Shield Security

Smart Contract Security Audit Report

For

MeowMotion

Date Issued: January.22, 2025

Version: v1.0

Confidentiality Level: Public

Contents

1 Abstract

This report was prepared for MeowMotion smart contract to identify issues and vulnerabilities in its smart contract source code. A thorough examination of MeowMotion smart contracts was conducted through timely communication with MeowMotion, static analysis using multiple audit tools and manual auditing of their smart contract source code.

The audit process paid particular attention to the following considerations.

- A thorough review of the smart contract logic flow
- Assessment of the code base to ensure compliance with current best practice and industry standards
- Ensured the contract logic met the client's specifications and intent
- Internal vulnerability scanning tools tested for common risks and writing errors
- Testing smart contracts for common attack vectors
- Test smart contracts for known vulnerability risks
- Conduct a thorough line-by-line manual review of the entire code base

As a result of the security assessment, issues ranging from critical to informational were identified. We recommend that these issues are addressed to ensure a high level of security standards and industry practice. The recommendations we made could have better served the project from a security perspective.

- Enhance general coding practices to improve the structure of the source code.
- Provide more comments for each function to improve readability.
- Provide more transparency of privileged activities once the agreement is in place.

2 Overview

2.1 Project Summary

Project Summary	Project Information
Name	MeowMotion
Start date	January.21, 2025
End date	January.22, 2025
Contract type	NFT
Language	Solidity

2.2 Report HASH

Name	HASH
MeowMotion	https://github.com/MeowMotion/meow_contract/commit/eb3929e50618a1e51c9dd3ff46edb57e007c60a6

3 Project contract details

3.1 Contract Overview

NftPool.sol

The NftPool contract is an NFT sales pool designed to sell two types of non-transferable NFTs (Genesis NFT and Community NFT). The contract enforces a purchase restriction, allowing each address to buy only one type of NFT, and ensures transaction security through a reentrancy guard. It also supports setting and updating the revenue receiver address, toggling sale status, managing fund withdrawals, and updating NFT metadata URIs, primarily operated by the contract owner.

NonTransferableNFT.sol

The contract NonTransferableNFT is a smart contract focused on non-transferable NFTs, with main functions including minting NFTs and managing their metadata. The NFTs in the contract have the following characteristics: non-transferable, any attempt to transfer NFTs will be rejected, ensuring that each NFT always belongs to the initial recipient, suitable for identity, membership or other scenarios requiring uniqueness and attribution; privileged operation, the contract owner can call the mint function to mint NFTs for the specified address, and update the metadata URI of all NFTs through the setURIBase function. In addition, the contract inherits the functions of the standard ERC-721, such as querying NFT owners, querying balances, etc., and expands the management function of the base URI, making it more suitable for scenarios of non-transferable assets.

4 Audit results

4.1 Key messages

ID	Title	Severity	Status
01	Multiple Address Exploitation for NFT Purchase	Low	Ignore
02	Privileged roles	Low	Ignore
03	Unused _reentrancyGuardEntered Function	Informational	Ignore

