FairLaunch, Token, Vault & Workers

Smart Contract Audit Report
Prepared for Meow Finance

Date Issued:	Oct 25, 2021
Project ID:	AUDIT2021021
Version:	v1.0
Confidentiality Level:	Public

inspeX
CYBERSECURITY PROFESSIONAL SERVICE

Public

Report Information

Project ID	AUDIT2021021
Version	v1.0
Client	Meow Finance
Project	FairLaunch, Token, Vault & Workers
Auditor(s)	Suvicha Buakhom Peeraphut Punsuwan
Author	Suvicha Buakhom
Reviewer	Weerawat Pawanawiwat
Confidentiality Level	Public

Version History

Version	Date	Description	Author(s)
1.0	Oct 25, 2021	Full report	Suvicha Buakhom

Contact Information

Company	Inspex
Phone	(+66) 90 888 7186
Telegram	t.me/inspexco
Email	audit@inspex.co

Table of Contents

1. Executive Summary

As requested by Meow Finance, Inspex team conducted an audit to verify the security posture of the FairLaunch, Token, Vault & Workers smart contracts between Sep 29, 2021 and Oct 5, 2021. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of FairLaunch, Token, Vault & Workers smart contracts. Static code analysis, dynamic analysis, and manual review were done in conjunction to identify smart contract vulnerabilities together with technical & business logic flaws that may be exposed to the potential risk of the platform and the ecosystem. Practical recommendations are provided according to each vulnerability found and should be followed to remediate the issue.

1.1. Audit Result

In the initial audit, Inspex found 3 high, 4 medium, 6 low, 2 very low, and 5 info-severity issues. With the project team's prompt response, 3 high, 4 medium, 5 low, 2 very low and 5 info-severity issues were resolved in the reassessment, while 1 low-severity issue was acknowledged by the team. Therefore, Inspex trusts that FairLaunch, Token, Vault & Workers smart contracts have sufficient protections to be safe for public use. However, in the long run, Inspex suggests resolving all issues found in this report.

1.2. Disclaimer

This security audit is not produced to supplant any other type of assessment and does not guarantee the discovery of all security vulnerabilities within the scope of the assessment. However, we warrant that this audit is conducted with goodwill, professional approach, and competence. Since an assessment from one single party cannot be confirmed to cover all possible issues within the smart contract(s), Inspex suggests conducting multiple independent assessments to minimize the risks. Lastly, nothing contained in this audit report should be considered as investment advice.

2. Project Overview

2.1. Project Introduction

Meow Finance is a DeFi leveraged yield farming and lending protocol on the Fantom chain. They aim to provide users an experience of yield farming with their desired leverage, built on the Fantom framework where it builds and connects Ethereum-compatible blockchain networks.

Fairlaunch is a mechanism to distribute $MEOW to the users who deposit or stake tokens to the platform.

Vault & Workers are components of lending, leveraged yield farming, auto compounding, and farming position managing. On the Vault, users can lend their tokens and open leveraged yield farming positions. Workers use the rewards obtained from farming for reinvestment and managing the users' opened positions.

Scope Information:

Project Name	FairLaunch, Token, Vault & Workers
Website	https://meowfinance.org/
Smart Contract Type	Ethereum Smart Contract
Chain	Fantom Opera
Programming Language	Solidity

Audit Information:

Audit Method	Whitebox
Audit Date	Sep 29, 2021 - Oct 5, 2021
Reassessment Date	Oct 18, 2021

The audit method can be categorized into two types depending on the assessment targets provided:

1. **Whitebox**: The complete source code of the smart contracts are provided for the assessment.
2. **Blackbox**: Only the bytecodes of the smart contracts are provided for the assessment.

2.2. Scope

The following smart contracts were audited and reassessed by Inspex in detail:

Initial Audit: (Commit: 4a4f13efaf5e5fbed74c0ed23b665751e655d715)

Contract	Location (URL)
Vault	https://github.com/meow-finance/Meow-Finance/blob/4a4f13efaf/contracts/protocol/Vault.sol
TripleSlopeModel	https://github.com/meow-finance/Meow-Finance/tree/4a4f13efaf/contracts/protocol/interest-models
MeowMining	https://github.com/meow-finance/Meow-Finance/blob/4a4f13efaf/contracts/token/MeowMining.sol
SpookyswapWorker	https://github.com/meow-finance/Meow-Finance/blob/4a4f13efaf/contracts/protocol/workers/SpookyswapWorker.sol
MeowToken	https://github.com/meow-finance/Meow-Finance/blob/4a4f13efaf/contracts/token/MeowToken.sol
FeeDistribute	https://github.com/meow-finance/Meow-Finance/blob/4a4f13efaf/contracts/token/FeeDistribute.sol
DevelopmentFund	https://github.com/meow-finance/Meow-Finance/blob/4a4f13efaf/contracts/token/DevelopmentFund.sol

Reassessment: (Commit: 0912b0099114939c3452117c1a25de82cfb6cd75)

Contract	Location (URL)
Vault	https://github.com/meow-finance/Meow-Finance/blob/0912b00991/contracts/protocol/Vault.sol
TripleSlopeModel	https://github.com/meow-finance/Meow-Finance/tree/0912b00991/contracts/protocol/interest-models
MeowMining	https://github.com/meow-finance/Meow-Finance/blob/0912b00991/contracts/token/MeowMining.sol
SpookyswapWorker	https://github.com/meow-finance/Meow-Finance/blob/0912b00991/contracts/protocol/workers/SpookyswapWorker.sol
MeowToken	https://github.com/meow-finance/Meow-Finance/blob/0912b00991/contracts/token/MeowToken.sol
FeeDistribute	https://github.com/meow-finance/Meow-Finance/blob/0912b00991/contracts/token/FeeDistribute.sol
DevelopmentFund	https://github.com/meow-finance/Meow-Finance/blob/0912b00991/contracts/token/DevelopmentFund.sol

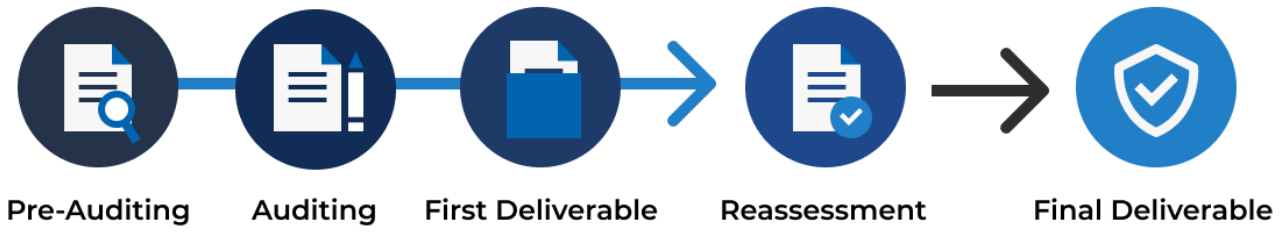
The assessment scope covers only the in-scope smart contracts and the smart contracts that they are inherited from.

The `setSpookyFee()` function has been added in the reassessment commit, and is outside of the audit scope. The Meow Finance team has clarified that this function is used to change the swapping fee when the fee rate on the SpookySwap platform changes.

3. Methodology

Inspex conducts the following procedure to enhance the security level of our clients' smart contracts:

1. **Pre-Auditing**: Getting to understand the overall operations of the related smart contracts, checking for readiness, and preparing for the auditing
2. **Auditing**: Inspecting the smart contracts using automated analysis tools and manual analysis by a team of professionals
3. **First Deliverable and Consulting**: Delivering a preliminary report on the findings with suggestions on how to remediate those issues and providing consultation
4. **Reassessment**: Verifying the status of the issues and whether there are any other complications in the fixes applied
5. **Final Deliverable**: Providing a full report with the detailed status of each issue



3.1. Test Categories

Inspex smart contract auditing methodology consists of both automated testing with scanning tools and manual testing by experienced testers. We have categorized the tests into 3 categories as follows:

1. **General Smart Contract Vulnerability (General)** - Smart contracts are analyzed automatically using static code analysis tools for general smart contract coding bugs, which are then verified manually to remove all false positives generated.
2. **Advanced Smart Contract Vulnerability (Advanced)** - The workflow, logic, and the actual behavior of the smart contracts are manually analyzed in-depth to determine any flaws that can cause technical or business damage to the smart contracts or the users of the smart contracts.
3. **Smart Contract Best Practice (Best Practice)** - The code of smart contracts is then analyzed from the development perspective, providing suggestions to improve the overall code quality using standardized best practices.

3.2. Audit Items

The following audit items were checked during the auditing activity.

General
Reentrancy Attack
Integer Overflows and Underflows
Unchecked Return Values for Low-Level Calls
Bad Randomness
Transaction Ordering Dependence
Time Manipulation
Short Address Attack
Outdated Compiler Version
Use of Known Vulnerable Component
Deprecated Solidity Features
Use of Deprecated Component
Loop with High Gas Consumption
Unauthorized Self-destruct
Redundant Fallback Function
Insufficient Logging for Privileged Functions
Invoking of Unreliable Smart Contract
Advanced
Business Logic Flaw
Ownership Takeover
Broken Access Control
Broken Authentication
Use of Upgradable Contract Design
Improper Kill-Switch Mechanism

Improper Front-end Integration
Insecure Smart Contract Initiation
Denial of Service
Improper Oracle Usage
Memory Corruption
Best Practice
Use of Variadic Byte Array
Implicit Compiler Version
Implicit Visibility Level
Implicit Type Inference
Function Declaration Inconsistency
Token API Violation
Best Practices Violation

3.3. Risk Rating

OWASP Risk Rating Methodology[1] is used to determine the severity of each issue with the following criteria:

- **Likelihood**: a measure of how likely this vulnerability is to be uncovered and exploited by an attacker.
- **Impact**: a measure of the damage caused by a successful attack

