



PALADIN
BLOCKCHAIN SECURITY

Smart Contract Security Assessment

Final Report

For Merchant Moe

18 December 2023



paladinsec.co



info@paladinsec.co

Table of Contents

Table of Contents	2
Disclaimer	5
1 Overview	6
1.1 Summary	6
1.2 Contracts Assessed	7
1.3 Findings Summary	8
1.3.1 Global Issues	9
1.3.2 Moe	9
1.3.3 MasterChef	9
1.3.4 MoeStaking	9
1.3.5 StableMoe	10
1.3.6 VeMoe	10
1.3.7 Libraries/BaseRewarder	11
1.3.8 Libraries/MasterChefRewarder	11
1.3.9 Libraries/VeMoeRewarder	11
1.3.10 DEX/MoeFactory	11
1.3.11 DEX/MoeERC20	11
1.3.12 DEX/MoePair	12
1.3.13 DEX/MoeRouter	12
1.3.14 DEX/MoeLibrary	12
1.3.15 Libraries/Amounts	12
1.3.16 Libraries/Math	12
1.3.17 Libraries/Constants	13
1.3.18 Libraries/Rewarder	13
1.3.19 JoeStaking	13
1.3.20 JoeStakingRewarder	13
1.3.21 RewarderFactory	14

1.3.22 VestingContract	14
2 Findings	15
2.1 Global Issues	15
2.1.1 Issues & Recommendations	16
2.2 Moe	18
2.2.1 Privileged Functions	18
2.2.2 Issues & Recommendations	18
2.3 MasterChef	19
2.3.1 Privileged Functions	20
2.3.2 Issues & Recommendations	21
2.4 MoeStaking	26
2.4.1 Issues & Recommendations	27
2.5 StableMoe	28
2.5.1 Privileged Functions	28
2.5.2 Issues & Recommendations	29
2.6 VeMoe	34
2.6.1 Privileged Functions	35
2.6.2 Issues & Recommendations	36
2.7 Libraries/BaseRewarder	45
2.7.1 Privileged Functions	45
2.7.2 Issues & Recommendations	46
2.8 Libraries/MasterChefRewarder	48
2.8.1 Privileged Functions	48
2.8.2 Issues & Recommendations	48
2.9 Libraries/VeMoeRewarder	49
2.9.1 Privileged Functions	49
2.9.2 Issues & Recommendations	49
2.10 DEX/MoeFactory	50
2.10.1 Privileged Functions	50
2.10.2 Issues & Recommendations	50

2.11 DEX/MoeERC20	51
2.11.1 Issues & Recommendations	52
2.12 DEX/MoePair	54
2.12.1 Privileged Functions	54
2.12.2 Issues & Recommendations	55
2.13 DEX/MoeRouter	56
2.13.1 Issues & Recommendations	57
2.14 DEX/MoeLibrary	59
2.14.1 Issues & Recommendations	59
2.15 Libraries/Amounts	60
2.15.1 Issues & Recommendations	60
2.16 Libraries/Math	61
2.16.1 Issues & Recommendations	61
2.17 Libraries/Constants	62
2.17.1 Issues & Recommendations	62
2.18 Libraries/Rewarder	63
2.18.1 Issues & Recommendations	64
2.19 JoeStaking	65
2.19.1 Privileged Functions	65
2.19.2 Issues & Recommendations	66
2.20 JoeStakingRewarder	68
2.20.1 Privileged Functions	68
2.20.2 Issues & Recommendations	68
2.21 RewarderFactory	69
2.21.1 Privileged Functions	69
2.21.2 Issues & Recommendations	70
2.22 VestingContract	74
2.22.1 Privileged Functions	74
2.22.2 Issues & Recommendations	75

Disclaimer

Paladin Blockchain Security ("Paladin") has conducted an independent audit to verify the integrity of and highlight any vulnerabilities or errors, intentional or unintentional, that may be present in the codes that were provided for the scope of this audit. This audit report does not constitute agreement, acceptance or advocacy for the Project that was audited, and users relying on this audit report should not consider this as having any merit for financial advice in any shape, form or nature. The contracts audited do not account for any economic developments that may be pursued by the Project in question, and that the veracity of the findings thus presented in this report relate solely to the proficiency, competence, aptitude and discretion of our independent auditors, who make no guarantees nor assurance that the contracts are completely free of exploits, bugs, vulnerabilities or deprecation of technologies. Further, this audit report shall not be disclosed nor transmitted to any persons or parties on any objective, goal or justification without due written assent, acquiescence or approval by Paladin.

All information provided in this report does not constitute financial or investment advice, nor should it be used to signal that any persons reading this report should invest their funds without sufficient individual due diligence regardless of the findings presented in this report. Information is provided 'as is', and Paladin is under no covenant to the completeness, accuracy or solidity of the contracts audited. In no event will Paladin or its partners, employees, agents or parties related to the provision of this audit report be liable to any parties for, or lack thereof, decisions and/or actions with regards to the information provided in this audit report.

Cryptocurrencies and any technologies by extension directly or indirectly related to cryptocurrencies are highly volatile and speculative by nature. All reasonable due diligence and safeguards may yet be insufficient, and users should exercise considerable caution when participating in any shape or form in this nascent industry.

The audit report has made all reasonable attempts to provide clear and articulate recommendations to the Project team with respect to the rectification, amendment and/or revision of any highlighted issues, vulnerabilities or exploits within the contracts provided. It is the sole responsibility of the Project team to sufficiently test and perform checks, ensuring that the contracts are functioning as intended, specifically that the functions therein contained within said contracts have the desired intended effects, functionalities and outcomes of the Project team.

Paladin retains full rights over all intellectual property (including expertise and new attack or exploit vectors) discovered during the audit process. Paladin is therefore allowed and expected to re-use this knowledge in subsequent audits and to inform existing projects that may have similar vulnerabilities. Paladin may, at its discretion, claim bug bounties from third-parties while doing so.

1 Overview

This report has been prepared for Merchant Moe on the Mantle network. Paladin provides a user-centred examination of the smart contracts to look for vulnerabilities, logic errors or other issues from both an internal and external perspective.

1.1 Summary

Project Name	Merchant Moe
URL	https://merchantmoe.com/
Network	Mantle
Language	Solidity
Preliminary Contract	https://github.com/traderjoe-xyz/moe-core/tree/425673d01967d18afa561909c62dd5d83a748fc8/src https://github.com/traderjoe-xyz/moe-core/tree/cc92e9403f370d1d38536dedfb9a9ceb2352a073
Resolution	https://github.com/traderjoe-xyz/moe-core/tree/d3a52737e77da5b89bb80e5ad0a0642dfa763cec