4.2 Audit details

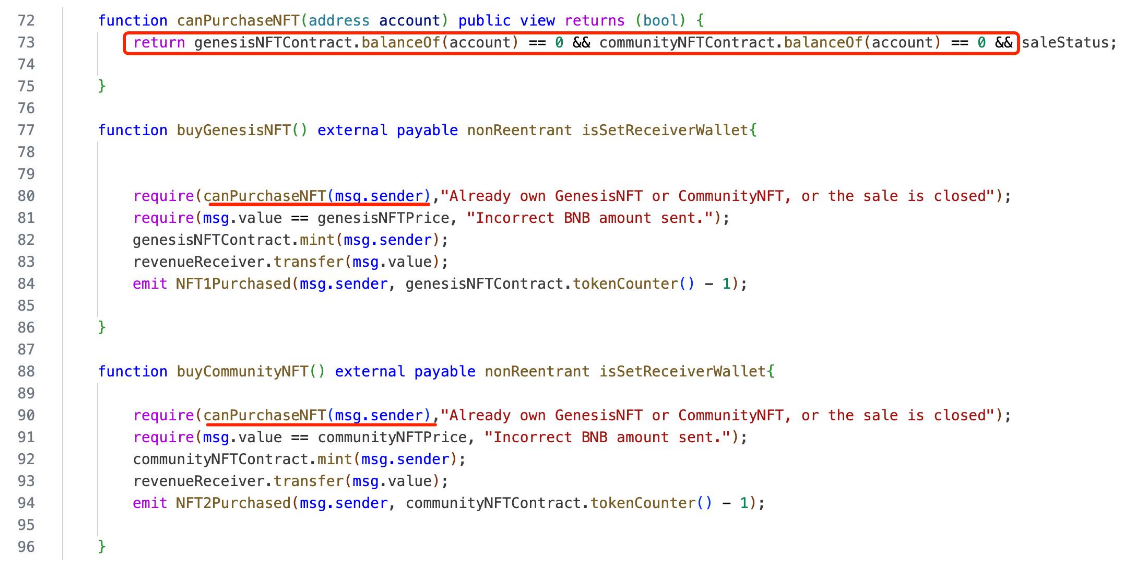
4.2.1 Multiple Address Exploitation for NFT Purchase

ID	Severity	Location	Status
1	Low	NftPool.sol	Ignore

Description

The contract allows a user to purchase one NFT per address, but it does not prevent users from bypassing this restriction by using multiple addresses or transferring NFTs after purchase to buy more. This defeats the purpose of limiting purchases to one NFT per user and can lead to unfair distribution.

Code location:



```
    function canPurchaseNFT(address account) public view returns (bool) {
        return genesisNFTContract.balanceOf(account) == 0 && communityNFTContract.balanceOf(account) == 0 && saleStatus;

    }

    function buyGenesisNFT() external payable nonReentrant isSetReceiverWallet{


        require(canPurchaseNFT(msg.sender),"Already own GenesisNFT or CommunityNFT, or the sale is closed");
        require(msg.value == genesisNFTPrice, "Incorrect BNB amount sent.");
        genesisNFTContract.mint(msg.sender);
        revenueReceiver.transfer(msg.value);
        emit NFT1Purchased(msg.sender, genesisNFTContract.tokenCounter() - 1);

    }

    function buyCommunityNFT() external payable nonReentrant isSetReceiverWallet{

        require(canPurchaseNFT(msg.sender),"Already own GenesisNFT or CommunityNFT, or the sale is closed");
        require(msg.value == communityNFTPrice, "Incorrect BNB amount sent.");
        communityNFTContract.mint(msg.sender);
        revenueReceiver.transfer(msg.value);
        emit NFT2Purchased(msg.sender, communityNFTContract.tokenCounter() - 1);

    }
```

Recommendation

Implement stronger mechanisms to enforce the one-per-user policy, such as whitelist verification.

Status

Ignore.

4.2.2 Privileged roles

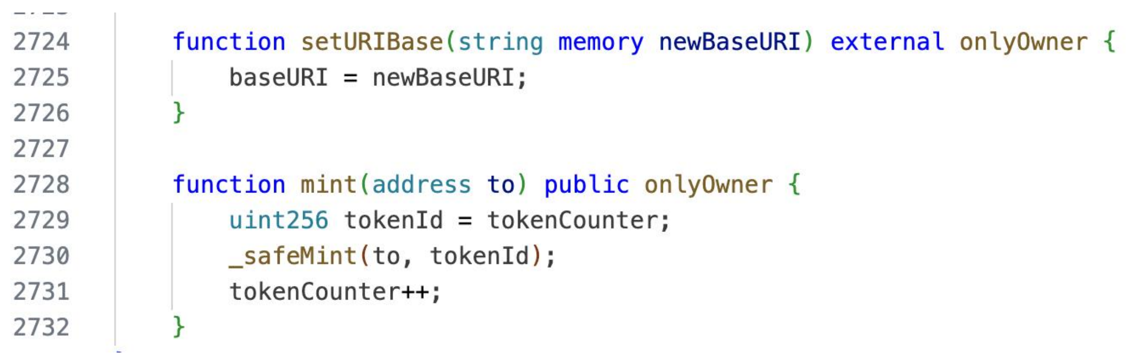
ID	Severity	Location	Status
2	Low	NftPool.sol,NonTransferableNFT.sol	Ignore

Description

The mint function in the NonTransferableNFT contract allows the owner to mint an unlimited number of NFTs. This creates a risk of abuse, where the privileged role could mint excessive NFTs, potentially diluting the value of existing NFTs or exploiting buyers.

