

2.1. Solidity 0.8.13 Features and Best Practices

The "EthTransferWithFee" contract is written in Solidity 0.8.13, which includes built-in protections against certain vulnerabilities like integer overflow and underflow. Best practices in this version of Solidity include:

- Utilizing explicit visibility declarations for functions and state variables.
- Employing immutable and constant keywords for values that do not change, optimizing gas costs.
- Implementing error handling through require, revert, and assert.

2.2. Ethereum Smart Contract Security Best Practices

Key practices for ensuring Ethereum smart contract security encompass:

- Adhering to the Checks-Effects-Interactions pattern to mitigate reentrancy attacks.
- Avoiding common pitfalls like reusing code without understanding its context and dependencies.
- Regularly updating and testing contracts to ensure compatibility with new Ethereum updates.

2.3. Common Vulnerabilities as Outlined by the SWC Registry

The Smart Contract Weakness Classification (SWC) Registry categorizes common smart contract vulnerabilities, including:

- Reentrancy attacks.
- Issues with gas handling and DoS with (unexpected) revert.
- Improper access control and authentication flaws.
- Shortcomings in randomness and pseudorandomness.
- Pitfalls in contract upgradeability and delegatecall.

This contract's assessment will particularly focus on vulnerabilities relevant to its use case and the specific features of Solidity 0.8.13.

3. Contract Overview

3.1. Description of the Contract's Functionality

The "EthTransferWithFee" smart contract facilitates the transfer of Ethereum with a fee structure. Its primary functionalities include:

- Single Ethereum transfers with an applied fee.
- Group transfers where fees are applied differently based on ERC721 token ownership.
- Administrative functions allowing the owner to change fee percentages and update the linked ERC721 contract.

3.2. Analysis of the Contract Structure and Key Components

- Owner Management: Functions for transferring contract ownership, ensuring control is managed securely.
- Fee Handling: The contract applies different fee percentages for transfers depending on ERC721 token ownership.
- Transfer Functions: Both single and group transfer functions are included, with fee calculations based on predefined percentages.
- ERC721 Interaction: Functions to check the balance of ERC721 tokens for fee determination.
- Event Logging: For transparency and tracking, events are emitted for transfers and administrative changes.

The structure of the contract reflects a focus on secure transfer mechanisms while providing flexibility in fee management and integration with ERC721 tokens.

4. Vulnerabilities and Risks

4.1. Reentrancy Risks

- The contract uses the .send() method for transferring Ether, which is susceptible to reentrancy attacks, though less so than .call().
- The risk arises when external calls are made before updating the contract's state, potentially allowing attackers to drain funds.

4.2. Gas Limitations and Loop Inefficiencies

- Functions groupTransferWithSeperateAmount and groupTransferWithSameAmount use loops for batch processing, which can consume significant gas.
- Large arrays can cause transactions to fail due to gas limit constraints, making these functions potentially vulnerable to DoS attacks.

4.3. Fee Management

- The contract allows the owner to update fee percentages, which could be exploited if not properly managed.
- Risks involve setting unreasonable fees, either too high or too low, affecting the contract's fairness and functionality.

5. Mitigation Strategies

5.1. Implementing the Checks-Effects-Interactions Pattern to Mitigate Reentrancy Risks

- Update the state variables before making external calls. This pattern reduces the risk of reentrancy by ensuring that the contract's state is settled before interacting with other contracts.
- Example: In the singleTransfer function, update the balance or state before sending Ether.

5.2. Best Practices for Loop Management to Avoid Gas Limitations

- Implement limits on the number of iterations or array sizes in functions like groupTransferWithSeperateAmount.
- Consider using pagination or splitting large transactions into smaller batches.

5.3. Secure Methods for Managing and Updating Fee Percentages

- Implement a multi-step process for changing fees, such as a timelock or multi-signature requirement, to prevent abrupt or unauthorized changes.
- Set reasonable bounds for fee percentages and include validation checks to prevent setting fees that are too high or too low.

6. Code Samples

6.1. Improved Single Transfer Function with Checks-Effects-Interactions Pattern

```
function singleTransfer[address payable receiver] external payable
{
    require[receiver != address[0], "Invalid address"];
    require[msg.value > 0, "Invalid amount"];

    uint256 feeAmount = getERC721Balance[msg.sender] ? 0 :
[msg.value * feePercentage] / 100;
    uint256 transferAmount = msg.value - feeAmount;

    // Check and Effect
    if [feeAmount > 0] {
        owner.transfer[feeAmount]; // Transfer fee to owner
    }

    // Interaction
    receiver.transfer[transferAmount]; // Transfer amount to
receiver

    emit Transfer[msg.sender, receiver, transferAmount, feeAmount];
}
```

6.2. Implementing Gas-Efficient Loop in Group Transfer

```
function groupTransferWithSeperateAmount[address payable[] memory
recipients, uint256[] memory amounts] external payable {
    require[recipients.length == amounts.length, "Mismatched array
lengths"];
    require[recipients.length <= 100, "Too many recipients"]; //
Limit the number of recipients

    for [uint256 i = 0; i < recipients.length; i++] {
        singleTransfer[recipients[i], amounts[i]];
    }
}
```

In these examples, the singleTransfer function has been modified to follow the Checks-Effects-Interactions pattern, and the groupTransferWithSeperateAmount function includes a limit on the number of recipients to manage gas costs efficiently.

7. Additional Recommendations

7.1. Regular Audits and Testing

- Conduct thorough audits periodically, especially after major updates or changes to the contract.
- Engage in continuous testing, including unit tests, integration tests, and stress tests, to ensure contract robustness under various conditions.

7.2. Keeping Up-to-Date with Ethereum Updates and Security Advisories

- Stay informed about the latest Ethereum platform updates and changes in Solidity versions.
- Monitor security advisories and discussions in the Ethereum community to be aware of new vulnerabilities and best practices.

These proactive measures are crucial in maintaining the security and reliability of smart contracts over time.

8. Conclusion

8.1. Summary of Findings

- The "EthTransferWithFee" smart contract is well-structured, leveraging Solidity 0.8.13's features for secure Ethereum transfers with a fee mechanism.
- Key vulnerabilities identified include potential reentrancy risks, gas limitations due to loops, and concerns around fee management.
- The contract generally adheres to best practices but can benefit from specific security enhancements.

8.2. Final Recommendations

- Implement the Checks-Effects-Interactions pattern to mitigate reentrancy risks.
- Optimize loop operations to handle gas limitations effectively.
- Establish secure and transparent methods for managing fee percentages.
- Regularly audit and test the contract and stay updated with Ethereum developments and security advisories for ongoing contract security.