KittyFi First Flight Audit Report

Lead Auditors:

- Aditya Mishra

Table of Contents

Protocol Summary

KittyFi, a EUR pegged stablecoin based protocol which proactively maintains the user's deposited collateral to earn yield on it via Aave protocol.

With KittyFi, the collateral deposited by user will not just remain in there for backing the KittyCoin but will earn yield on it via Aave protocol.

By utilizing the interest earned on collateral, the protocol will reduce the risk of user getting liquidated by equally allocating the interest earned on collateral to every user in the pool.

KittyCoin

The stable coin of KittyFi protocol which is pegged to EUR and can be minted by supplying collateral and minting via KittyPool.

KittyPool

This smart contract is assigned the role to allow user to deposit collateral and mint KittyCoin from it. The KittyPool contract routes the call to the respective vault for deposit and withdrawal fo collateral which is created for every collateral token used in protocol.
The user is required to main overcollateralization in order to prevent liquidation of their pawsition (I mean position (meows, purrss)).

The pool also handles liquidations, the user gets some percentage reward on the collateral liquidated from the user's vault.

KittyVault

Every collateral token have their own vault deployed via `KittyPool` contract. The vault is responsible for maintaining the collateral deposited by user and supply it to Aave protocol to earn yield on it. The KittyVault when queried with the amount of collateral present in it or collateral of user, then it will return the interest earned total collateral.

Actors

- `User` - Performing deposit and withdrawal of collateral along with minting and burning of KittyCoin
- `Meowntainer` - Responsible for performing executions to supply and withdraw collateral from Aave protocol on KittyVault

Disclaimer

The Aditya Mishra 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.

Executive Summary

Issues found

Severity	Number of issues found
High	4
Medium	3
Low	0
Info	2
Total	9

High

[H-1] Lack of Validation for Token Addresses in `KittyPool::meownufactureKittyVault`

Description: The `KittyPool::meownufactureKittyVault` function does not validate the `_token` and `_priceFeed` addresses, which could lead to the creation of a vault with invalid or malicious addresses.

Impact: Creation of vaults with invalid or malicious addresses could lead to loss of funds or other unexpected behavior.

Proof of Concept: An attacker could potentially call the function with invalid addresses:

```
    // Attacker contract
    contract AttackKittyPool {
        KittyPool public kittyPool;

        constructor(address _kittyPool) {
            kittyPool = KittyPool(_kittyPool);
        }

        function attack() public {
            // Create a vault with invalid addresses
            kittyPool.meownufactureKittyVault(address(0), address(0));
        }
    }
```

Recommended Mitigation: Add validation checks to ensure that addresses are non-zero and valid.

```
    function meownufactureKittyVault(address _token, address _priceFeed) external
onlyMeowntainer {
        require(_token != address(0), "Invalid token address");
        require(_priceFeed != address(0), "Invalid price feed address");
        require(tokenToVault[_token] == address(0),
KittyPool__TokenAlreadyExistsMeeooww());

        address _kittyVault = address(new KittyVault{ salt:
bytes32(abi.encodePacked(ERC20(_token).symbol())) }(_token, address(this),
_priceFeed, i_euroPriceFeed, meowntainer, i_aavePool));

        tokenToVault[_token] = _kittyVault;
        vaults.push(_kittyVault);
    }
```

[H-2] Reentrancy Vulnerability in `KittyPool::whiskdrawMeowllateral`

Description: The `KittyPool::whiskdrawMeowllateral` function calls an external contract and then performs a state-changing operation. This could lead to a reentrancy attack where the external contract calls back into the KittyPool contract before the state change is completed.

Impact: An attacker could drain funds or cause other unexpected behavior by reentering the contract.