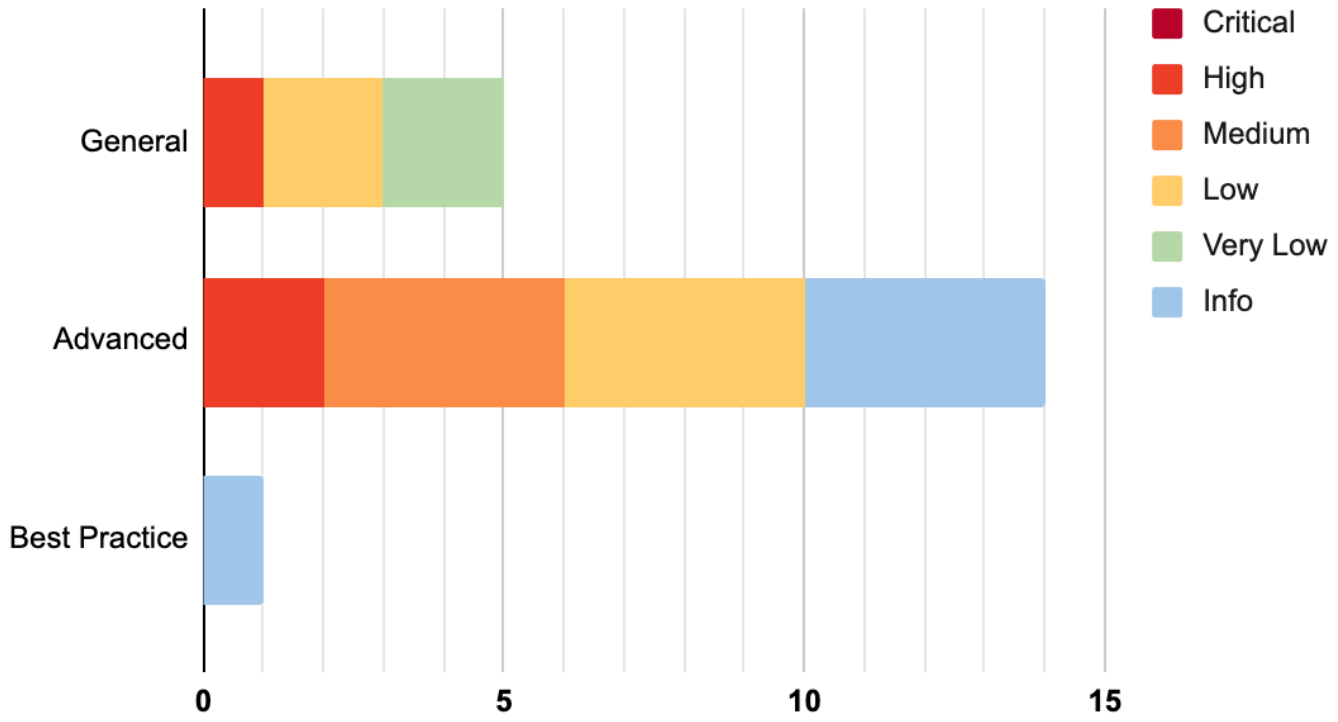
Both likelihood and impact can be categorized into three levels: **Low**, **Medium**, and **High**.

Severity is the overall risk of the issue. It can be categorized into five levels: **Very Low**, **Low**, **Medium**, **High**, and **Critical**. It is calculated from the combination of likelihood and impact factors using the matrix below. The severity of findings with no likelihood or impact would be categorized as **Info**.

Impact \ Likelihood	Low	Medium	High
Low	**Very Low**	**Low**	**Medium**
Medium	**Low**	**Medium**	**High**
High	**Medium**	**High**	**Critical**

4. Summary of Findings

From the assessments, Inspex has found 20 issues in three categories. The following chart shows the number of the issues categorized into three categories: **General**, **Advanced**, and **Best Practice**.



The statuses of the issues are defined as follows:

Status	Description
Resolved	The issue has been resolved and has no further complications.
Resolved *	The issue has been resolved with mitigations and clarifications. For the clarification or mitigation detail, please refer to Chapter 5.
Acknowledged	The issue's risk has been acknowledged and accepted.
No Security Impact	The best practice recommendation has been acknowledged.

The information and status of each issue can be found in the following table:

ID	Title	Category	Severity	Status
IDX-001	Denial of Service in Beneficiary Mechanism	Advanced	High	Resolved
IDX-002	Use of Upgradable Contract Design	Advanced	High	Resolved *
IDX-003	Centralized Control of State Variable	General	High	Resolved *
IDX-004	Improper Reward Calculation in MeowMining	Advanced	Medium	Resolved
IDX-005	Improper Reward Calculation in FeeDistribute	Advanced	Medium	Resolved
IDX-006	Improper Compliance to the Tokenomics	Advanced	Medium	Resolved
IDX-007	Denial of Service on Minting Cap Exceeding	Advanced	Medium	Resolved
IDX-008	Improper Delegation Handling in Token Burning	Advanced	Low	Resolved
IDX-009	Design Flaw in massUpdatePool() Function	General	Low	Acknowledged
IDX-010	Transaction Ordering Dependence	General	Low	Resolved
IDX-011	Missing Input Validation (maxReinvestBountyBps)	Advanced	Low	Resolved
IDX-012	Denial of Service in reinvest() Function	Advanced	Low	Resolved
IDX-013	Missing Input Validation of preShare and lockShare Values	Advanced	Low	Resolved
IDX-014	Outdated Compiler Version	General	Very Low	Resolved
IDX-015	Insufficient Logging for Privileged Functions	General	Very Low	Resolved
IDX-016	Unavailability of manualMint() Function	Advanced	Info	Resolved
IDX-017	Improper Access Control for Development Fund Locking	Advanced	Info	Resolved
IDX-018	Improper Access Control for burnFrom() Function	Advanced	Info	Resolved
IDX-019	Unsupported Design for Deflationary Token	Advanced	Info	Resolved
IDX-020	Improper Function Visibility	Best Practice	Info	Resolved

* The mitigations or clarifications by Meow Finance can be found in Chapter 5.

5. Detailed Findings Information

5.1. Denial of Service in Beneficiary Mechanism

ID	IDX-001
Target	MeowMining
Category	Advanced Smart Contract Vulnerability
CWE	CWE-755: Improper Handling of Exceptional Conditions
Risk	**Severity:** High **Impact:** High The victim won't be able to execute the `deposit()` function of the `MeowMining` contract, causing disruption of service and loss of reputation to the platform. **Likelihood:** Medium This attack can be done by anyone to any address without any prior deposit; however, there is no direct benefit for the attacker, resulting in low motivation for the attack.
Status	Resolved Meow Finance team has resolved this issue by editing the `withdraw()` function as suggested in commit `15137b093aab2fa27cc00a459058a52108333a51`.

5.1.1. Description

In the `MeowMining` contract, users can deposit tokens specified in each pool to gain $MEOW reward using the `deposit()` function. The `_for` variable in the function can be controlled by the users, allowing the deposit by one address for another beneficiary address to gain the reward. The first address that deposits for each `_for` address will be set in the `user.fundedBy` in line 199, preventing others from depositing or withdrawing for that beneficiary due to the condition in line 195.

MeowMining.sol

```
188 function deposit(
189     address _for,
190     uint256 _pid,
191     uint256 _amount
192 ) external nonReentrant {
193     PoolInfo storage pool = poolInfo[_pid];
194     UserInfo storage user = userInfo[_pid][_for];
195     if (user.fundedBy != address(0)) require(user.fundedBy == msg.sender,
    "MeowMining::deposit:: bad sof.");
196     require(pool.stakeToken != address(0), "MeowMining::deposit:: not accept
    deposit.");
197     updatePool(_pid);
```

```
198     if (user.amount > 0) _harvest(_for, _pid);
199     if (user.fundedBy == address(0)) user.fundedBy = msg.sender;
200     IERC20(pool.stakeToken).safeTransferFrom(address(msg.sender),
address(this), _amount);
201     user.amount = user.amount.add(_amount);
202     user.rewardDebt =
user.amount.mul(pool.accMeowPerShare).div(ACC_MEOW_PRECISION);
203     emit Deposit(msg.sender, _pid, _amount);
204 }
```

This behavior can be abused by others to disrupt the use of the smart contract. Malicious actors can perform deposits with 0 `amount` for another `_for` address without prior any deposit, preventing that `_for` address from being used by the actual owner.

5.1.2. Remediation

Inspex suggests allowing the `_for` address to perform withdrawal to return the funds to the `fundedBy` address and set the `fundedBy` to `address(0)` when `user.amount` is 0, for example:

MeowMining.sol

```
219 function _withdraw(
220     address _for,
221     uint256 _pid,
222     uint256 _amount
223 ) internal {
224     PoolInfo storage pool = poolInfo[_pid];
225     UserInfo storage user = userInfo[_pid][_for];
226     require(user.fundedBy == msg.sender || msg.sender == _for,
"MeowMining::withdraw:: only funder.");
227     require(user.amount >= _amount, "MeowMining::withdraw:: not good.");
228     updatePool(_pid);
229     _harvest(_for, _pid);
230     user.amount = user.amount.sub(_amount);
231     user.rewardDebt =
user.amount.mul(pool.accMeowPerShare).div(ACC_MEOW_PRECISION);
232     if (user.amount == 0) user.fundedBy = address(0);
233     if (pool.stakeToken != address(0)) {
234         IERC20(pool.stakeToken).safeTransfer(address(user.fundedBy), _amount);
235     }
236     emit Withdraw(user.fundedBy, _pid, user.amount);
237 }
```

5.2. Use of Upgradable Contract Design

ID	IDX-002
Target	Vault SpookyswapWorker
Category	Advanced Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity: High** **Impact: High** The logic of the affected contracts can be arbitrarily changed. This allows the proxy owner to perform malicious actions e.g., stealing the user funds anytime they want. **Likelihood: Medium** This action can be performed by the proxy owner without any restriction.
Status	Resolved * Meow Finance team has confirmed that the team will mitigate this issue by implementing the timelock mechanism when deploying the smart contracts to mainnet. The users will be able to monitor the timelock for the upgrade of the contract and act accordingly if it is being misused. At the time of reassessment, the contracts are not deployed yet, so the use of timelock is not confirmed. For the platform users, please verify that the timelock is properly deployed before using this platform.

5.2.1. Description

Smart contracts are designed to be used as agreements that cannot be changed forever. When a smart contract is upgraded, the agreement can be changed from what was previously agreed upon.

As the `Vault` and the `SpookyswapWorker` smart contracts are upgradable, the logic of them could be modified by the owner anytime, making the smart contracts untrustworthy.

5.2.2. Remediation

Inspex suggests deploying the contracts without the proxy pattern or any solution that can make the smart contracts upgradable.

However, if upgradability is needed, Inspex suggests mitigating this issue by implementing a timelock mechanism with a sufficient length of time to delay the changes. This allows the platform users to monitor the timelock and be notified of the potential changes being done on the smart contracts.

5.3. Centralized Control of State Variable

ID	IDX-003
Target	Vault TripleSlopeModel MeowMining SpookyswapWorker MeowToken FeeDistribute DevelopmentFund
Category	General Smart Contract Vulnerability
CWE	CWE-710: Improper Adherence to Coding Standard
Risk	**Severity: High** **Impact: High** The controlling authorities can change the critical state variables to gain additional profit. Thus, it is unfair to the other users and can cause significant monetary loss to the users. **Likelihood: Medium** There is nothing to restrict the changes from being done; however, these actions can only be performed by the contract owner.
Status	Resolved * Meow Finance team has confirmed that the team will implement the timelock mechanism when deploying the smart contracts to mainnet. The users will be able to monitor the timelock for the execution of critical functions and act accordingly if they are being misused. At the time of the reassessment, the contracts are not deployed yet, so the use of timelock is not confirmed. For the platform users, please verify that the timelock is properly deployed before using this platform.

5.3.1. Description

Critical state variables can be updated at any time by the controlling authorities. Changes in these variables can cause impacts to the users, so the users should accept or be notified before these changes are effective.

However, as the contract is not yet deployed, there is potentially no constraint to prevent the authorities from modifying these variables without notifying the users.