1.2 Contracts Assessed

Name	Contract	Live Code Match
Moe	0x4515A45337F461A11Ff0FE8aBF3c606AE5dC00c9	✓ MATCH
MasterChef	0xd4BD5e47548D8A6ba2a0Bf4cE073Cbf8fa523DcC	✓ MATCH
MoeStaking	0xE92249760e1443FbBeA45B03f607Ba84471Fa793	✓ MATCH
StableMoe	0x5Ab84d68892E565a8bF077A39481D5f69edAAC02	✓ MATCH
VeMoe	0x240616e2448e078934863fB6eb5133834BF14Ef1	✓ MATCH
BaseRewarder	Dependency	✓ MATCH
MasterChefRewarder	0xcc076c7c657DCAfC738991297903610896d2E938	✓ MATCH
VeMoeRewarder	0x151B82CA3a0c9dA9Dfde200F9C527cD89dd6aea8	✓ MATCH
MoeFactory	0x5bef015ca9424a7c07b68490616a4c1f094bedec	✓ MATCH
MoeERC20	Dependency	✓ MATCH
MoePair	0x08477e01A19d44C31E4C11Dc2aC86E3BBE69c28B	✓ MATCH
MoeRouter	0xeaEE7EE68874218c3558b40063c42B82D3E7232a	✓ MATCH
MoeLibrary	Dependency	✓ MATCH
Amounts	Dependency	✓ MATCH
Math	Dependency	✓ MATCH
Constants	Dependency	✓ MATCH
Rewarder	Dependency	✓ MATCH
JoeStaking	0x7fb0Fc8514D817c655276A2895307176F253D303	✓ MATCH
JoeStakingRewarder	0x1D16326BA904546b4DA88d357Dd556Ebe1f08dD6	✓ MATCH
RewarderFactory	0x18d3F4Df4959503C5F2C8B562da3118939890025	✓ MATCH
VestingContract	Not deployed	N/A

1.3 Findings Summary

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
● Governance	1	-	-	1
● High	3	3	-	-
● Medium	4	4	-	-
● Low	7	6	-	1
● Informational	25	18	1	6
Total	40	31	1	8

Classification of Issues

Severity	Description
● Governance	Issues under this category are where the governance or owners of the protocol have certain privileges that users need to be aware of, some of which can result in the loss of user funds if the governance's private keys are lost or if they turn malicious, for example.
● High	Exploits, vulnerabilities or errors that will certainly or probabilistically lead towards loss of funds, control, or impairment of the contract and its functions. Issues under this classification are recommended to be fixed with utmost urgency.
● Medium	Bugs or issues that may be subject to exploit, though their impact is somewhat limited. Issues under this classification are recommended to be fixed as soon as possible.
● Low	Effects are minimal in isolation and do not pose a significant danger to the project or its users. Issues under this classification are recommended to be fixed nonetheless.
● Informational	Consistency, syntax or style best practices. Generally pose a negligible level of risk, if any.

1.3.1 Global Issues

ID	Severity	Summary	Status
01	GOV	Governance risk	ACKNOWLEDGED
02	INFO	Foundry EVM is not explicitly fixed to Paris	✓ RESOLVED

1.3.2 Moe

No issues found.

1.3.3 MasterChef

ID	Severity	Summary	Status
03	MEDIUM	_updateAll does not mint accumulated rewards nor distribute a treasury share	✓ RESOLVED
04	MEDIUM	Top pools receive less emissions than they should as their emission weight is divided over all votes instead of just over the votes of the top pools which are eligible for emissions	✓ RESOLVED
05	LOW	The pending rewards returned from getPendingRewards are erroneously high as they do not account for the treasury share subtraction on yet to be minted rewards	✓ RESOLVED
06	LOW	MOE may be minted and become stuck in the MasterChef for pools that do not have any stake in them	✓ RESOLVED
07	INFO	Typographical issues	✓ RESOLVED

1.3.4 MoeStaking

ID	Severity	Summary	Status
08	LOW	Not using SafeCast could potentially lead to implicit underflow/overflow errors in an unlikely deployment scenario	✓ RESOLVED
09	INFO	Typographical issues	✓ RESOLVED

1.3.5 StableMoe

ID	Severity	Summary	Status
10	HIGH	StableMoe rewards can be drained through a reentrancy attack in the reward distribution logic if either the gas token or a reentrancy token is added as a reward	✓ RESOLVED
11	LOW	Reward distributions can be frontrun	ACKNOWLEDGED
12	INFO	Gas optimizations	✓ RESOLVED
13	INFO	Typographical issues	✓ RESOLVED

1.3.6 VeMoe

ID	Severity	Summary	Status
14	HIGH	setBribes is vulnerable to a reentrancy exploit where an attacker can receive rewards for their vote from multiple bribe contracts at the same time and, more critically, fully drain all bribes of their rewards	✓ RESOLVED
15	HIGH	The vote function is vulnerable to a reentrancy exploit which allows users to amplify the emissions and drain the rewards from the masterchef	✓ RESOLVED
16	LOW	setTopPoolIds does not update the new pools, causing new pool rewards to be distributed retroactively, potentially resulting in incorrect amounts of MOE being distributed	✓ RESOLVED
17	LOW	Functions such as getPendingRewards can be spoofed as bribes can be arbitrary contracts	✓ RESOLVED
18	INFO	Withdrawals that wipe out the VeMOE balance can be circumvented by staking MOE in a large number of smaller accounts	ACKNOWLEDGED
19	INFO	Typographical issues	✓ RESOLVED

1.3.7 Libraries/BaseRewarder

ID	Severity	Summary	Status
20	LOW	getTotalRewards will return rewards even if there are no stakers in the pool	✓ RESOLVED
21	INFO	Typographical issues	✓ RESOLVED

1.3.8 Libraries/MasterChefRewarder

No issues found.

1.3.9 Libraries/VeMoeRewarder

No issues found.

1.3.10 DEX/MoeFactory

ID	Severity	Summary	Status
22	INFO	Typographical issues	✓ RESOLVED

1.3.11 DEX/MoeERC20

ID	Severity	Summary	Status
23	INFO	Approval event is not emitted if allowance is changed in transferFrom as suggested in the ERC20 Token Standard (this issue is also present in Uniswap)	✓ RESOLVED
24	INFO	permit can be frontrun to prevent someone from calling removeLiquidityWithPermit (also present in Uniswap)	ACKNOWLEDGED

1.3.12 DEX/MoePair

ID	Severity	Summary	Status
25	INFO	Typographical issue	✓ RESOLVED

1.3.13 DEX/MoeRouter

ID	Severity	Summary	Status
26	INFO	The functions for adding liquidity are slightly inefficient for tokens that have a fee on transfer (also present in Uniswap)	ACKNOWLEDGED
27	INFO	Phishing is possible by a malicious frontend by adjusting routes, tokens or from parameters (also present in Uniswap)	ACKNOWLEDGED
28	INFO	Gas optimizations	✓ RESOLVED

1.3.14 DEX/MoeLibrary

No issues found.

1.3.15 Libraries/Amounts

No issues found.

1.3.16 Libraries/Math

No issues found.



1.3.17 Libraries/Constants

No issues found.