Proof of Concept: An attacker could create a malicious KittyVault contract that reenters the KittyPool contract:

Use the below contract as Malicious Vault Contract:-

```
    contract MaliciousKittyVault is KittyVault {
        KittyPool public kittyPool;

        constructor(address _kittyPool) KittyVault(address(0), address(this),
address(0), address(0), address(0), address(0)) {
            kittyPool = KittyPool(_kittyPool);
        }

        function executeWhiskdrawal(address _user, uint256 _cattyNipToWithdraw)
external override {
            kittyPool.whiskdrawMeowllateral(address(this), 1);
        }
    }
```

Use the below contract as Attacker Contract:-

```
    contract AttackKittyPool {
        KittyPool public kittyPool;
        MaliciousKittyVault public maliciousVault;

        constructor(address _kittyPool) {
            kittyPool = KittyPool(_kittyPool);
            maliciousVault = new MaliciousKittyVault(address(kittyPool));
        }
        function attack() public {
            // Register the malicious vault
            kittyPool.meownufactureKittyVault(address(maliciousVault),
address(0));
            // Call whiskdrawMeowllateral to start the reentrancy attack
            kittyPool.whiskdrawMeowllateral(address(maliciousVault), 1);
        }
    }
```

Recommended Mitigation: Use the Checks-Effects-Interactions pattern to update the state before calling external contracts.

```
    function whiskdrawMeowllateral(address _token, uint256 _ameownt) external
tokenExists(_token) {
        require(_hasEnoughMeowllateral(msg.sender),
KittyPool__NotEnoughMeowllateralPurrrr());
        IKittyVault(tokenToVault[_token]).executeWhiskdrawal(msg.sender,
_ameownt);
    }
```

[H-3] Lack of Validation for Token Address in `KittyVault::constructor` function

Description: The constructor does not validate the `_token`, `_kittyPool`, `_priceFeed`, `_euroPriceFeed`, `_meowntainer`, and `_aavePool` addresses, which could lead to setting invalid or malicious addresses.

Impact: Invalid or malicious addresses could lead to loss of funds or other unexpected behavior.

Proof of Concept:

Recommended Mitigation: Add validation checks to ensure that addresses are non-zero and valid.

```
    constructor(
        address _token,
        address _kittyPool,
        address _priceFeed,
        address _euroPriceFeed,
        address _meowntainer,
        address _aavePool
    ) {
        require(_token != address(0), "Invalid token address");
        require(_kittyPool != address(0), "Invalid kitty pool address");
        require(_priceFeed != address(0), "Invalid price feed address");
        require(_euroPriceFeed != address(0), "Invalid euro price feed address");
        require(_meowntainer != address(0), "Invalid meowntainer address");
        require(_aavePool != address(0), "Invalid aave pool address");

        token = _token;
        kittyPool = _kittyPool;
        priceFeed = _priceFeed;
        euroPriceFeed = _euroPriceFeed;
        meowntainer = _meowntainer;
        aavePool = _aavePool;
    }
```

[H-4] Reentrancy Vulnerability in `KittyVault::executeWhiskdrawal`

Description: The `KittyVault::executeWhiskdrawal` function calls an external contract to transfer tokens before updating the state, which could lead to a reentrancy attack.

Impact: An attacker could drain funds or cause other unexpected behavior by reentering the contract.

Proof of Concept: An attacker could create a malicious contract that reenters the KittyVault contract:

```
    contract Malicious {
        KittyVault public vault;

        constructor(address _vault) {
            vault = KittyVault(_vault);
        }

        function attack() public {
            vault.executeWhiskdrawal(address(this), 1);
        }

        function receive() external payable {
            vault.executeWhiskdrawal(address(this), 1);
        }
    }
```

Recommended Mitigation: Use the Checks-Effects-Interactions pattern to update the state before calling external contracts.

```
    function executeWhiskdrawal(address _user, uint256 _cattyNipToWithdraw)
  external onlyKittyPool {
        collateral[_user] -= _cattyNipToWithdraw;
        IERC20(token).transfer(_user, _cattyNipToWithdraw);
    }
```

Medium

[M-1] Lack of Validation for `KittyCoin::mint` and `KittyCoin::burn` Functions

Description: The absence of validation checks for addresses and amounts in the `KittyCoin::mint` and `KittyCoin::burn` functions could allow the KittyPool contract to mint tokens to or burn tokens from unintended or zero addresses, leading to potential token mismanagement or loss.