The controllable privileged state update functions are as follows:

File	Contract	Function	Modifier
Vault.sol (L:454)	Vault	updateConfig()	onlyOwner
Vault.sol (L:460)	Vault	updateDebtToken()	onlyOwner
Vault.sol (L:471)	Vault	setMeowMiningPoolId()	onlyOwner
Vault.sol (L:479)	Vault	withdrawReserve()	onlyOwner
Vault.sol (L:486)	Vault	reduceReserve()	onlyOwner
TripleSlopeModel.sol (L:29)	TripleSlopeModel	setParams()	onlyOwner
MeowMining.sol (L:107)	MeowMining	setMeowPerSecond()	onlyOwner
MeowMining.sol (L:113)	MeowMining	addPool()	onlyOwner
MeowMining.sol (L:126)	MeowMining	setPool()	onlyOwner
MeowMining.sol (L:140)	MeowMining	manualMint()	onlyOwner
SpookyswapWorker.sol (L:288)	SpookyswapWorker	setReinvestBountyBps()	onlyOwner
SpookyswapWorker.sol (L:298)	SpookyswapWorker	setMaxReinvestBountyBps()	onlyOwner
SpookyswapWorker.sol (L:309)	SpookyswapWorker	setStrategyOk()	onlyOwner
SpookyswapWorker.sol (L:319)	SpookyswapWorker	setReinvestorOk()	onlyOwner
SpookyswapWorker.sol (L:329)	SpookyswapWorker	setCriticalStrategies()	onlyOwner
FeeDistribute.sol (L:52)	FeeDistribute	setParams()	onlyOwner
FeeDistribute.sol (L:62)	FeeDistribute	addPool()	onlyOwner

5.3.2. Remediation

In the ideal case, the critical state variables should not be modifiable to keep the integrity of the smart contract. However, if modifications are needed, Inspex suggests limiting the use of these functions via the following options:

- Implementing community-run governance to control the use of these functions
- Using a `Timelock` contract to delay the changes for a sufficient amount of time

5.4. Improper Reward Calculation in MeowMining

ID	IDX-004
Target	MeowMining
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** Medium The reward of the pool that has the same staking token as the reward token will be slightly lower than what it should be, resulting in monetary loss for the users and loss of reputation for the platform. **Likelihood:** Medium It is likely that the pool with the same staking token as the reward token will be added by the contract owner.
Status	Resolved Meow Finance team has resolved this issue by checking the value of `_stakeToken` as suggested in commit `15137b093aab2fa27cc00a459058a52108333a51`.

5.4.1. Description

In the `MeowMining` contract, a new staking pool can be added using the `addPool()` function. The staking token for the new pool is defined using the `_stakeToken` variable; however, there is no additional checking whether the `_stakeToken` is the same as the reward token ($MEOW) or not.

MeowMining.sol

```
113 function addPool(uint256 _allocPoint, address _stakeToken) external onlyOwner {
114     massUpdatePools();
115     require(_stakeToken != address(0), "MeowMining::addPool:: not ZERO
    address.");
116     require(!isPoolExist[_stakeToken], "MeowMining::addPool:: stakeToken
    duplicate.");
117     uint256 lastRewardTime = block.timestamp > startTime ? block.timestamp :
    startTime;
118     totalAllocPoint = totalAllocPoint.add(_allocPoint);
119     poolInfo.push(
120         PoolInfo({ stakeToken: _stakeToken, allocPoint: _allocPoint,
    lastRewardTime: lastRewardTime, accMeowPerShare: 0 })
121     );
122     isPoolExist[_stakeToken] = true;
123 }
```

When the `_stakeToken` is the same token as $MEOW, the reward calculation for that pool in the `updatePool()` function can be incorrect.

This is because the current balance of the `_stakeToken` in the contract is used in the calculation of the reward.

Since the `_stakeToken` is the same token as the reward, the reward minted to the contract will inflate the value of `stakeTokenSupply`, causing the reward of that pool to be less than what it should be.

MeowMining.sol

```
168 function updatePool(uint256 _pid) public {
169     PoolInfo storage pool = poolInfo[_pid];
170     if (block.timestamp > pool.lastRewardTime) {
171         uint256 stakeTokenSupply = IERC20(pool.stakeToken)
    .balanceOf(address(this));
172         if (stakeTokenSupply > 0 && totalAllocPoint > 0) {
173             uint256 time = block.timestamp.sub(pool.lastRewardTime);
174             uint256 meowReward =
    time.mul(meowPerSecond).mul(pool.allocPoint).div(totalAllocPoint);
175             // Every 11.4286 Meow minted will mint 1 Meow for dev, come from
    80/7 = 11.4286 use 10,000 to avoid floating.
176             uint256 devfund = meowReward.mul(10000).div(114286);
177             meow.mint(address(this), devfund);
178             meow.mint(address(this), meowReward);
179             safeMeowTransfer(devaddr, devfund.mul(preShare).div(10000));
180             developmentFund.lock(devfund.mul(lockShare).div(10000));
181             pool.accMeowPerShare = pool.accMeowPerShare.add(meowReward
    .mul(ACC_MEOW_PRECISION).div(stakeTokenSupply));
182         }
183         pool.lastRewardTime = block.timestamp;
184     }
185 }
```

5.4.2. Remediation

Inspex suggests checking the value of the `_stakeToken` in the `addPool()` function to prevent the pool with the same staking token as the reward token from being added, for example:

MeowMining.sol

```
113 function addPool(uint256 _allocPoint, address _stakeToken) external onlyOwner {
114     massUpdatePools();
115     require(_stakeToken != address(0), "MeowMining::addPool:: not ZERO
    address.");
116     require(_stakeToken != meow, "MeowMining::addPool:: the _stakeToken is
    meow.");
117     require(!isPoolExist[_stakeToken], "MeowMining::addPool:: stakeToken
```

```
duplicate.");
118     uint256 lastRewardTime = block.timestamp > startTime ? block.timestamp :
startTime;
119     totalAllocPoint = totalAllocPoint.add(_allocPoint);
120     poolInfo.push(
121         PoolInfo({ stakeToken: _stakeToken, allocPoint: _allocPoint,
lastRewardTime: lastRewardTime, accMeowPerShare: 0 })
122     );
123     isPoolExist[_stakeToken] = true;
124 }
```

However, if the pool with the same staking token as the reward token is required, Inspex suggests minting the reward token to another contract to prevent the amount of the staked token from being mixed up with the reward token, or store the amount of the token staked to use in the reward calculation.

5.5. Improper Reward Calculation in FeeDistribute

ID	IDX-005
Target	FeeDistribute
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** Medium The reward of the pool that has the same staking token as the reward token will be slightly higher than what it should be, so not all users will be able to claim the reward or withdraw their funds, resulting in monetary loss for some users and loss of reputation for the platform. **Likelihood:** Medium It is likely that the pool with the same staking token as the reward token will be added by the contract owner.
Status	Resolved Meow Finance team has resolved this issue by checking the value of the `_rewardToken` as suggested in commit `15137b093aab2fa27cc00a459058a52108333a51`.

5.5.1. Description

In the `FeeDistribute` contract, a new staking pool can be added using the `addPool()` function. The reward token for the new pool is defined using the `_rewardToken` variable; however, there is no additional checking whether the `_rewardToken` is already used as `_stakeToken` or not.

FeeDistribute.sol

```
1  function addPool(address _stakeToken, address _rewardToken) external onlyOwner
   {
2      massUpdatePools();
3      require(_stakeToken != address(0), "FeeDistribute::addPool:: not ZERO
   address.");
4      require(!isPoolExist[_rewardToken], "FeeDistribute::addPool:: pool
   exist.");
5      poolInfo.push(
6          PoolInfo({
7              stakeToken: _stakeToken,
8              rewardToken: _rewardToken,
9              depositedAmount: 0,
10             latestRewardAmount: 0,
11             totalRewardAmount: 0,
12             rewardPerShare: 0
```

```
13          })
14      );
15      isPoolExist[_rewardToken] = true;
16  }
```

When the `_rewardToken` is already used as `_stakeToken`, the reward calculation for that pool in the `updatePool()` function can be incorrect.

This is because the current balance of the `_rewardToken` in the contract is used in the calculation of the reward.

Since the `_rewardToken` is already used as `_stakeToken`, the token staked to the contract will inflate the value of `_rewardBalance`, causing the reward of that pool to be more than what it should be.

FeeDistribute.sol

```
88  function updatePool(uint256 _pid) public {
89      PoolInfo storage pool = poolInfo[_pid];
90      uint256 _rewardBalance = IERC20(pool.rewardToken).balanceOf(address(this));
91      uint256 _pendingReward = _rewardBalance.sub(pool.latestRewardAmount);
92      uint256 _totalDeposited = pool.depositedAmount;
93
94      if (_pendingReward != 0 && _totalDeposited != 0) {
95          uint256 _pendingRewardPerShare = _pendingReward.mul(PRECISION)
    .div(_totalDeposited);
96          pool.totalRewardAmount = pool.totalRewardAmount.add(_pendingReward);
97          pool.latestRewardAmount = _rewardBalance;
98          pool.rewardPerShare = pool.rewardPerShare.add(_pendingRewardPerShare);
99      }
100 }
```

With the inflated reward, some users may not be able to claim their reward or withdraw their funds from the contract.

5.5.2. Remediation

Inspex suggests checking the value of the `_rewardToken` in the `addPool()` function to prevent the pool with the same staking token as the reward token from being added, for example:

FeeDistribute.sol

```
62  mapping(address => bool) public isStakeToken;
63
64  function addPool(address _stakeToken, address _rewardToken) external onlyOwner
    {
65      massUpdatePools();
66      require(_stakeToken != address(0), "FeeDistribute::addPool:: not ZERO
    address.");
```

```
67      require(!isPoolExist[_rewardToken], "FeeDistribute::addPool:: pool
    exist.");
68      require(!isStakeToken[_rewardToken], "FeeDistribute::addPool:: reward token
    is already used as stake token.");
69      require(!isPoolExist[_stakeToken], "FeeDistribute::addPool:: stake token is
    already used as reward token");
70      require(_stakeToken != _rewardToken, "FeeDistribute::addPool:: _stakeToken
    token same as _rewardtoken");
71      poolInfo.push(
72          PoolInfo({
73              stakeToken: _stakeToken,
74              rewardToken: _rewardToken,
75              depositedAmount: 0,
76              latestRewardAmount: 0,
77              totalRewardAmount: 0,
78              rewardPerShare: 0
79          })
80      );
81      isPoolExist[_rewardToken] = true;
82      isStakeToken[_stakeToken] = true;
83  }
```

However, if the pool with the same staking token as the reward token is required, Inspex suggests storing the reward token in another contract to prevent the amount of the staked token from being mixed up with the reward token.

5.6. Improper Compliance to the Tokenomics

ID	IDX-006
Target	MeowToken MeowMining
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** Medium The $MEOW token allocated for the distribution in the `MeowMining` contract can be decreased due to the use of the `manualMint()` function, making it different from the tokenomics announced to the users. The reward distribution period will end early and cause the users to earn less reward than they should. This can result in monetary loss for the users and reputation damage for the platform. **Likelihood:** Medium Only the contract owner can use the `manualMint()` function, but there is no restriction to prevent the owner from using it.
Status	Resolved Meow Finance team has resolved this issue by removing the manual minting functionality as suggested in commit `15137b093aab2fa27cc00a459058a52108333a51`.

5.6.1. Description

The $MEOW has a limit of 250m maximum supply, and it is separated into 3 portions as follows:

1. `meowMining` is the portion that is reserved for the distribution in the `MeowMining` contract, allocated as 80% of max supply.
2. `reserve` is the portion that is pre-minted for the cost of the platform, allocated as 13% of max supply.