1.3.18 Libraries/Rewarder

ID	Severity	Summary	Status
29	INFO	Gas optimizations	✓ RESOLVED
30	INFO	Typographical issues	✓ RESOLVED

1.3.19 JoeStaking

ID	Severity	Summary	Status
31	INFO	The owner should not be able to set a previously-used rewarder as the current one	ACKNOWLEDGED
32	INFO	Typographical issues	PARTIAL

1.3.20 JoeStakingRewarder

ID	Severity	Summary	Status
33	INFO	Typographical issue	✓ RESOLVED

1.3.21 RewarderFactory

ID	Severity	Summary	Status
34	MEDIUM	Blockchain reorg can be abused by a malicious actor to steal funds from creators of VeMoeRewarder contracts	✓ RESOLVED
35	INFO	Deployers of VeMoeRewarder contracts can abuse privileged functions to grief users	ACKNOWLEDGED
36	INFO	Typographical issue	✓ RESOLVED

1.3.22 VestingContract

ID	Severity	Summary	Status
37	MEDIUM	The owner will receive less funds than intended when vesting is revoked	✓ RESOLVED
38	INFO	The MasterChef contract owner can be renounced, potentially resulting in locked funds	✓ RESOLVED
39	INFO	Releasing vested tokens may start reverting due to arithmetic overflow exception	✓ RESOLVED
40	INFO	Typographical issue	✓ RESOLVED



2 Findings



2.1 Global Issues

The issues in this section apply to the protocol as a whole. We have consolidated the global issues to simplify the report.



2.1.1 Issues & Recommendations

Issue #01	Governance risk
Severity	 GOVERNANCE
Description	<p>Most of the contracts within the system are upgradeable, which means that the team behind the codebase can arbitrarily adjust the logic of these contracts via the proxy admin of these contracts.</p> <p>This means that if the ownership of the proxy admin falls into the wrong hands, or if the team is malicious, risks to the protocol include but are not limited to loss of all staked funds in the system. Open approvals could also be compromised and exploited through a malicious upgrade.</p>
Recommendation	<p>Consider carefully designing the proxy admin ownership. Ideally, this is a strong multi-signature set-up of reputable and independent parties.</p> <p>Consider limiting approvals by default to prevent them from being drained if a malicious upgrade does occur.</p> <p>Consider safeguarding the contract owners as well as properties. For example, the MOE emission rate is not capped and the owner could therefore allow for the full MOE supply to be instantly minted. Safeguarded ownership is therefore important as well.</p>
Resolution	 ACKNOWLEDGED
	The team will carefully manage the governance privileges.

Issue #02 Foundry EVM is not explicitly fixed to Paris	
Severity	 INFORMATIONAL
Description	The PUSH0 opcode is not supported currently on Mantle, so Solidity versions 0.8.20 and higher cannot be used with the Shanghai EVM version. As, to our knowledge, foundry currently does not default to Shanghai, this should be fine. However, we still recommend explicitly fixing the EVM version.
Recommendation	Consider fixing the evm_version to paris.
Resolution	 RESOLVED



2.2 Moe

Moe is the governance token for the Moe ecosystem. It is a simple ERC20 token with a capped supply. During deployment, a minter can be configured that can mint up to the token's maximum supply. This minter is expected to be the MasterChef and cannot be changed. A configurable initial supply is also minted during deployment to the deployer.

It should be noted that as the MasterChef is expected to be upgradeable, Moe mintership can effectively be reclaimed as an upgrade to the masterchef could mint the remaining supply at once. Holders should take this into account and ensure that the proxy admins within the system are trustable and secure.

2.2.1 Privileged Functions

- `mint [minter: masterchef]`

2.2.2 Issues & Recommendations

No issues found.



2.3 MasterChef

MasterChef is a staking contract which allows users to stake LP tokens in return for MOE rewards which accrue over time. MOE rewards are distributed over the top-voted pools according to the percentage of VeMOE votes that went to each pool respectively. The MasterChef is also designed to distribute additional rewards on top of these MOE rewards using custom rewarders which can be configured per pool.

During deployment, a treasury share can be configured, where a share of the rewards is sent to the Moe team treasury. This portion of the MOE emission rate does not go to LP stakers.

It should be noted that the MasterChef does not support the staking of tokens with a fee on transfer or other non-standard ERC-20 implementations. These tokens should never be added.

UPDATE: During the resolution round, the minted MOE has been split up from liquidity incentives and treasury share into multiple smaller portions:


- `treasuryAmount`
- `futureFundingAmount`
- `teamAmount`
- `liquidityMiningAmount` (all the remaining MOE)

2.3.1 Privileged Functions

- `setMoePerSecond`
- `setFutureFunding` [ADDED IN RESOLUTIONS]
- `setTeam` [ADDED IN RESOLUTIONS]
- `add`
- `setExtraRewarder`
- `setTreasury`
- `transferOwnership`
- `renounceOwnership`



2.3.2 Issues & Recommendations

Issue #03	_updateAll does not mint accumulated rewards nor distribute a treasury share
Severity	 MEDIUM SEVERITY
Description	<p>The MasterChef allows pool rewards to be indexed via the <code>_updateAll</code> function. This is particularly useful for the VeMOE token as adjustments are often made on which pools are eligible for rewards, and prefacing those adjustments with an index allows the adjustments to not be applied retroactively.</p> <p>Through regular deposits, withdrawals and harvests, rewards get indexed as well. During each of these indexing, newly minted rewards are automatically minted to both the treasury and the masterchef according to the treasury share split. Later on, during harvests, the portion allocated to the contract is then claimed by users.</p> <p>However, this logic is not present within <code>_updateAll</code>, which means that the full reward amount is granted to users. Furthermore, this amount is not minted, meaning that as users claim the rewards from the <code>_updateAll</code> function, the contract's MOE balance will eventually run out.</p> <p>A user can repeatedly call <code>_updateAll</code> in a malicious attempt to drain the contract and cause reputational damage to the client. Another benefit for this user is that they receive the treasury share, instead of this amount going to the treasury (if the actor is staking themselves).</p>
Recommendation	Consider providing a function that appropriately subtracts the treasury share from this amount (and returns both the reward and treasury amount), and consider reusing it in all locations. Consider minting MOE as appropriate.

Resolution



The MOE rewards are now minted in `_updateAll` as they should. However, an error was made in the newly introduced `_mintMoe` function as it does not account for the case where the actual minted amount of MOE is less than the amount passed or 0. This could occur when the maximum MOE supply is reached, resulting in the same impact described above. This portion has been resolved in an additional resolution round, marking this issue as fully resolved.

Issue #04

Top pools receive less emissions than they should as their emission weight is divided over all votes instead of just over the votes of the top pools which are eligible for emissions

Severity



Location

```
L210-212  
uint256 totalVotes = _veMoe.getTotalVotes();  
return totalVotes == 0 ? 0 : _moePerSecond *  
_veMoe.getVotes(pid) / totalVotes;
```

Description

The top-voted pools within the protocol are eligible for MOE emissions within the MasterChef. Their share of these emissions, which are at a fixed rate per second, is determined based on the number of votes each individual pool has.

However, the business logic of the MasterChef incorrectly bases this share as the share of all votes. This means that emissions will be subtracted due to votes being made on pools that did not make it to the top.

Recommendation

Consider whether it is safe and/or feasible to replace `getTotalVotes` in this location with for example `getTopPidsTotalVotes`. Such a replacement must be carefully tested against synchronization issues as the top pids often change.

All other locations where this occurs should be fixed as well.

Resolution




The total votes of the top pool IDs are now used in the correct places.