Impact: Token mismanagement, potential loss or locking of tokens in zero addresses.

Proof of Concept: An attacker could potentially call these functions with zero addresses, leading to loss of tokens:

```
    contract AttackKittyCoin {
        KittyCoin public kittyCoin;

        constructor(address _kittyCoin) {
            kittyCoin = KittyCoin(_kittyCoin);
        }

        function attackMint() public {
            // Mint tokens to the zero address
            kittyCoin.mint(address(0), 1000);
        }

        function attackBurn() public {
            // Burn tokens from the zero address
            kittyCoin.burn(address(0), 1000);
        }
    }
```

Recommended Mitigation: Add validation checks to ensure that addresses are non-zero and amounts are greater than zero.

```
    function mint(address _to, uint256 _amount) external onlyKittyPool {
        require(_to != address(0), "Invalid address");
        require(_amount > 0, "Invalid amount");
        _mint(_to, _amount);
    }

    function burn(address _from, uint256 _amount) external onlyKittyPool {
        require(_from != address(0), "Invalid address");
        require(_amount > 0, "Invalid amount");
        _burn(_from, _amount);
    }
```

[M-2] Private `KittyCoin::pool` variable initialization

Description: The `KittyCoin::pool` address is set only during contract deployment and cannot be changed thereafter. The pool address is set only once during contract initialization. If this address needs to be changed (e.g., due to a key compromise or pool upgrade), there is no way to do so.

Impact: Inability to update the pool address in case of an emergency, leading to potential loss of control over minting and burning capabilities.

Recommended Mitigation: Introduce a function to update the pool address with appropriate access control.

```
    function setPool(address _newPool) external onlyKittyPool {
        require(_newPool != address(0), "Invalid address");
        pool = _newPool;
    }
```

[M-3] Lack of Validation for Addresses in `KittyPool::constructor` function

Description: The constructor does not validate the addresses provided, which could lead to setting invalid or malicious addresses.

Impact: Invalid or malicious addresses could lead to loss of funds or other unexpected behavior.

Recommended Mitigation: Add validation checks to ensure that addresses are non-zero and valid.

```
    constructor(address _meowntainer, address _euroPriceFeed, address aavePool) {
        require(_meowntainer != address(0), "Invalid meowntainer address");
        require(_euroPriceFeed != address(0), "Invalid euro price feed address");
        require(aavePool != address(0), "Invalid aave pool address");
        meowntainer = _meowntainer;
        i_kittyCoin = new KittyCoin(address(this));
        i_euroPriceFeed = _euroPriceFeed;
        i_aavePool = aavePool;
    }
```

Informational

[I-1] Inefficient Storage of `KittyCoin::pool` address

Description: There is the issue of storing the address of the `KittyCoin::pool` in the contract.

Impact: Storing the pool address in a private state variable costs more gas than storing it in an immutable state variable, which would only store it once during deployment.

Recommended Mitigation: Use an immutable state variable for the `pool` address.

```
    address private immutable pool;

    constructor(address _pool) ERC20("Kitty Token", "MEOWDY") {
        pool = _pool;
    }
```

[I-2] Lack of Validation on `_amount` parameter in `KittyCoin::mint` and `KittyCoin::burn` functions

Description: There is a lack of validation that how much amount is being minted and burned. So there is no validation in these functions.

Impact: Potential for minting or burning zero tokens, which may not be desired behavior.

Recommended Mitigation: Add validation to ensure `_amount` is greater than zero.

```
    function mint(address _to, uint256 _amount) external onlyKittyPool {
        require(_amount > 0, "Mint amount must be greater than zero");
        _mint(_to, _amount);
        emit KittyCoinMinted(_to, _amount);
    }

    function burn(address _from, uint256 _amount) external onlyKittyPool {
        require(_amount > 0, "Burn amount must be greater than zero");
        _burn(_from, _amount);
        emit KittyCoinBurned(_from, _amount);
    }
```