Portions 1 and 2 are defined in the `MeowToken` contract at lines 13 and 15.

MeowToken.sol

```
10 // MaxTotalSupply 250m.
11 uint256 private constant CAP = 250000000e18;
12 // Meow mining 200m (80% of 250m).
13 uint256 public meowMining = 200000000e18;
14 // Meow reserve 32.5m (13% of 250m).
15 uint256 public reserve = 32500000e18;
```

3. `devfund` is the portion that is reserved for the development fund, allocated as 7% of max supply. This portion is defined in the `MeowMining` contract at line 176.

MeowMining.sol

```
168 function updatePool(uint256 _pid) public {
169     PoolInfo storage pool = poolInfo[_pid];
170     if (block.timestamp > pool.lastRewardTime) {
171         uint256 stakeTokenSupply =
    IERC20(pool.stakeToken).balanceOf(address(this));
172         if (stakeTokenSupply > 0 && totalAllocPoint > 0) {
173             uint256 time = block.timestamp.sub(pool.lastRewardTime);
174             uint256 meowReward =
    time.mul(meowPerSecond).mul(pool.allocPoint).div(totalAllocPoint);
175             // Every 11.4286 Meow minted will mint 1 Meow for dev, come from
    80/7 = 11.4286 use 10,000 to avoid floating.
176             uint256 devfund = meowReward.mul(10000).div(114286);
177             meow.mint(address(this), devfund);
178             meow.mint(address(this), meowReward);
179             safeMeowTransfer(devaddr, devfund.mul(preShare).div(10000));
180             developmentFund.lock(devfund.mul(lockShare).div(10000));
181             pool.accMeowPerShare =
    pool.accMeowPerShare.add(meowReward.mul(ACC_MEOW_PRECISION).div(stakeTokenSuppl
    y));
182         }
183         pool.lastRewardTime = block.timestamp;
184     }
185 }
```

In addition to the predetermined proportions, tokens can be generated in addition to the specified proportions by manual minting.

MeowToken.sol

```
140 function manualMint(address _to, uint256 _amount) public onlyOwner {
141     require(block.timestamp >= manualMintAllowedAfter, "MeowToken::manualMint::
    manualMint not allowed yet.");
142     require(_amount <= (canManualMint()), "MeowToken::manualMint:: manual mint
    limit exceeded.");
143     manualMintAllowedAfter = block.timestamp.add(minimumTimeBetweenManualMint);
144     manualMinted = manualMinted.add(_amount);
145     mint(_to, _amount);
146 }
```

The amount of tokens that comes from manual minting will be deducted from the `meowMining` portion.

MeowToken.sol

```
35 function canManualMint() public view returns (uint256) {
36     uint256 miningMinted = totalSupply().sub(reserve); // Totalsupply =
    MeowMining + DevFund + reserve.
37     // Every 11.4286 Meow minted will mint 1 Meow for dev, come from 80/7 =
```

```
11.4286 use 10,000 to avoid floating.
38      uint256 devFund = miningMinted.mul(10000).div(114286);
39      return (uint256(200000000e18).sub((miningMinted).sub(devFund))).div(5); //
20% of (MeowMining - DevFund)
40  }
```

When the amount allocated for `meowMining` portion is reduced, the duration of token distribution will also be reduced, causing the user to earn less reward than they should without complying to the tokenomics.

5.6.2. Remediation

Inspex suggests removing the manual minting functionality or redesigning the token allocation to define a clear portion for manual minting.

5.7. Denial of Service on Minting Cap Exceeding

ID	IDX-007
Target	MeowMining
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** Medium Multiple functions of the `MeowMining` contract will be unusable from the failed token minting, disrupting the availability of the service. The users can withdraw their funds using the `emergencyWithdraw()` function, but the pending reward will be discarded. **Likelihood:** Medium It is likely that $MEOW released from the `MeowMining` contract will eventually reach the cap.
Status	Resolved Meow Finance team has resolved this issue by modifying the `MeowMining` contract to handle the case when the cap is filled as suggested in commit `15137b093aab2fa27cc00a459058a52108333a51`.

5.7.1. Description

The `updatePool()` function in the `MeowMining` contract is used to calculate and distribute the reward to the users. The $MEOW reward is minted to `MeowMining` contract using the `mint()` function at line 177-178.

MeowMining.sol

```
168 function updatePool(uint256 _pid) public {
169     PoolInfo storage pool = poolInfo[_pid];
170     if (block.timestamp > pool.lastRewardTime) {
171         uint256 stakeTokenSupply =
    IERC20(pool.stakeToken).balanceOf(address(this));
172         if (stakeTokenSupply > 0 && totalAllocPoint > 0) {
173             uint256 time = block.timestamp.sub(pool.lastRewardTime);
174             uint256 meowReward = time.mul(meowPerSecond).mul(pool.allocPoint)
    .div(totalAllocPoint);
175             // Every 11.4286 Meow minted will mint 1 Meow for dev, come from
    80/7 = 11.4286 use 10,000 to avoid floating.
176             uint256 devfund = meowReward.mul(10000).div(114286);
177             meow.mint(address(this), devfund);
178             meow.mint(address(this), meowReward);
179             safeMeowTransfer(devaddr, devfund.mul(preShare).div(10000));
180             developmentFund.lock(devfund.mul(lockShare).div(10000));
```

```
181             pool.accMeowPerShare = pool.accMeowPerShare.add(meowReward
    .mul(ACC_MEOW_PRECISION).div(stakeTokenSupply));
182         }
183         pool.lastRewardTime = block.timestamp;
184     }
185 }
```

The amount of reward to be minted is limited by the max supply of $MEOW token (`cap()`).

MeowToken.sol

```
29 function mint(address _to, uint256 _amount) public onlyOwner {
30     require(totalSupply().add(_amount) <= cap(), "MeowToken::mint:: cap
    exceeded.");
31     _mint(_to, _amount);
32     _moveDelegates(address(0), _delegates[_to], _amount);
33 }
```

However, when the sum of the reward to be minted and the minted amount is more than the max supply, the `mint()` function will be unusable, causing the transactions that call this function to be reverted, disrupting the availability of the platform.

5.7.2. Remediation

Inspex suggests modifying the `MeowMining` contract to handle the case when the cap is filled, for example:

MeowMining.sol

```
168 uint256 public MAX_MEOW_REWARD = 200000000e18;
169 uint256 public MAX_DEV_FUND = 17500000e18;
170 uint256 public mintedDevFund;
171 uint256 public mintedMeowReward;
172
173 function updatePool(uint256 _pid) public {
174     PoolInfo storage pool = poolInfo[_pid];
175     if (block.timestamp > pool.lastRewardTime) {
176         uint256 stakeTokenSupply = IERC20(pool.stakeToken)
    .balanceOf(address(this));
177         if (stakeTokenSupply > 0 && totalAllocPoint > 0) {
178             uint256 time = block.timestamp.sub(pool.lastRewardTime);
179             uint256 meowReward =
    time.mul(meowPerSecond).mul(pool.allocPoint).div(totalAllocPoint);
180             // Every 11.4286 Meow minted will mint 1 Meow for dev, come from
    80/7 = 11.4286 use 10,000 to avoid floating.
181             uint256 devfund = meowReward.mul(10000).div(114286);
182             if(mintedMeowReward.add(meowReward) >
    MAX_MEOW_REWARD.sub(meow.manualMinted())) {
183                 meowReward = MAX_MEOW_REWARD.sub(mintedMeowReward);
```

```
184             }
185             if(mintedDevFund.add(devfund) > MAX_DEV_FUND) {
186                 devfund = MAX_DEV_FUND.sub(mintedDevFund);
187             }
188             meow.mint(address(this), devfund);
189             meow.mint(address(this), meowReward);
190             mintedDevFund = mintedDevFund.add(devfund);
191             mintedMeowReward = mintedMeowReward.add(meowReward);
192             safeMeowTransfer(devaddr, devfund.mul(preShare).div(10000));
193             developmentFund.lock(devfund.mul(lockShare).div(10000));
194             pool.accMeowPerShare = pool.accMeowPerShare.add(meowReward
    .mul(ACC_MEOW_PRECISION).div(stakeTokenSupply));
195         }
196         pool.lastRewardTime = block.timestamp;
197     }
198 }
```

5.8. Improper Delegation Handling in Token Burning

ID	IDX-008
Target	MeowToken
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Low **Impact:** Medium The number of votes can be higher than the amount of tokens available, causing the result of the vote to be unfair and untrustworthy, resulting in loss of reputation for the platform. **Likelihood:** Low This issue occurs when the token is burned. There is no burning mechanism in the use case of $MEOW token, and there is no benefit for the token holder to burn their own tokens.
Status	Resolved Meow Finance team has resolved this issue by deducting the delegation amount in the `burn()` function as suggested in commit `15137b093aab2fa27cc00a459058a52108333a51`.

5.8.1. Description

In the `MeowToken` contract, there is a voting mechanism implemented, allowing the users (delegators) to delegate their votes to another address (delegatees) without transferring their tokens.

The users can delegate their votes to another address using the `delegate()` function, which calls the `_delegate()` function.

MeowToken.sol

```
128 function delegate(address delegatee) external {
129     return _delegate(msg.sender, delegatee);
130 }
```

The `_delegate()` function sets the delegatee of the address in line 218, and transfers the number of votes from the old delegatee to the new delegatee with the current token balance of the delegator by using the `_moveDelegates()` function as in line 222.

MeowToken.sol

```
215 function _delegate(address delegator, address delegatee) internal {
216     address currentDelegate = _delegates[delegator];
```

```
217     uint256 delegatorBalance = balanceOf(delegator); // balance of underlying
    Meows (not scaled);
218     _delegates[delegator] = delegatee;
219
220     emit DelegateChanged(delegator, currentDelegate, delegatee);
221
222     _moveDelegates(currentDelegate, delegatee, delegatorBalance);
223 }
```

The `_moveDelegates()` function calculates the new amount of voting for the delegatee.

MeowToken.sol

```
1  function _moveDelegates(
2      address srcRep,
3      address dstRep,
4      uint256 amount
5  ) internal {
6      if (srcRep != dstRep && amount > 0) {
7          if (srcRep != address(0)) {
8              // decrease old representative
9              uint32 srcRepNum = numCheckpoints[srcRep];
10             uint256 srcRepOld = srcRepNum > 0 ? checkpoints[srcRep][srcRepNum -
    1].votes : 0;
11             uint256 srcRepNew = srcRepOld.sub(amount);
12             _writeCheckpoint(srcRep, srcRepNum, srcRepOld, srcRepNew);
13         }
14
15         if (dstRep != address(0)) {
16             // increase new representative
17             uint32 dstRepNum = numCheckpoints[dstRep];
18             uint256 dstRepOld = dstRepNum > 0 ? checkpoints[dstRep][dstRepNum -
    1].votes : 0;
19             uint256 dstRepNew = dstRepOld.add(amount);
20             _writeCheckpoint(dstRep, dstRepNum, dstRepOld, dstRepNew);
21         }
22     }
23 }
```

When the token is minted, the delegate amount is added to the delegatee of the `_to` address which receives the minted token.