Issue #05

The pending rewards returned from `getPendingRewards` are erroneously high as they do not account for the treasury share subtraction on yet to be minted rewards

Severity

 LOW SEVERITY

Description

The MasterChef contains a `getPendingRewards` function which is used primarily by the frontend to show the rewards that will be harvested when the user claims their next rewards.

These pending rewards, however, do not account for the fact that a treasury share will be subtracted from the additionally-minted portion (the amount of rewards that will be minted since the last timestamp any user harvested this pool).

This effectively means that `getPendingRewards` will consistently overestimate the pending rewards.

Recommendation


Consider providing a function that appropriately subtracts the treasury share from this amount (and returns both the reward and treasury amount), and consider reusing it in all locations.

Resolution

 RESOLVED

The fees are no longer included in the returned amounts of pending rewards.



Issue #06**MOE may be minted and become stuck in the MasterChef for pools that do not have any stake in them****Severity** LOW SEVERITY**Description**

The MasterChef will mint MOE according to any MOE which is owed to reward pool stakers (the top VE pools). Essentially the MOE rewards for any individual pool are distributed amongst all depositors of that pool.

However, the very first staker (after everyone has withdrawn) in a pool will still cause rewards to be minted to that pool for the stakers that staked over the period when the pool was empty.

```
uint256 totalMoeRewardForPid = _getRewardForPid(rewarder, pid);
```

This line would still return a positive number and cause MOE to be minted. However, within the actual rewarder we see the following logic:

```
return totalDeposit == 0 ? 0 : (totalRewards << Constants.ACC_PRECISION_BITS) / totalDeposit;
```

This logic indicates that the actual rewards are not accrued anywhere and are just burned (which makes sense as these rewards cannot be allocated to anyone). This means that these minted MOE tokens are effectively stuck in the MasterChef and were minted for no reason except for the treasury share.

Recommendation

Given that the MasterChef is upgradeable, these stuck rewards could eventually be taken out. There is no reason to fix this because of that. If desired, the minting logic could be adjusted to not mint MOE if the total deposit is 0.

Resolution RESOLVED

MOE rewards are no longer minted to pools with 0 deposits.



Issue #07	Typographical issues
Severity	<div data-bbox="454 203 486 232"></div> INFORMATIONAL
Description	<p data-bbox="454 286 518 315"><u>L7-8</u></p> <p data-bbox="454 331 751 405">OwnableUpgradeable, Initializable</p> <p data-bbox="454 443 884 472">These imports appear unused.</p> <p data-bbox="454 562 518 591"><u>L157</u></p> <p data-bbox="454 607 1238 636">Amounts.Parameter storage amounts = farm.amounts;</p> <p data-bbox="454 674 1380 801">This statement can be moved up as farm.amounts is referenced explicitly earlier in the function. That reference could be replaced with this amounts reference to tidy up the function.</p> <p data-bbox="454 891 518 920"><u>L280</u></p> <p data-bbox="454 936 916 965">_moePerSecond = moePerSecond;</p> <p data-bbox="454 1003 1418 1131">It might make sense to cap the maximum amount for this variable to safeguard the minting role from being maliciously used to drain the outstanding supply.</p> <p data-bbox="454 1220 518 1249"><u>L380</u></p> <p data-bbox="454 1265 1366 1294">if (pid >= nbOfFarms) revert MasterChef__InvalidPid(pid);</p> <p data-bbox="454 1384 518 1413"><u>L417</u></p> <p data-bbox="454 1429 1299 1503">if (moeReward > 0) IERC20(_moe).safeTransfer(account, moeReward);</p> <p data-bbox="454 1541 1369 1615">This line can be simplified by adding using SafeERC20 for IMoe to the contract.</p>
Recommendation	Consider fixing the typographical issues.
Resolution	<div data-bbox="454 1715 486 1744"></div> RESOLVED



2.4 MoeStaking

MoeStaking allows users to stake their MOE tokens. When users stake their MOE within the contract, the contract will notify both the StableMoe and VeMoe contracts about this increase in stake. These two subcontracts therefore contain the actual business logic for staking rewards.



2.4.1 Issues & Recommendations

Issue #08	Not using SafeCast could potentially lead to implicit underflow/overflow errors in an unlikely deployment scenario
Severity	 LOW SEVERITY
Location	<u>Lines 87 and 97</u> <code>_modify(msg.sender, int256(amount));</code> <code>_modify(msg.sender, -int256(amount));</code>
Description	Within two locations of the code, implicit downcasting of amount is done. This could lead to overflow on the sign in the unlikely case where MOE has a larger supply than the maximum int256 value.
Recommendation	We recommend simply using checked casting to avoid any issues as the additional gas cost here is minimal. We do understand that realistically the MOE supply will be way too low for this overflow to occur.
Resolution	 RESOLVED The amounts are now casted using a safe cast.

Issue #09	Typographical issues
Severity	 INFORMATIONAL
Description	<u>Lines 6 and 20</u> <code>import {Math} from "./libraries/Math.sol";</code> <code>using Math for uint256;</code> The Math import appears unused and can be removed. — getMoe, getVeMoe and getSMoe return their explicit type instead of address within other contracts such as the MasterChef. We recommend being consistent and perhaps also doing that here.
Recommendation	Consider fixing the typographical issues.
Resolution	 RESOLVED

2.5 StableMoe

StableMoe is a reward processor for the MoeStaking contract. In other words, it is responsible for granting rewards to users who stake into MoeStaking. The rewards distributed by this contract are derived from the DEX volume and more specifically the governance share of the swap fees.

StableMoe is capable of distributing a large set of tokens at a time, though the team will likely still swap certain swap fee tokens into one of these supported tokens.


To be gas efficient, the contract owner can delete a token again from the list of reward tokens. However, this can only be done once, which means that once the owner decides to retire a token from being an active reward, it can never be re-added again (except through a contract upgrade).

It should be noted that StableMoe does not support tokens with certain non-standard ERC-20 implementations such as negatively rebasing tokens.

2.5.1 Privileged Functions

- `addReward`
- `removeReward`
- `sweep`
- `transferOwnership`
- `renounceOwnership`

2.5.2 Issues & Recommendations

Issue #10	StableMoe rewards can be drained through a reentrancy attack in the reward distribution logic if either the gas token or a reentrancy token is added as a reward
Severity	 HIGH SEVERITY
Location	<p>L217-237</p> <pre>function _claim(address account, uint256 oldBalance, uint256 newBalance, uint256 totalSupply) private { uint256 length = _activeRewardIds.length(); for (uint256 i; i < length; ++i) { [...] uint256 rewards = reward.rewarder.update(account, oldBalance, newBalance, totalSupply, totalRewards); [...] _safeTransferTo(token, account, rewards); [...] } }</pre>
Description	<p>StableMoe facilitates the distribution of a large set of reward tokens all at once when users claim their rewards. This is useful as it allows users to, for example, receive MOE, USDC, and even the native gas token all at the same time.</p> <p>However, within this reward distribution logic, a critical error was made. As can be seen in the above code snippet, <code>oldBalance</code> and <code>newBalance</code> are cached throughout the <code>claim</code> loop. However, during this loop, an interaction occurs as the (gas) token gets transferred to the account. Specifically for the gas token and tokens with reentrancy, this allows for users to make a call into the system again.</p> <p>When a malicious actor makes such a call to withdraw their whole balance from the <code>MoeStaking</code> contract, this will succeed. However, the rest of the loop still completes with the cached <code>oldBalance</code> and <code>newBalance</code>, which means that the user can continue harvesting as if they still have a balance.</p>

This becomes particularly critical as during the reentrancy withdraw, the tracking variable which tracks how much rewards the user has already harvested gets reset (the reward debt of the account) because their balance is back to 0. This means that in the course of the rest of the actual reward loop with the outdated balances, those rewards will be harvested as if the user has no reward debt, meaning their rewards are massively amplified as if they had staked from day 1 and never claimed a single reward.