The setURIBase function allows the owner to modify the base URI of the NFTs. This could potentially be misused to disrupt the metadata or redirect it to malicious or irrelevant content, undermining user trust.

Code location:



```
    function setURIBase(string memory newBaseURI) external onlyOwner {
        baseURI = newBaseURI;
    }

    function mint(address to) public onlyOwner {
        uint256 tokenId = tokenCounter;
        _safeMint(to, tokenId);
        tokenCounter++;
    }
```

Recommendation

Implement a maximum cap on the total supply of NFTs. Use multi-signatures to manage privileged roles.

Status

Ignore.

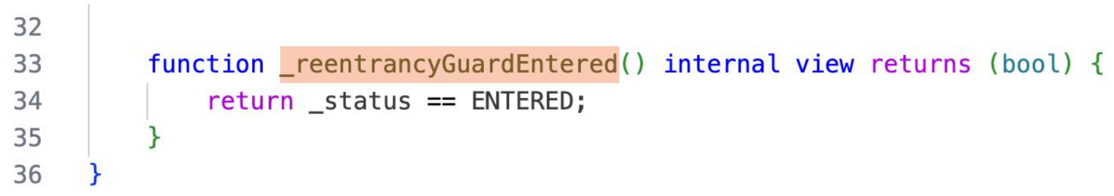
4.2.3 Unused _reentrancyGuardEntered Function

ID	Severity	Location	Status
3	Informational	NftPool.sol	Ignore

Description

The _reentrancyGuardEntered function is defined but never used in the contract. This results in unnecessary storage of unused code, which increases deployment gas costs and can confuse developers reviewing the code.

Code location:



```

    function _reentrancyGuardEntered() internal view returns (bool) {
        return _status == ENTERED;
    }
}
```

Recommendation

Remove the _reentrancyGuardEntered function if it is not needed. This will reduce gas usage during deployment and improve code clarity.

Status

Ignore.

5 Finding Categories

Centralization / Privilege

Centralization / Privilege findings refer to either feature logic or implementation of components that act against the nature of decentralization, such as explicit ownership or specialized access roles in combination with a mechanism to relocate funds.

Gas Optimization

Gas Optimization findings do not affect the functionality of the code but generate different, more optimal EVM opcodes resulting in a reduction on the total gas cost of a transaction.

Mathematical Operations

Mathematical Operation findings relate to mishandling of math formulas, such as overflows, incorrect operations etc.

Logical Issue

Logical Issue findings detail a fault in the logic of the linked code, such as an incorrect notion on how block.timestamp works.

Control Flow

Control Flow findings concern the access control imposed on functions, such as owner-only functions being invoke-able by anyone under certain circumstances.

Volatile Code

Volatile Code findings refer to segments of code that behave unexpectedly on certain edge cases that may result in a vulnerability.

Data Flow

Data Flow findings describe faults in the way data is handled at rest and in memory, such as the result of a struct assignment operation affecting an in-memory struct rather than an in-storage one.

Language Specific

Language Specific findings are issues that would only arise within Solidity, i.e. incorrect usage of private or delete.

Coding Style

Coding Style findings usually do not affect the generated byte-code but rather comment on how to make the codebase more legible and, as a result, easily maintainable.

Inconsistency

Inconsistency findings refer to functions that should seemingly behave similarly yet contain different code, such as a constructor assignment imposing different require statements on the input variables than a setter function.

Magic Numbers

Magic Number findings refer to numeric literals that are expressed in the codebase in their raw format and should otherwise be specified as constant contract variables aiding in their legibility and maintainability.

Compiler Error

Compiler Error findings refer to an error in the structure of the code that renders it impossible to compile using the specified version of the project.

Disclaimer