MeowToken.sol

```
29 function mint(address _to, uint256 _amount) public onlyOwner {
30     require(totalSupply().add(_amount) <= cap(), "MeowToken::mint:: cap
    exceeded.");
31     _mint(_to, _amount);
```

```
32          _moveDelegates(address(0), _delegates[_to], _amount);
33      }
```

However, when the token is burned, the delegate amount is not removed. Therefore, the votes can still be cast even when the address does own the token.

MeowToken.sol

```
50  function burnFrom(address _account, uint256 _amount) external onlyOwner {
51      _burn(_account, _amount);
52  }
53
54  function burn(uint256 _amount) external {
55      _burn(msg.sender, _amount);
56  }
```

5.8.2. Remediation

Inspex suggests deducting the delegation vote on the burning of token, for example:

MeowToken.sol

```
50  function burnFrom(address _account, uint256 _amount) external onlyOwner {
51      _burn(_account, _amount);
52      _moveDelegates(_delegates[_account], address(0), _amount);
53  }
54
55  function burn(uint256 _amount) external {
56      _burn(msg.sender, _amount);
57      _moveDelegates(_delegates[msg.sender], address(0), _amount);
58  }
```

5.9. Design Flaw in massUpdatePool() Function

ID	IDX-009
Target	MeowMining FeeDistribute
Category	General Smart Contract Vulnerability
CWE	CWE-400: Uncontrolled Resource Consumption
Risk	**Severity:** Low **Impact:** Medium The `massUpdatePools()` function will eventually be unusable due to excessive gas usage. **Likelihood:** Low It is very unlikely that the `poolInfo` size will be raised until the `massUpdatePools()` function is unusable.
Status	**Acknowledged** Meow Finance team has acknowledged this issue. The team explained that the risk of this issue is quite low since the number of pools that will be added by the team is not high enough to cause the unfunctional smart contract issue.

5.9.1. Description

The `massUpdatePools()` function executes the `updatePool()` function, which is a state modifying function for all added pools as shown below:

MeowMining.sol

```
160 function massUpdatePools() public {
161     uint256 length = poolInfo.length;
162     for (uint256 pid = 0; pid < length; ++pid) {
163         updatePool(pid);
164     }
165 }
```

With the current design, the added pools cannot be removed. They can only be disabled by setting the `pool.allocPoint` to 0. Even if a pool is disabled, the `updatePool()` function for this pool is still called. Therefore, if new pools continue to be added to this contract, the `poolInfo.length` will continue to grow and this function will eventually be unusable due to excessive gas usage.

5.9.2. Remediation

Inspex suggests making the contract capable of removing unnecessary or ended pools to reduce the loop rounds in the `massUpdatePools()` function.

5.10. Transaction Ordering Dependence

ID	IDX-010
Target	SpookyswapWorker
Category	General Smart Contract Vulnerability
CWE	CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')
Risk	**Severity:** Low **Impact:** Medium The front-running attack can be performed, resulting in a bad swapping rate for the reinvestment. This causes the reinvestment fund to be lower, which is a minor monetary loss for the platform users. **Likelihood:** Low It is easy to perform the attack. However, with a low profit, there is low motivation to attack with this vulnerability.
Status	Resolved Meow Finance team has resolved this issue by implementing price oracle and calculating expected amount out when using the `router.swapExactTokensForTokens()` inside the `reinvest()` function in commit `0912b0099114939c3452117c1a25de82cfb6cd75`

5.10.1. Description

In `SpookyswapWorker` contracts, the reward of the farming is compounded using the `reinvest()` function.

In the compounding process, there are many subprocesses, the token swapping process is one of them. The swapping can be performed by calling `router.swapExactTokensForTokens()` function to swap the reward token (`boo`) to `baseToken` in line 163.

SpookyswapWorker.sol

```
138 function reinvest() external override onlyEOA onlyReinvestor nonReentrant {
139     // 1. Approve tokens
140     boo.safeApprove(address(router), uint256(-1));
141     address(lpToken).safeApprove(address(masterChef), uint256(-1));
142     // 2. Withdraw all the rewards.
143     masterChef.withdraw(pid, 0);
144     uint256 reward = boo.balanceOf(address(this));
145     if (reward == 0) return;
146     // 3. Send the reward bounty to the caller.
147     uint256 bounty = reward.mul(reinvestBountyBps) / 10000;
148     if (bounty > 0) boo.safeTransfer(msg.sender, bounty);
149     // 4. Convert all the remaining rewards to BaseToken via Native for
```

```
liquidity.
150     address[] memory path;
151     if (baseToken != boo) {
152         if (baseToken == wNative) {
153             path = new address[](2);
154             path[0] = address(boo);
155             path[1] = address(wNative);
156         } else {
157             path = new address[](3);
158             path[0] = address(boo);
159             path[1] = address(wNative);
160             path[2] = address(baseToken);
161         }
162     }
163     router.swapExactTokensForTokens(reward.sub(bounty), 0, path, address(this),
    now);
164
165     // 5. Use add Token strategy to convert all BaseToken to LP tokens.
166     baseToken.safeTransfer(address(addStrat), baseToken.myBalance());
167     addStrat.execute(address(0), 0, abi.encode(0));
168     // 6. Mint more LP tokens and stake them for more rewards.
169     masterChef.deposit(pid, lpToken.balanceOf(address(this)));
170     // 7. Reset approve
171     boo.safeApprove(address(router), 0);
172     address(lpToken).safeApprove(address(masterChef), 0);
173     emit Reinvest(msg.sender, reward, bounty);
174 }
```

However, as seen in the source code above, the swapping tolerance (`amountOutMin`) of the swapping function is set to 0. This allows a front-running attack to be done, resulting in fewer tokens gained from the swap. This reduces the amount of token being reinvested and causes the users to gain less reward.

5.10.2. Remediation

The tolerance value (`amountOutMin`) should not be set to 0. Inspex suggests calculating the expected amount out with the token price fetched from the price oracles, and setting it to the `amountOutMin` parameter while calling the `router.swapExactTokensForTokens()` function in the `SpookyswapWorker` contract, for example:

SpookyswapWorker.sol

```
1 function reinvest() external override onlyEOA onlyReinvestor nonReentrant {
2     // 1. Approve tokens
3     boo.safeApprove(address(router), uint256(-1));
4     address(lpToken).safeApprove(address(masterChef), uint256(-1));
5     // 2. Withdraw all the rewards.
6     masterChef.withdraw(pid, 0);
```

```
7     uint256 reward = boo.balanceOf(address(this));
8     if (reward == 0) return;
9     // 3. Send the reward bounty to the caller.
10    uint256 bounty = reward.mul(reinvestBountyBps) / 10000;
11    if (bounty > 0) boo.safeTransfer(msg.sender, bounty);
12    // 4. Convert all the remaining rewards to BaseToken via Native for
      liquidity.
13    address[] memory path;
14    if (baseToken != boo) {
15        if (baseToken == wNative) {
16            path = new address[](2);
17            path[0] = address(boo);
18            path[1] = address(wNative);
19        } else {
20            path = new address[](3);
21            path[0] = address(boo);
22            path[1] = address(wNative);
23            path[2] = address(baseToken);
24        }
25    }
26    uint256 amountOutMin = calculateAmountOutMinFromOracle(reward.sub(bounty));
27    router.swapExactTokensForTokens(reward.sub(bounty), amountOutMin, path,
      address(this), now);
28
29    // 5. Use add Token strategy to convert all BaseToken to LP tokens.
30    baseToken.safeTransfer(address(addStrat), baseToken.myBalance());
31    addStrat.execute(address(0), 0, abi.encode(0));
32    // 6. Mint more LP tokens and stake them for more rewards.
33    masterChef.deposit(pid, lpToken.balanceOf(address(this)));
34    // 7. Reset approve
35    boo.safeApprove(address(router), 0);
36    address(lpToken).safeApprove(address(masterChef), 0);
37    emit Reinvest(msg.sender, reward, bounty);
38 }
```

5.11. Missing Input Validation (maxReinvestBountyBps)

ID	IDX-011
Target	SpookyswapWorker
Category	Advanced Smart Contract Vulnerability
CWE	CWE-20: Improper Input Validation
Risk	**Severity:** Low **Impact:** Medium By setting `reinvestBountyBps` to be greater than 10,000, the cause the transaction reverting for all `work()` function executions. **Likelihood:** Low It is very unlikely that the owner will set an improperly large `reinvestBountyBps` because there is no profit to perform this action.
Status	Resolved Meow Finance team has resolved this issue by setting the upper limit of the `maxReinvestBountyBps` as suggested in commit `15137b093aab2fa27cc00a459058a52108333a51`.

5.11.1. Description

The `setReinvestBountyBps()` function can be used to set the `reinvestBountyBps` state.

SpookyswapWorker.sol

```
288 function setReinvestBountyBps(uint256 _reinvestBountyBps) external onlyOwner {
289     require(
290         _reinvestBountyBps <= maxReinvestBountyBps,
291         "SpookyswapWorker::setReinvestBountyBps:: _reinvestBountyBps exceeded
    maxReinvestBountyBps"
292     );
293     reinvestBountyBps = _reinvestBountyBps;
294 }
```

The `reinvestBountyBps` is limited by `maxReinvestBountyBps` state. However, the `maxReinvestBountyBps` can be set without any limitation as shown below:

SpookyswapWorker.sol

```
298 function setMaxReinvestBountyBps(uint256 _maxReinvestBountyBps) external
    onlyOwner {
299     require(
300         _maxReinvestBountyBps >= reinvestBountyBps,
```

```
301         "SpookyswapWorker::setMaxReinvestBountyBps:: _maxReinvestBountyBps
    lower than reinvestBountyBps"
302     );
303     maxReinvestBountyBps = _maxReinvestBountyBps;
304 }
```

The `reinvestBountyBps` state is used in the `reinvest()` function to determine the bounty rate of reinvesting as follow:

SpookyswapWorker.sol

```
138 function reinvest() external override onlyEOA onlyReinvestor nonReentrant {
139     // 1. Approve tokens
140     boo.safeApprove(address(router), uint256(-1));
141     address(lpToken).safeApprove(address(masterChef), uint256(-1));
142     // 2. Withdraw all the rewards.
143     masterChef.withdraw(pid, 0);
144     uint256 reward = boo.balanceOf(address(this));
145     if (reward == 0) return;
146     // 3. Send the reward bounty to the caller.
147     uint256 bounty = reward.mul(reinvestBountyBps) / 10000;
148     if (bounty > 0) boo.safeTransfer(msg.sender, bounty);
149     // 4. Convert all the remaining rewards to BaseToken via Native for
    liquidity.
150     address[] memory path;
151     if (baseToken != boo) {
152         if (baseToken == wNative) {
153             path = new address[](2);
154             path[0] = address(boo);
155             path[1] = address(wNative);
156         } else {
157             path = new address[](3);
158             path[0] = address(boo);
159             path[1] = address(wNative);
160             path[2] = address(baseToken);
161         }
162     }
163     router.swapExactTokensForTokens(reward.sub(bounty), 0, path, address(this),
    now);
164
165     // 5. Use add Token strategy to convert all BaseToken to LP tokens.
166     baseToken.safeTransfer(address(addStrat), baseToken.myBalance());
167     addStrat.execute(address(0), 0, abi.encode(0));
168     // 6. Mint more LP tokens and stake them for more rewards.
169     masterChef.deposit(pid, lpToken.balanceOf(address(this)));
170     // 7. Reset approve
171     boo.safeApprove(address(router), 0);
172     address(lpToken).safeApprove(address(masterChef), 0);
```

```
173     emit Reinvest(msg.sender, reward, bounty);
174 }
```

By setting `reinvestBountyBps` to be greater than 10,000, the bounty will be greater than the harvested reward and cause the transaction to be reverted for all `reinvest()` function executions.