Looping this exploit allows the exploiter to effectively drain all the rewards from the StableMoe contract, resulting in significant financial and reputational damage.

Proof of concept

Let's assume StableMoe presently distributes two rewards: the Mantle gas token and USDC. Let's assume Alice (an exploit contract) has a balance of 10 MOE staked into MoeStaking with a total of 20 MOE being staked in the contract (`balance = 10`, `totalSupply = 20`).

1. Alice call `claim()` — at this time, `claim` records Alice's balance as 10 and the `totalSupply` as 20 for the whole loop over all reward tokens.
2. Alice harvests the ETH rewards, which are immediately sent to her contract.
3. On receiving the ETH, Alice is able to execute arbitrary code in the reentrancy callback. Alice calls `MoeStaking.withdraw(10)` which withdraws her whole balance and calls `StableMoe.onModify`.
 - a. StableMoe does a full harvest in the reentrancy callback, updating all the reward details as if Alice now has 0 MOE, which she does. The reward debt of alice is set to 0 as well.
 - b. The reentrancy is finished and the outer `claim()` goes onto the next token.
4. Next, USDC is harvested while the balance of Alice is still cached at 10 even though it is actually 0.

-
5. `getDebt(accDebtPerShare, oldBalance)` - `rewarder.debt[account]`; is used to calculate the pending USDC rewards, but as mentioned earlier, all account debt have now been reset to 0, thus this calculation returns an insanely high amount which allows Alice to drain the USDC balance in a few iterations.
 6. Alice ends up claiming a significantly larger amount of USDC than she was eligible for, and loops the exploit until the USDC `StableMoe` balance is drained.

It should be noted that this exploit can also be executed by staking extra MOE during the reentrancy, as this would cause the final reward debt to be too low, thus increasing the rewards for the next harvest.


It should be noted that privileged reentrancies, such as the owner reentering into `addReward` or `removeReward`, might also cause issues and privilege escalation. If the code is kept as-is for some reason, this should be kept in mind and the owner role should be treated carefully.

Recommendation Consider writing the function to adhere to checks-effects-interactions where the transfers are all done at the end of the function. Alternatively, the function could be looped on the condition that no state is cached between the iterations. Finally, reentrancy guards could be considered if desired — these must however be multi-contract in this instance.

Resolution



The checks-effects-interactions pattern is now followed correctly. All transfers were moved out of the loop and no state variables are cached between the external calls.

Issue #11**Reward distributions can be frontrun****Severity** LOW SEVERITY**Description**

Rewards can be distributed amongst MoeStaking stakers by simply sending them to the StableMoe contract to instantly distribute them according to their total MOE deposit.

However, an exploiter can sandwich a reward distribution with a deposit and withdraw of MOE tokens to claim a significant portion of the rewards without actually being a full-time staker.

This could for example occur if there is a lending market for MOE tokens. Furthermore, if reward distributions can be triggered by anyone, this could be done via a flashloan.

Recommendation



Consider whether this will be an issue. The client might be able to address this by making distributions frequent and low value enough in the short term.



A proper solution could be having frictions to deposit/withdrawal such as a minimum stake time or a deposit/withdrawal fee.

A complete solution would distribute the reward over a distribution period (e.g. such as the Synthetix StakingRewards contract does).

Resolution ACKNOWLEDGED

If this becomes an issue, the team indicated that they will address this later as StableMoe is an upgradeable proxy.

Issue #12	Gas optimizations
Severity	 INFORMATIONAL
Location	<u>L31</u> EnumerableSet.UintSet private _activeRewardIds;
Description	An AddressSet might be a more sensible data structure here and be more gas efficient on the critical path as the key of the set would also contain the reward address, resulting in one less storage slot being read per asset.
Recommendation	Consider implementing the gas optimizations mentioned above.
Resolution	 RESOLVED

Issue #13	Typographical issues
Severity	 INFORMATIONAL
Description	<p>L8, 9, 12 and 25</p> <pre>OwnableUpgradeable, Initializable import {Math} from "../libraries/Math.sol";</pre> <p>These lines appear to be unused.</p> <p>—</p> <p>We should note that timestamp accounting occurs during claims within the rewarders, which is not actually used for anything. We do not recommend fixing this as it does not do any harm and might be complex to fix.</p>
Recommendation	Consider fixing the typographical issues.
Resolution	 RESOLVED

2.6 VeMoe

VeMoe allows users to generate VeM0E over time as they stake their M0E tokens. Their VeM0E balance will grow linearly according to the configured `veMoePerSecondPerMoe` variable until the `maxVeMoePerMoe` is reached. This means that users will accumulate VeM0E over time as they stake, but up to a maximum amount. As soon as users withdraw any M0E, all accumulated VeM0E will be reset (i.e. the user's VeM0E balance goes to zero).

Users can use their VeM0E to vote for masterchef pools. The top pools with the most votes will be granted M0E emissions according to the number of votes received. The top pools are manually picked by governance. Once users have voted, they can allocate their votes to arbitrary bribe contracts. This makes them eligible for additional bribe rewards for voting on a specific pool. Users can only collect bribes from one contract at a time but as these contracts can be literally anything, this contract could eventually aggregate multiple bribes into one.

We would like to re-iterate that bribe contracts can be anything and can therefore also misrepresent their rewards. Specifically, the frontend should carefully vet the eligible bribe contracts.

Also, voters can switch between bribes at any time as there is no concept of epochs (bribes and votes affect emissions continuously), which means that bribe contracts should be designed while keeping this in mind (e.g. bribe rewards should be released continuously as well).

Only the top pools will be eligible for masterchef emissions, however, bribes may still incentivize voting on pools which are not yet in the top.

The owner of the contract can freely adjust the rate at which VeM0E gets accumulated by stakers. However, the maximum multiplier of VeM0E to M0E is fixed. VeM0E must be manually claimed, and it must be manually voted to a pool whenever it is claimed.

2.6.1 Privileged Functions

- `setTopPoolIds`
- `setVeMoePerSecondPerMoe`
- `transferOwnership`
- `renounceOwnership`



2.6.2 Issues & Recommendations

Issue #14	setBribes is vulnerable to a reentrancy exploit where an attacker can receive rewards for their vote from multiple bribe contracts at the same time and, more critically, fully drain all bribes of their rewards
------------------	--

Severity

 HIGH SEVERITY

Description

VeMoe allows users to delegate votes to any bribe contract. This bribe contract will then reward them for voting on the pool.

The VeMoe system has been designed to allow users to configure at most one bribe contract per pool they have voted on. All votes to that pool the user has created will then be eligible for the bribe rewards of that bribe contract. The setBribes function which allows users to configure their vote's bribe contract.

Unfortunately, the way setBribes is implemented is vulnerable. Specifically, there is a vulnerability in how it unsets the old bribe, as unsetting this old bribe could allow reentrancy back into the setBribes function and set a new bribe. Through this reentrancy where the old bribe gets unset, a new bribe is configured. However, once the reentrancy is done, the original setBribes call completes and a new bribe is also configured. This means that the user has effectively configured two active bribes for their votes.

Finally, and this makes the exploit critical: A reentrancy allows for a bribe deposit to be unset before it is set. This means that the bribe will receive a message that the user has withdrawn their balance from the bribe before the user has even deposited. Due to the way reward debt is managed, the whole balance of the rewarder can be drained through an iterative exploit.

It should finally be noted that the user can potentially also reenter in vote and execute further exploits.

Proof of concept

Assume that Alice has a malicious bribe contract "B1" and aims to convince two genuine bribe contracts "B2" and "B3" that she voted using their bribe. Her goal is therefore to be able to claim rewards from both contracts at the same time for her votes, even though she should only be allowed to claim from one at a time as they are both for the same pool.

1. Alice calls `setBribes(pool, B1)`. This sets B1 as Alice's pool and calls `B1.onModify(old: 0, new: aliceVotes)`. Alice does not yet reenter.
2. Alice calls `setBribes(pool, B2)`. This sets B2 as Alice's pool and first calls `B1.onModify(old: aliceVotes, new: 0)`. Alice reenters here.
3. Alice calls `setBribes(pool, B3)`. This sets B3 as Alice's pool and first calls `B2.onModify(old: aliceVotes, new: 0)`. This drains significant rewards from B2 due to Alice's reward debt still being zero at this point. Through iterating this exploit, all bribes will be fully drained in this step, and all rewards will be in the hands of Alice. Next, `B3.onModify(old: 0, new: aliceVotes)` is called.
4. As the reentrancy completes, the second section of the `setBribes(pool, B2)` executes and `B2.onModify(old: 0, new: aliceVotes)`.

The end state is that Alice has drained significant rewards from pool B2 and has marked herself as a stalker of both B2 and B3, resulting in massive financial and reputational damage to the protocol and bribers.

Test-based PoC (Credits to [@Louis_Mslf](#) for confirming this with an executable PoC)

```

function test_ReenterBribes() public {
    MaliciousBribe maliciousBribe = new MaliciousBribe(veMoe);
    moe.mint(address(maliciousBribe), 1e18);

    veMoe.setVeMoePerSecondPerMoe(1e18);

    MockERC20(address(token18d)).mint(address(bribes0), 100e18);
    bribes0.setRewardPerSecond(1e18, 100);

    vm.prank(alice);
    staking.stake(1e18);

    uint256[] memory pids = new uint256[](1);
    int256[] memory deltaAmounts = new int256[](1);
    IVeMoeRewarder[] memory bribes = new IVeMoeRewarder[](1);

    vm.startPrank(address(maliciousBribe));
    moe.approve(address(staking), type(uint256).max);
    staking.stake(1e18);

    vm.warp(block.timestamp + 50);

    pids[0] = 0;
    deltaAmounts[0] = 1e18;
    bribes[0] = IVeMoeRewarder(address(maliciousBribe));

    veMoe.vote(pids, deltaAmounts);
    veMoe.setBribes(pids, bribes);

    bribes[0] = bribes0;

    veMoe.setBribes(pids, bribes);

    vm.stopPrank();
}

```

```

contract MaliciousBribe {
    IVeMoe public veMoe;

    constructor(IVeMoe _veMoe) {
        veMoe = _veMoe;
    }

    function onModify(address, uint256 pid, uint256 oldBalance, uint256, uint256) external {
        if (oldBalance == 0) return;

        uint256[] memory pids = new uint256[](1);
        IVeMoeRewarder[] memory bribes = new IVeMoeRewarder[](1);

        pids[0] = pid;
        bribes[0] = IVeMoeRewarder(address(this));

        veMoe.setBribes(pids, bribes);
    }
}

```

[illegible]

Recommendation

Making multiple generic rewarder updates reentrancy proof is a challenging endeavor. We do not have a perfect solution for this.

Consider whether it makes sense to prevent reentrancy on this critical path by only allowing a certain bribe contract instead of generic ones. It should be noted that reentrancy tokens such as the gas token could still be granted permitting for this exploit. In this case, the critical path should not carry out any reentrancy interactions (e.g. token transfers) on this `onModify` call. Instead, tokens should only be claimable with the `claim` function.

Consider adding global reentrancy guards to the system that prevent these functions from being called in a reentrancy fashion. Note that oftentimes read-only reentrancy remains possible and this is not an elegant solution either.

Resolution



The checks-effects-interactions pattern is now followed correctly. The `claim` and `onModify` functions are now separated. Additionally, the bribe contracts are also now trusted as they can only be the ones deployed by the `RewarderFactory`.

It should be noted that an interaction with the rewarder still occurs. However, as the rewarder is now trusted, the team has indicated they will never do any reentrancy interactions within these specific functions. Advanced users are strongly advised to validate this.

Issue #15

The vote function is vulnerable to a reentrancy exploit which allows users to amplify the emissions and drain the rewards from the masterchef

Severity

 HIGH SEVERITY

Description

The VeMoe contract allows users to delegate votes to any bribe contract. This bribe contract will then reward you for voting on the pool. Whenever votes are delegated, the configured bribe for that user is called to notify that the votes to that bribe are adjusted (these can be decreased or increased as the user can withdraw votes from a pool or add them to a pool).

The issue with this logic implementation is that the user can vote on many pools at the same time. This is inherently not a problem but as bribes are generic contracts, the user can configure a bribe as a reentrancy contract.

The critical issue is due to the way the vote function is written — when the user reenters on their malicious bribe contract and calls vote again, the internal reentrancy vote's `topPidsTotalVotes` configuration will not persist as the outer call simply overwrites it. This is because the inner portions of the vote loop do not adhere to checks-effects-interactions.

This issue can be abused by removing votes in the outer loop and then adding them in the inner loop. When this is done, the votes are unchanged but `topPidsTotalVotes` gets decremented. This allows for the attacker to eventually decrement the total votes to exactly "1" even though their votes might be millions of times larger. When such an amplification attack is fully finalized, the attacker can then execute a harvest on the masterchef with a proportional multiplier of their votes compared to the total votes, which will instantly drain the masterchef of all rewards.

Proof of concept

1. Alice calls `vote([pid0, alice VeM0E]`
`_topPidsTotalVotes = alice VeM0E;`
2. Alice configures her bribe for `pid0` to be a malicious reentrancy contract.
3. Alice calls `vote([pid0], [-alice VeM0E + 1])` to vote on the malicious PID, however, she aims to leave "1" remaining.
4. As this is called, her malicious contract is eventually notified that it now has a changed amount of votes.
 - a. REENTER: Alice votes again.
 - b. `_topPidsTotalVotes = 2*aliceVeMoe;` as the vote withdrawal has not been accounted for yet.
5. As the reentrancy finishes, `_topPidsTotalVotes` gets overwritten to exactly "1".