5.11.2. Remediation

Inspex suggests setting the upper limit of the `maxReinvestBountyBps` for example:

SpookyswapWorker.sol

```
298 function setMaxReinvestBountyBps(uint256 _maxReinvestBountyBps) external
    onlyOwner {
299     require(
300         _maxReinvestBountyBps >= reinvestBountyBps,
301         "SpookyswapWorker::setMaxReinvestBountyBps:: _maxReinvestBountyBps
    lower than reinvestBountyBps"
302     );
303     require(_maxReinvestBountyBps <= 10000,
    "SpookyswapWorker::setMaxReinvestBountyBps:: _maxReinvestBountyBps higher than
    harvested reward");
304     maxReinvestBountyBps = _maxReinvestBountyBps;
305 }
```

5.12. Denial of Service in reinvest() Function

ID	IDX-012
Target	SpookyswapWorker
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Low **Impact:** Medium The `reinvest()` function will be unusable, disrupting the availability of the service. The users will not receive additional profit from the compounding mechanism. **Likelihood:** Low The `baseToken` can be set by only the initializer of the `SpookyswapWorker` contract. It is very unlikely that the baseToken will be the same as the reward token.
Status	Resolved Meow Finance team has resolved this issue by moving the swapping statement to the inside of the condition which checks if reward token is the same as the `baseToken` as suggested in commit `15137b093aab2fa27cc00a459058a52108333a51`.

5.12.1. Description

In `SpookyswapWorker` contracts, the reward of the farming is compounded using the `reinvest()` function.

In the compounding process, there are many subprocesses, the token swapping process is one of them.

The condition `baseToken != boo` in line 151 is used to check if the `baseToken` is not a reward token then set the path to swap the reward for the `baseToken`.

SpookyswapWorker.sol

```
138 function reinvest() external override onlyEOA onlyReinvestor nonReentrant {
139     // 1. Approve tokens
140     boo.safeApprove(address(router), uint256(-1));
141     address(lpToken).safeApprove(address(masterChef), uint256(-1));
142     // 2. Withdraw all the rewards.
143     masterChef.withdraw(pid, 0);
144     uint256 reward = boo.balanceOf(address(this));
145     if (reward == 0) return;
146     // 3. Send the reward bounty to the caller.
147     uint256 bounty = reward.mul(reinvestBountyBps) / 10000;
148     if (bounty > 0) boo.safeTransfer(msg.sender, bounty);
149     // 4. Convert all the remaining rewards to BaseToken via Native for
    liquidity.
```

```
150     address[] memory path;
151     if (baseToken != boo) {
152         if (baseToken == wNative) {
153             path = new address[](2);
154             path[0] = address(boo);
155             path[1] = address(wNative);
156         } else {
157             path = new address[](3);
158             path[0] = address(boo);
159             path[1] = address(wNative);
160             path[2] = address(baseToken);
161         }
162     }
163     router.swapExactTokensForTokens(reward.sub(bounty), 0, path, address(this),
    now);
164
165     // 5. Use add Token strategy to convert all BaseToken to LP tokens.
166     baseToken.safeTransfer(address(addStrat), baseToken.myBalance());
167     addStrat.execute(address(0), 0, abi.encode(0));
168     // 6. Mint more LP tokens and stake them for more rewards.
169     masterChef.deposit(pid, lpToken.balanceOf(address(this)));
170     // 7. Reset approve
171     boo.safeApprove(address(router), 0);
172     address(lpToken).safeApprove(address(masterChef), 0);
173     emit Reinvest(msg.sender, reward, bounty);
174 }
```

When the `baseToken` is a reward token, the path will be empty, causing the reinvest transaction to be reverted, because the `getAmountsOut()` function called by the `swapExactTokensForTokens()` function has a `require` statement to check that the path length is equal to or more than 2. This can be seen in line 266 in the example code from `UniswapV2Router02` contract of SpookySwap platform.

UniswapV2Router02.sol at https://ftmscan.com/address/0xf491e7b69e4244ad4002bc14e878a34207e38c29

```
561 function swapExactTokensForTokens(
562     uint amountIn,
563     uint amountOutMin,
564     address[] calldata path,
565     address to,
566     uint deadline
567 ) external virtual override ensure(deadline) returns (uint[] memory amounts) {
568     amounts = UniswapV2Library.getAmountsOut(factory, amountIn, path);
569     require(amounts[amounts.length - 1] >= amountOutMin, 'UniswapV2Router:
    INSUFFICIENT_OUTPUT_AMOUNT');
570     TransferHelper.safeTransferFrom(
571         path[0], msg.sender, UniswapV2Library.pairFor(factory, path[0],
    path[1]), amounts[0]
```

```
572     );
573     _swap(amounts, path, to);
574 }
```

UniswapV2Router02.sol at https://ftmscan.com/address/0xf491e7b69e4244ad4002bc14e878a34207e38c29

```
265 function getAmountsOut(address factory, uint amountIn, address[] memory path)
    internal view returns (uint[] memory amounts) {
266     require(path.length >= 2, 'UniswapV2Library: INVALID_PATH');
267     amounts = new uint[](path.length);
268     amounts[0] = amountIn;
269     for (uint i; i < path.length - 1; i++) {
270         (uint reserveIn, uint reserveOut) = getReserves(factory, path[i],
    path[i + 1]);
271         amounts[i + 1] = getAmountOut(amounts[i], reserveIn, reserveOut);
272     }
273 }
```

5.12.2. Remediation

When the reward token is the same as the **baseToken**, there is no need to swap the token before sending it to the **addStrat** contract. Inspex suggests moving swapping statement to the inside of the condition in line 151 in the **reinvest()** function which checks if reward token is the same as the **baseToken** to ignore the swapping, for example:

SpookyswapWorker.sol

```
138 function reinvest() external override onlyEOA onlyReinvestor nonReentrant {
139     // 1. Approve tokens
140     boo.safeApprove(address(router), uint256(-1));
141     address(lpToken).safeApprove(address(masterChef), uint256(-1));
142     // 2. Withdraw all the rewards.
143     masterChef.withdraw(pid, 0);
144     uint256 reward = boo.balanceOf(address(this));
145     if (reward == 0) return;
146     // 3. Send the reward bounty to the caller.
147     uint256 bounty = reward.mul(reinvestBountyBps) / 10000;
148     if (bounty > 0) boo.safeTransfer(msg.sender, bounty);
149     // 4. Convert all the remaining rewards to BaseToken via Native for
    liquidity.
150     address[] memory path;
151     if (baseToken != boo) {
152         if (baseToken == wNative) {
153             path = new address[](2);
154             path[0] = address(boo);
155             path[1] = address(wNative);
156         } else {
157             path = new address[](3);
```

```
158             path[0] = address(boo);
159             path[1] = address(wNative);
160             path[2] = address(baseToken);
161         }
162     router.swapExactTokensForTokens(reward.sub(bounty), 0, path, address(this),
    now);
163     }
164
165     // 5. Use add Token strategy to convert all BaseToken to LP tokens.
166     baseToken.safeTransfer(address(addStrat), baseToken.myBalance());
167     addStrat.execute(address(0), 0, abi.encode(0));
168     // 6. Mint more LP tokens and stake them for more rewards.
169     masterChef.deposit(pid, lpToken.balanceOf(address(this)));
170     // 7. Reset approve
171     boo.safeApprove(address(router), 0);
172     address(lpToken).safeApprove(address(masterChef), 0);
173     emit Reinvest(msg.sender, reward, bounty);
174 }
```

5.13. Missing Input Validation of preShare and lockShare Values

ID	IDX-013
Target	MeowMining
Category	Advanced Smart Contract Vulnerability
CWE	CWE-20: Improper Input Validation
Risk	**Severity:** Low **Impact:** Medium When the sum of `preShare` and `lockShare` is lower than 10,000, the reward minted for the allocation of `MeowMining` portion will be left in the contract, causing the total reward distributed to the users to be lower. When the sum is greater than 10,000, some functions in the `MeowMining` contract will be unusable, resulting in monetary loss for the users and loss of reputation for the platform. **Likelihood:** Low It is unlikely that the contract owner will set improper values of `preShare` and `lockShare`; however, it is possible since there is no validation.
Status	Resolved Meow Finance team has resolved this issue in commit `15137b093aab2fa27cc00a459058a52108333a51` as follows: - limiting the `preShare` value to 10,000 or lower - removing the `lockShare` variable - calculating the lock amount from the value of `preShare` variable

5.13.1. Description

The amount of harvested reward in the `MeowMining` contract is split from the pending reward amount into two parts: `preShare` and `lockShare`.

MeowMining.sol

```
248 function _harvest(address _to, uint256 _pid) internal {
249     PoolInfo storage pool = poolInfo[_pid];
250     UserInfo storage user = userInfo[_pid][_to];
251     require(user.amount > 0, "MeowMining::harvest:: nothing to harvest.");
    uint256 pending =
252 user.amount.mul(pool.accMeowPerShare).div(ACC_MEOW_PRECISION).sub(user.rewardDe
    bt);
253     uint256 preAmount = pending.mul(preShare).div(10000);
254     uint256 lockAmount = pending.mul(lockShare).div(10000);
255     lock(_pid, _to, lockAmount);
256     require(preAmount <= meow.balanceOf(address(this)), "MeowMining::harvest::
```

```
    not enough Meow.");
257     safeMeowTransfer(_to, preAmount);
258     emit Harvest(msg.sender, _pid, preAmount);
259 }
```

The `preShare` is a part of the pending amount that can be immediately harvested.

The `lockShare` is another part of the pending amount that is locked for 90 days.

The `preShare` and `lockShare` are used to calculate `preAmount` and `lockAmount` in the `_harvest()` function.

MeowMining.sol

```
248 function _harvest(address _to, uint256 _pid) internal {
249     PoolInfo storage pool = poolInfo[_pid];
250     UserInfo storage user = userInfo[_pid][_to];
251     require(user.amount > 0, "MeowMining::harvest:: nothing to harvest.");
252     uint256 pending = user.amount.mul(pool.accMeowPerShare)
    .div(ACC_MEOW_PRECISION).sub(user.rewardDebt);
253     uint256 preAmount = pending.mul(preShare).div(10000);
254     uint256 lockAmount = pending.mul(lockShare).div(10000);
255     lock(_pid, _to, lockAmount);
256     require(preAmount <= meow.balanceOf(address(this)), "MeowMining::harvest::
    not enough Meow.");
257     safeMeowTransfer(_to, preAmount);
258     emit Harvest(msg.sender, _pid, preAmount);
259 }
```

From the source code above, the `preAmount` and `lockAmount` are calculated by a denominator (10,000).

The sum of `preShare` and `lockShare` must be equal to 10,000 for the calculation to be correct.

If the sum is greater than 10,000, the `_harvest()` function and other functions that execute `_harvest()` will be unusable because the pending amount is insufficient to be split for over 100%.