The end state is that the `topPidsTotalVotes` is "1" even though Alice has an arbitrary number of votes. When Alice harvests, her harvest will be amplified millions of times and she will drain the masterchef of all M0E, completing the exploit and leaving the protocol with massive financial and reputational damage.

Recommendation	Consider executing each vote as an independent checks-effects-compliant unit.
-----------------------	---

Resolution




The checks-effects-interactions pattern is now followed correctly. All claims were moved out of the loop. Additionally, the bribe contracts are now trusted as they can only be ones deployed by the `RewarderFactory`.

It should be noted that an interaction with the rewarder still occurs. However, as the rewarder is now trusted, the team has indicated they will never do any reentrancy interactions within these specific functions. Advanced users are strongly advised to validate this.

Issue #16

setTopPoolIds does not update the new pools, causing the new pool rewards to be distributed retroactively, potentially resulting in incorrect amounts of MOE being distributed

Severity

 LOW SEVERITY

Description

VeMoe allows a set of top pools that receive MOE emissions to be defined. The MOE team can adjust this set of top pools at any time by calling setTopPoolIds.

This issue is only rated as low as the retroactive distribution is limited to the time between the setTopPoolIds call and the last deposit or withdrawal into the pool before the call. If there is any scenario where a top pool would be configured without it having recent deposits or withdrawals, this issue would be rated as high severity.

Recommendation


Consider updating the new top pools as well.

Resolution

 RESOLVED

The new top pools are now updated as well.



Issue #17**Functions such as `getPendingRewards` can be spoofed as bribes can be arbitrary contracts****Severity** LOW SEVERITY**Description**

VeMoe is designed to allow arbitrary contracts to be configurable as bribes. This has the advantage that arbitrary bribing logic can be created over time. However, the disadvantage is that it significantly increases the attack vectors of the codebase while reducing the assertions that can be made over bribes.



One such assertion is that the values reported are genuine. For example, when a user calls `getPendingRewards` in the VeMoe contract, the contract will query all their bribes to check how many rewards the user has pending. These contracts are free to respond with any amount as bribes can have arbitrary logic. This means the response could be that the user has 10 million USDC pending, even though this is not the case at all.



Recommendation

Consider carefully filtering the bribes that are shown on the frontend. Consider whether it makes sense to do a whitelisted approach to reduce the attack vectors.

Resolution RESOLVED

Users can now only add bribes deployed by the `RewarderFactory` contract.

Issue #18	Withdrawals that wipe out the VeMOE balance can be circumvented by staking MOE in a large number of smaller accounts
Severity	 INFORMATIONAL
Description	Even if a user withdraws just a tiny amount of their staked MOE, their whole VeMOE balance will be zeroed out. However, more sophisticated actors might stake their MOE into a large number of smaller accounts to be able to withdraw MOE as they please without having their whole VeMOE balance reset. As there is no downside to doing this, less sophisticated actors will be disadvantaged.
Recommendation	This is inherent to the current design and cannot be easily fixed from our perspective.
Resolution	 ACKNOWLEDGED

Issue #19	Typographical issues
Severity	 INFORMATIONAL
Description	<p><u>L8-9</u></p> <p>OwnableUpgradeable, Initializable</p> <p>These imports are unused and can be deleted.</p> <p><u>L193-194</u></p> <p>* @dev Returns all the top pool IDs. * @return The top pool IDs.</p> <p>These comments are outdated.</p>
Recommendation	Consider fixing the typographical issues.
Resolution	 RESOLVED

2.7 Libraries/BaseRewarder

BaseRewarder defines shared logic between the MasterChef secondary rewarders and the bribe rewarders used in the VeMoe codebase.

The owner of the rewarder has significant control over the rewards and can stop emissions and retrieve all reward tokens.



Once the rewarder is stopped by the owner, people need to forcefully exit the rewarder without calling it by using the emergency functions as the `onModify` function will revert by design at that point. All accumulated rewards which have not yet been harvested at that point will be void. This might negatively affect users who do not frequently harvest. Users should keep this in mind.

It should be noted that there is no treasury share for secondary rewards.

2.7.1 Privileged Functions

- `setRewarderParameters`
- `setRewardPerSecond`
- `stop`
- `sweep`
- `transferOwnership`
- `renounceOwnership`

2.7.2 Issues & Recommendations

Issue #20	getTotalRewards will return rewards even if there are no stakers in the pool
Severity	 LOW SEVERITY
Description	<p>getTotalRewards will indicate that rewards need to be allocated even if there are no stakers into the pool yet. These rewards will become stuck in <code>_totalUnclaimedRewards</code> until the rewarder is retired.</p>
Recommendation	<p>Consider whether this is an issue. It may very well be fine as it keeps the contract simpler. If desired, these new rewards can be ignored from being added to <code>_totalUnclaimedRewards</code> if the old supply is zero.</p> <p>A cleaner solution would be to return the amount of allocated rewards in <code>_rewarder.update</code>, as not allocating the reward is a side-effect and assuming the side-effect without explicitly forwarding reduces code quality.</p>
Resolution	 RESOLVED

Issue #21	Typographical issues
Severity	<div data-bbox="456 203 485 232"></div> INFORMATIONAL
Description	<p data-bbox="448 286 596 315"><u>L7 and 17</u></p> <pre data-bbox="448 331 1142 409">import {Math} from "../libraries/Math.sol"; using Math for uint256;</pre> <p data-bbox="448 450 1326 479">These imports and definitions are unused and can be deleted.</p> <p data-bbox="448 573 517 602"><u>L113</u></p> <pre data-bbox="448 618 1238 647">* @dev Sets the start of the reward distribution.</pre> <p data-bbox="448 687 1390 766">The documentation for this function appears incomplete as not all parameters are documented.</p> <p data-bbox="448 860 517 889"><u>L121</u></p> <pre data-bbox="448 904 1206 934">if (!_isStopped) revert BaseRewarder__Stopped();</pre> <p data-bbox="448 974 1382 1052">This line appears to be redundant as <code>_setRewardParameters</code> also checks this.</p> <p data-bbox="448 1104 485 1120">—</p> <p data-bbox="448 1171 847 1200">sweep and stop lack events.</p>
Recommendation	Consider fixing the typographical issues.
Resolution	<div data-bbox="456 1312 485 1344"></div> RESOLVED



2.8 Libraries/MasterChefRewarder

MasterChefRewarder is the recommended implementation for MasterChef secondary rewarders. It extends the BaseRewarder and adds functionality to link and unlink the rewarder from the masterchef. The stop() function now explicitly reverts as linking and unlinking is the only mechanism that should be used for this.

2.8.1 Privileged Functions

- setRewarderParameters
- setRewardPerSecond
- sweep
- transferOwnership
- renounceOwnership

2.8.2 Issues & Recommendations

No issues found.



2.9 Libraries/VeMoeRewarder

VeMoeRewarder is the recommended implementation for VeMoe bribe rewarders. It extends the BaseRewarder and simply adds the functionality to query the totalsupply correctly from the VeMoe contract.

2.9.1 Privileged Functions

- `setRewarderParameters`
- `setRewardPerSecond`
- `stop`
- `sweep`
- `transferOwnership`
- `renounceOwnership`

2.9.2 Issues & Recommendations

No issues found.