And if the sum is lower than 10,000, the $MEOW will be left in the `MeowMining` contract because a part of the pending amount other than the amount harvested and locked is not used.

5.13.2. Remediation

Inspex suggests adding a condition to make sure that the sum of `preShare` and `lockShare` is equal to 10,000, for example:

MeowMining.sol

```
81 constructor(
82     MeowToken _meow,
```

```
83      uint256 _meowPerSecond,
84      uint256 _startTime,
85      uint256 _preShare,
86      uint256 _lockShare,
87      address _devaddr,
88      DevelopmentFund _developmentFund
89  ) public {
90      require(_preShare.add(_lockShare) == 10000, "MeowMining:: sum of _preShare
        and _lockShare should be 10000.");
91      totalAllocPoint = 0;
92      meow = _meow;
93      meowPerSecond = _meowPerSecond;
94      startTime = block.timestamp > _startTime ? block.timestamp : _startTime;
95      preShare = _preShare;
96      lockShare = _lockShare;
97      devaddr = _devaddr;
98      developmentFund = _developmentFund;
99      meow.approve(address(_developmentFund), uint256(-1));
100 }
```

5.14. Outdated Compiler Version

ID	IDX-014
Target	Vault TripleSlopeModel MeowMining SpookyswapWorker MeowToken FeeDistribute DevelopmentFund
Category	General Smart Contract Vulnerability
CWE	CWE-1104: Use of Unmaintained Third Party Components
Risk	**Severity:** Very Low **Impact:** Low From the list of known Solidity bugs, direct impact cannot be caused by those bugs themselves. **Likelihood:** Low From the list of known Solidity bugs, it is very unlikely that those bugs would affect these smart contracts.
Status	Resolved Meow Finance team has resolved this issue by upgrading the Solidity compiler version as suggested in commit `15137b093aab2fa27cc00a459058a52108333a51`.

5.14.1. Description

The Solidity compiler version specified in the smart contracts was outdated. This version has publicly known inherent bugs[2] that may potentially be used to cause damage to the smart contracts or the users of the smart contracts.

MeowMining.sol

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity 0.6.6;
```

5.14.2. Remediation

Inspex suggests upgrading the Solidity compiler to the latest stable version[3]. During the audit activity, the latest stable version of Solidity compiler in major 0.6 is v0.6.12.

5.15. Insufficient Logging for Privileged Functions

ID	IDX-015
Target	Vault TripleSlopeModel MeowMining SpookyswapWorker MeowToken FeeDistribute DevelopmentFund
Category	General Smart Contract Vulnerability
CWE	CWE-778: Insufficient Logging
Risk	**Severity:** Very Low **Impact:** Low Privileged functions' executions cannot be monitored easily by the users. **Likelihood:** Low It is not likely that the execution of the privileged functions will be a malicious action.
Status	Resolved Meow Finance team has resolved this issue by emitting events as suggested in commit `15137b093aab2fa27cc00a459058a52108333a51`.

5.15.1. Description

Privileged functions that are executable by the controlling parties are not logged properly by emitting events. Without events, it is not easy for the public to monitor the execution of those privileged functions, allowing the controlling parties to perform actions that cause big impacts on the platform.

For example, the owner can modify the `meowPerSecond` by executing `setMeowPerSecond()` function in the `MeowMining` contract, and no event is emitted.

The privileged functions without sufficient logging are as follows:

File	Contract	Function	Modifier
Vault.sol (L:454)	Vault	updateConfig()	onlyOwner
Vault.sol (L:460)	Vault	updateDebtToken()	onlyOwner
Vault.sol (L:471)	Vault	setMeowMiningPoolId()	onlyOwner
TripleSlopeModel.sol (L:29)	TripleSlopeModel	setParams()	onlyOwner

MeowMining.sol (L:107)	MeowMining	setMeowPerSecond()	onlyOwner
MeowMining.sol (L:113)	MeowMining	addPool()	onlyOwner
MeowMining.sol (L:126)	MeowMining	setPool()	onlyOwner
MeowMining.sol (L:140)	MeowMining	manualMint()	onlyOwner
SpookyswapWorker.sol (L:138)	SpookyswapWorker	reinvest()	onlyReinvestor
SpookyswapWorker.sol (L:181)	SpookyswapWorker	work()	onlyOperator
SpookyswapWorker.sol (L:244)	SpookyswapWorker	liquidate()	onlyOwner
SpookyswapWorker.sol (L:288)	SpookyswapWorker	setReinvestBountyBps()	onlyOwner
SpookyswapWorker.sol (L:298)	SpookyswapWorker	setMaxReinvestBountyBps()	onlyOwner
SpookyswapWorker.sol (L:309)	SpookyswapWorker	setStrategyOk()	onlyOwner
SpookyswapWorker.sol (L:319)	SpookyswapWorker	setReinvestorOk()	onlyOwner
SpookyswapWorker.sol (L:329)	SpookyswapWorker	setCriticalStrategies()	onlyOwner
MeowToken.sol (L:29)	MeowToken	mint()	onlyOwner
MeowToken.sol (L:42)	MeowToken	manualMint()	onlyOwner
MeowToken.sol (L:50)	MeowToken	burnFrom()	onlyOwner
FeeDistribute.sol (L:52)	FeeDistribute	setParams()	onlyOwner
FeeDistribute.sol (L:62)	FeeDistribute	addPool()	onlyOwner

5.15.2. Remediation

Inspex suggests emitting events for the execution of privileged functions, for example:

MeowMining.sol

```
107 event SetMeowPerSecond(uint256 _meowPerSecond);
108 function setMeowPerSecond(uint256 _meowPerSecond) external onlyOwner {
109     massUpdatePools();
110     meowPerSecond = _meowPerSecond;
111     emit SetMeowPerSecond(_meowPerSecond);
112 }
```

5.16. Unavailability of manualMint() Function

ID	IDX-016
Target	MeowToken
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved Meow Finance team has resolved this issue by removing the `manualMint()` function in commit `15137b093aab2fa27cc00a459058a52108333a51`.

5.16.1. Description

The `manualMint()` function is used for minting $MEOW manually by the contract owner. The available amount of minting is determined by the `canManualMint()` function.

MeowToken.sol

```
42 function manualMint(address _to, uint256 _amount) public onlyOwner {
43     require(block.timestamp >= manualMintAllowedAfter, "MeowToken::manualMint::
   manualMint not allowed yet.");
44     require(_amount <= (canManualMint()), "MeowToken::manualMint:: manual mint
   limit exceeded.");
45     manualMintAllowedAfter = block.timestamp.add(minimumTimeBetweenManualMint);
46     manualMinted = manualMinted.add(_amount);
47     mint(_to, _amount);
48 }
```

In the `canManualMint()` function, the `miningMinted` amount is calculated by the `totalSupply()` minus the `reserve` amount by using the `sub()` function from `SafeMath` library.

MeowToken.sol

```
35 function canManualMint() public view returns (uint256) {
36     uint256 miningMinted = totalSupply().sub(reserve); // Totalsupply =
   MeowMining + DevFund + reserve.
37     // Every 11.4286 Meow minted will mint 1 Meow for dev, come from 80/7 =
   11.4286 use 10,000 to avoid floating.
38     uint256 devFund = miningMinted.mul(10000).div(114286);
39     return (uint256(200000000e18).sub((miningMinted).sub(devFund))).div(5); //
```

```
20% of (MeowMining - DevFund)
}
```

The `reserve` amount is defined as a constant value in line 15.

MeowToken.sol

```
15 uint256 public reserve = 32500000e18;
```

The `totalSupply` state is declared but the value of it is not defined, so the value of the `totalSupply()` will be 0 by default.

When the `totalSupply()` is less than the reserve, the calculation of `miningMinted` value will be reverted by subtraction overflow protection mechanism from `SafeMath` library. This can cause the `manualMint()` function to be unusable until the minted $MEOW amount reaches the `reserve` amount.

5.16.2. Remediation

Inspex suggests minting the reserve token amount to the reserve wallet in the constructor, for example:

MeowToken.sol

```
constructor(address _reserveWallet) public {
    mint(_reserveWallet, 32500000e18);
}
```

5.17. Improper Access Control for Development Fund Locking

ID	IDX-017
Target	DevelopmentFund
Category	Advanced Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved Meow Finance team has resolved this issue by limiting the addresses that can call the lock function as suggested in commit `15137b093aab2fa27cc00a459058a52108333a51`.

5.17.1. Description

After the token distribution by the `MeowMining` contract has finished, there is a lock period of 2 years to gradually release the development fund token.

The `lock()` function is declared with `public` visibility, so anyone can call this function.

DevelopmentFund.sol

```
37 function lock(uint256 _amount) public {
38     Meow.safeTransferFrom(msg.sender, address(this), _amount);
39     unlock();
40     if (_amount > 0) {
41         lockedAmount = lockedAmount.add(_amount);
42         lockTo = block.timestamp.add(lockPeriod);
43     }
44 }
```

The lock duration is set to 2 years.

DevelopmentFund.sol

```
17 uint256 public lockPeriod = 365 days * 2;
```

Every time that anyone calls the `lock()` function, the lock period will be extended by the `lockPeriod` value, causing the developer to gain less token from the locked amount.

5.17.2. Remediation

Inspex suggests limiting the addresses that can call the `lock()` function to allow only trusted addresses to call the `lock()` function.

5.18. Improper Access Control for burnFrom() Function

ID	IDX-018
Target	MeowToken
Category	Advanced Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved Meow Finance team has resolved this issue by removing the `burnFrom()` function in commit `15137b093aab2fa27cc00a459058a52108333a51`.

5.18.1. Description

The `burnFrom()` function can be used to burn $MEOW from any wallet address by the contract owner of the `MeowToken` contract.

MeowToken.sol

```
50 function burnFrom(address _account, uint256 _amount) external onlyOwner {
51     _burn(_account, _amount);
52 }
```

Tokens in user wallets can be burned at any time without their consent, which poses a risk to the token holder.

However, the `MeowToken` contract is designed to be owned by the `MeowMining` contract and can not execute the `burnFrom()` function, so this function will not be executed once the ownership has been transferred to the `MeowMining` contract.

5.18.2. Remediation

The tokens should be burned only with the consent of the holder. Inspex suggests checking allowance amount before burning, for example:

MeowToken.sol

```
50 function burnFrom(address _account, uint256 _amount) external onlyOwner {
51     _approve(
52         _account,
53         _msgSender(),
54         allowance(_account, _msgSender()).sub(_amount, "MeowToken::burnFrom::
   burn amount exceeds allowance")
55     );
56     _burn(_account, _amount);
57 }
```

5.19. Unsupported Design for Deflationary Token

ID	IDX-019
Target	MeowMining
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved Meow Finance team has resolved this issue by modifying the `deposit()` function as suggested in commit `15137b093aab2fa27cc00a459058a52108333a51`.

5.19.1. Description

In the `MeowMining` contract, the users can deposit their tokens to acquire a reward ($MEOW). The deposited tokens can be a normal token or LP token depending on the pools added by the contract owner.

However, in the `deposit()` function, an issue could arise when the pool uses a deflationary token (the token that reduces the circulating supply itself when it is transferred).

This means the `_amount` that user deposits will be reduced due to the deflationary mechanism, but the contract recognizes it as the full amount as in line 201.

MeowMining.sol

```
188 function deposit(
189     address _for,
190     uint256 _pid,
191     uint256 _amount
192 ) external nonReentrant {
193     PoolInfo storage pool = poolInfo[_pid];
194     UserInfo storage user = userInfo[_pid][_for];
195     if (user.fundedBy != address(0)) require(user.fundedBy == msg.sender,
    "MeowMining::deposit:: bad sof.");
196     require(pool.stakeToken != address(0), "MeowMining::deposit:: not accept
    deposit.");
197     updatePool(_pid);
198     if (user.amount > 0) _harvest(_for, _pid);
199     if (user.fundedBy == address(0)) user.fundedBy = msg.sender;
200     IERC20(pool.stakeToken).safeTransferFrom(address(msg.sender),
    address(this), _amount);
```

```
201         user.amount = user.amount.add(_amount);
202         user.rewardDebt =
    user.amount.mul(pool.accMeowPerShare).div(ACC_MEOW_PRECISION);
203         emit Deposit(msg.sender, _pid, _amount);
204     }
```

The failure of recognizing the token amount could lead to the following scenarios:

Scenario 1: Unable to withdraw staking tokens

Assuming that there is a pool in the `MeowMining` contract which receives a deflationary token ($TOKEN) with a 10% burn rate when the token is transferred.

Currently, there is only User A who stakes $TOKEN to the $TOKEN pool in the `MeowMining` contract.

Holder	Balance
User A	100

Total $TOKEN in the `MeowMining` contract: 90

User B deposits 100 $TOKEN to the $TOKEN pool in the `MeowMining` contract. The `MeowMining` contract will receive 90 $TOKEN since $TOKEN has 10% deduction from the deflationary mechanism, in this case 10 $TOKEN.

Holder	Balance
User A	100
User B	100

Total $TOKEN in the `MeowMining` contract: 180

User B then withdraws 100 $TOKEN from the `MeowMining` contract. The `MeowMining` contract will validate whether the withdrawn `_amount` exceeds the `user.amount`.

MeowMining.sol

```
207 function withdraw(
208     address _for,
209     uint256 _pid,
210     uint256 _amount
211 ) external nonReentrant {
212     _withdraw(_for, _pid, _amount);
213 }
214
215 function withdrawAll(address _for, uint256 _pid) external nonReentrant {
```

```
216     _withdraw(_for, _pid, userInfo[_pid][_for].amount);
217 }
218
219 function _withdraw(
220     address _for,
221     uint256 _pid,
222     uint256 _amount
223 ) internal {
224     PoolInfo storage pool = poolInfo[_pid];
225     UserInfo storage user = userInfo[_pid][_for];
226     require(user.fundedBy == msg.sender, "MeowMining::withdraw:: only
    funder.");
227     require(user.amount >= _amount, "MeowMining::withdraw:: not good.");
228     updatePool(_pid);
229     _harvest(_for, _pid);
230     user.amount = user.amount.sub(_amount);
231     user.rewardDebt = user.amount.mul(pool.accMeowPerShare)
    .div(ACC_MEOW_PRECISION);
232     if (user.amount == 0) user.fundedBy = address(0);
233     if (pool.stakeToken != address(0)) {
234         IERC20(pool.stakeToken).safeTransfer(address(msg.sender), _amount);
235     }
236     emit Withdraw(msg.sender, _pid, user.amount);
237 }
```

Since User B deposited 100 $TOKEN and the balance of $TOKEN in the contract is greater than 100, User B is allowed to withdraw 100 $TOKEN.

Holder	Balance
User A	100
User B	0

Total $TOKEN in the `MeowMining` contract: 80

As a result, if User A decides to withdraw 100 $TOKEN, this transaction will be reverted since the balance in the contract is insufficient.

Scenario 2: Reward Calculation Exploit

Assuming that there is a pool in the `MeowMining` contract which receives a deflationary token ($TOKEN) with 10% burn rate when the token is transferred.

Currently, several users stake $TOKEN to the $TOKEN pool in the `MeowMining` contract with a total supply of 100 $TOKEN.

User A deposits 100 $TOKEN to the contract, and the contract receives 90 $TOKEN due to the deflationary mechanism, resulting in a total supply of 190 $TOKEN.

After that, User A withdraws 100 $TOKEN from staking, the `MeowMining` contract will then calculate the rewards as in line 252.

MeowMining.sol

```
248 function _harvest(address _to, uint256 _pid) internal {
249     PoolInfo storage pool = poolInfo[_pid];
250     UserInfo storage user = userInfo[_pid][_to];
251     require(user.amount > 0, "MeowMining::harvest:: nothing to harvest.");
252     uint256 pending = user.amount.mul(pool.accMeowPerShare)
    .div(ACC_MEOW_PRECISION).sub(user.rewardDebt);
253     uint256 preAmount = pending.mul(preShare).div(10000);
254     uint256 lockAmount = pending.mul(lockShare).div(10000);
255     lock(_pid, _to, lockAmount);
256     require(preAmount <= meow.balanceOf(address(this)), "MeowMining::harvest::
    not enough Meow.");
257     safeMeowTransfer(_to, preAmount);
258     emit Harvest(msg.sender, _pid, preAmount);
259 }
```

During the calculation, the reward is affected by the total amount of $TOKEN (`stakeTokenSupply`) as in line 171.

MeowMining.sol

```
168 function updatePool(uint256 _pid) public {
169     PoolInfo storage pool = poolInfo[_pid];
170     if (block.timestamp > pool.lastRewardTime) {
171         uint256 stakeTokenSupply = IERC20(pool.stakeToken)
    .balanceOf(address(this));
172         if (stakeTokenSupply > 0 && totalAllocPoint > 0) {
173             uint256 time = block.timestamp.sub(pool.lastRewardTime);
174             uint256 meowReward = time.mul(meowPerSecond)
    .mul(pool.allocPoint).div(totalAllocPoint);
175             // Every 11.4286 Meow minted will mint 1 Meow for dev, come from
    80/7 = 11.4286 use 10,000 to avoid floating.
176             uint256 devfund = meowReward.mul(10000).div(114286);
177             meow.mint(address(this), devfund);
178             meow.mint(address(this), meowReward);
179             safeMeowTransfer(devaddr, devfund.mul(preShare).div(10000));
180             developmentFund.lock(devfund.mul(lockShare).div(10000));
181             pool.accMeowPerShare = pool.accMeowPerShare.add(meowReward
    .mul(ACC_MEOW_PRECISION).div(stakeTokenSupply));
182         }
183         pool.lastRewardTime = block.timestamp;
```

```
    }
}
```

Since the `MeowMining` contract registers the `user.amount` of User A as 100 $TOKEN, the withdrawn $TOKEN amount will be 100, resulting in reducing the total amount of $TOKEN in the contract to 90 $TOKEN.

Hence, the value of `pool.accMeowPerShare` can be increased dramatically by manipulating the total amount of $TOKEN (`stakeTokenSupply`) to be as low as possible.

User A can repeatedly execute `withdraw()` and `deposit()` functions to drain the $TOKEN in the contract until it is as low as possible, for example, 1 `wei`, causing the `accMeowPerShare` state to be overly inflated, so the users can claim an exceedingly large amount of reward ($MEOW) from the contract.

However, since only LP tokens are planned to be used in `MeowMining` pools, there is no direct impact for this issue.

5.19.2. Remediation

Inspex suggests modifying the logic of the `deposit()` function to validate the amount of the received token from the user instead of using the value of the `_amount` parameter directly.

MeowMining.sol

```
function deposit(
    address _for,
    uint256 _pid,
    uint256 _amount
) external nonReentrant {
    PoolInfo storage pool = poolInfo[_pid];
    UserInfo storage user = userInfo[_pid][_for];
    if (user.fundedBy != address(0)) require(user.fundedBy == msg.sender,
"MeowMining::deposit:: bad sof.");
    require(pool.stakeToken != address(0), "MeowMining::deposit:: not accept
deposit.");
    updatePool(_pid);
    if (user.amount > 0) _harvest(_for, _pid);
    if (user.fundedBy == address(0)) user.fundedBy = msg.sender;
    uint256 currentBal = pool.lpToken.balanceOf(address(this));
    IERC20(pool.stakeToken).safeTransferFrom(address(msg.sender),
address(this), _amount);
    uint256 receivedAmount = pool.lpToken.balanceOf(address(this)) -
currentBal;
    user.amount = user.amount.add(receivedAmount);
    user.rewardDebt = user.amount.mul(pool.accMeowPerShare)
.div(ACC_MEOW_PRECISION);
    emit Deposit(msg.sender, _pid, receivedAmount);
}
```

5.20. Improper Function Visibility

ID	IDX-020
Target	DevelopmentFund FeeDistribute MeowMining
Category	Smart Contract Best Practice
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved Meow Finance team has resolved this issue by changing functions' visibility to external as suggested in commit `15137b093aab2fa27cc00a459058a52108333a51`.

5.20.1. Description

Functions with public visibility copy calldata to memory when being executed, while external functions can read directly from calldata. Memory allocation uses more resources (gas) than reading directly from calldata.

For example, the following source code shows that the `setDev()` function of the `DevelopmentFund` contract is set to public and it is never called from any internal function.

DevelopmentFund.sol

```
31 function setDev(address _devaddr) public {
32     require(msg.sender == devaddr, "DevelopmentFund::setDev:: Forbidden.");
33     devaddr = _devaddr;
34 }
```

The following table contains all functions that have `public` visibility and are never called from any internal function.

File	Contract	Function
DevelopmentFund.sol (L:31)	DevelopmentFund	setDev()
DevelopmentFund.sol (L:37)	DevelopmentFund	lock()
FeeDistribute.sol (L:31)	FeeDistribute	setParams()
MeowMining.sol (L:102)	MeowMining	setDev()

MeowMining.sol (L:290)	MeowMining	unlock()

5.20.2. Remediation

Inspex suggests changing all functions' visibility to `external` if they are not called from any `internal` function as shown in the following example:

DevelopmentFund.sol

```
31 function setDev(address _devaddr) external {
32     require(msg.sender == devaddr, "DevelopmentFund::setDev:: Forbidden.");
33     devaddr = _devaddr;
34 }
```

6. Appendix

6.1. About Inspex

Inspex is formed by a team of cybersecurity experts highly experienced in various fields of cybersecurity. We provide blockchain and smart contract professional services at the highest quality to enhance the security of our clients and the overall blockchain ecosystem.

Follow Us On:

Website	https://inspex.co
Twitter	@InspexCo
Facebook	https://www.facebook.com/InspexCo
Telegram	@inspex_announcement

6.2. References

[1] "OWASP Risk Rating Methodology." [Online]. Available: https://owasp.org/www-community/OWASP_Risk_Rating_Methodology. [Accessed: 08-May-2021]

[2] "List of Known Bugs — Solidity 0.6.6 documentation." [Online]. Available: https://docs.soliditylang.org/en/v0.6.6/bugs.html. [Accessed: 1-Oct-2021]

[3] ethereum, "Releases · ethereum/solidity." [Online]. Available: https://github.com/ethereum/solidity/releases. [Accessed: 1-Oct-2021]

inspeX

CYBERSECURITY PROFESSIONAL SERVICE