2.10 DEX/MoeFactory

MoeFactory facilitates the deployment of new MoePair instances. It also defines the fee recipient for the swap protocol fees.



Clones with immutable variables are used for deployment — these custom clone contracts are out of scope and users should carefully validate them or research their past audit coverage as Trader Joe has used these in the past to our knowledge (they come from the Trader Joe repository).

New pairs can be deployed using the createPair function.

2.10.1 Privileged Functions

- setFeeTo
- transferOwnership
- renounceOwnership

2.10.2 Issues & Recommendations



Issue #22	Typographical issue
Severity	 INFORMATIONAL
Description	setFeeTo lacks an event.
Recommendation	Consider fixing the typographical issue.
Resolution	 RESOLVED

2.11 DEX/MoeERC20

MoeERC20 is the underlying ERC20 token implementation for MOE pair LPs. It contains the functions which allow for the LP tokens to be transferred and queried. It is a direct fork of Uniswap V2 with no changes made except for the Solidity version and token metadata being updated.



2.11.1 Issues & Recommendations

Issue #23	Approval event is not emitted if allowance is changed in transferFrom as suggested in the ERC20 Token Standard (this issue is also present in Uniswap)
Severity	 INFORMATIONAL
Description	<p>The ERC-20 standard specifies that an approval event should be emitted when the allowance of a user changes. However, within the ERC20 implementation of both Uniswap and here, this is not done.</p> <p>You can read more about this improvement in Pull Request #65 of uniswap-core.</p>
Recommendation	Consider emitting an event.
Resolution	 RESOLVED

Issue #24

permit can be frontrun to prevent a user from calling `removeLiquidityWithPermit` (this issue is also present in Uniswap)

Severity

 INFORMATIONAL

Description

If `permit` is executed twice, the second execution will revert. It is thus in theory possible for a bot to pick up `permit` transactions in the mempool and execute them before a contract can.


The implications of this issue is that a malicious actor could prevent a user from removing liquidity with a `permit` through the router. It is a denial of service attack which is present in all AMMs but which we have yet to witness being used since there is no profit from this.

Recommendation

Consider this issue if there are ever complaints by users that their `removeLiquidityWithPermit` transactions are failing. It could be the case that someone is using this vector against them.

We do not recommend changing this behavior since it would involve a lot of extra work modifying the frontend to account for new `permit` behavior. This issue is also present in Uniswap after all.

Resolution

 ACKNOWLEDGED



2.12 DEX/MoePair

MoePair is the core contract for any individual LP within the Moe DEX. Moe Pairs hold all DEX liquidity and are therefore the most critical component. It extends MoeERC20 and allows contracts to add liquidity, remove liquidity and swap. Users should use the MoeRouter contract for these functions as it provides additional important checks on these operations.

MoePair extends the TJ Clone dependency which is out of scope for this audit. We recommend that readers validate the audit coverage for this contract as well as Paladin can not make assertions over it within this audit.

MoePair is a close fork of Uniswap V2, however, instead of minting the protocol fee, it is sent to the protocol recipient instead. The contract has also been adjusted to use clones with immutable variables (again, these dependencies are out of scope and users should carefully validate these clones) and has been updated to the more recent 0.8 Solidity version.

The factory owner can sweep tokens accidentally sent to the pair. The two main however tokens cannot be swept.

2.12.1 Privileged Functions

- sweep [factory owner]

2.12.2 Issues & Recommendations

Issue #25	Typographical issue
Severity	<div><div></div>INFORMATIONAL</div>
Description	token0() and token1() can be marked as external.
Recommendation	Consider fixing the typographical issue.
Resolution	<div><div></div>RESOLVED</div>



2.13 DEX/MoeRouter



MoeRouter is the user-interface contract into the Moe DEX core. It provides user-facing functions to add and remove liquidity, and execute swaps.

It is nearly identical to the Uniswap V2 router with the sole changes being moving the OpenZeppelin's SafeERC20, renaming error messages and using Solidity v0.8.



2.13.1 Issues & Recommendations

Issue #26	The functions for adding liquidity are slightly inefficient for tokens that have a fee on transfer (also present in Uniswap)
Severity	 INFORMATIONAL
Description	MoeRouter supports adding liquidity to LPs where one or both of the tokens have a fee on transfer. However, due to the way Uniswap implements these functions, this will result in too much of the token with no tax being sent to the pair, which effectively causes the pair to incorporate the accidental extra tokens into the reserves.
Recommendation	Consider whether this is an issue. We can point to a GitHub PR with a proposed alternative function if desired. This function wastes significant extra gas and does not exist within Uniswap so might not be worth it.
Resolution	 ACKNOWLEDGED

Issue #27	Phishing is possible by a malicious frontend by adjusting routes, tokens or from parameters (also present in Uniswap)
Severity	 INFORMATIONAL
Description	<p>A malicious (for example hacked) frontend can easily mislead users into approving malicious transactions, even if the router matches the address described in this report.</p> <p>An obvious example of how this can be done is by changing the to parameter which indicates to whom tokens or liquidity is to be sent. Other ways to phish could include using malicious routes or tokens.</p>
Recommendation	Consider carefully protecting the frontend and ideally having an unchangeable IPFS fallback implementation for it.
Resolution	 ACKNOWLEDGED

Severity

 INFORMATIONAL

Description

`pairFor` no longer seems to serve a purpose as initially it was used within Uniswap to avoid making external calls. However, `implementation()` is called within the router anyway. It might be better to just call a `get pair` function on the factory instead of using deterministic calculation which tightly couples the factory and the router.

Recommendation

Consider implementing the gas optimizations mentioned above.

Resolution

 RESOLVED

2.14 DEX/MoeLibrary

MoeLibrary is a utilities library used by MoeRouter. It defines various utility functions to calculate swap rates and figure out pair addresses.

2.14.1 Issues & Recommendations

No issues found.



2.15 Libraries/Amounts

Amounts is a library used by several staking contracts within the Moe protocol to keep track of the amount of tokens a user deposits in any pool or contract. It also tracks the total amount of tokens staked of all of these pools.

The library is useful as it abstracts away the logic for fetching and updating these values.

2.15.1 Issues & Recommendations

No issues found.



2.16 Libraries/Math

Math is a library is used for defining two common math operations used throughout the codebase:

- `addDelta(uint256 x, int256 delta)`: A gas-efficient way of adding a signed integer to an unsigned integer. The operation will revert if overflow or underflow occurs. Additionally, as a gas optimization, it restricts the resulting integer to at most the maximum value of a signed 256 bit integer. This is done as a gas optimization as none of the values this is used for need larger amounts.
- `toInt256(uint256 x)`: A gas-efficient way of converting an unsigned integer to its signed alternative. This function explicitly reverts if the input is larger than the maximum signed integer.

2.16.1 Issues & Recommendations

No issues found.

2.17 Libraries/Constants

Constants is a library used to define various constants used throughout the codebase. It is an extremely simple and brief dependency.

2.17.1 Issues & Recommendations

No issues found.




2.18 Libraries/Rewarder



Rewarder is a library that aggregates common reward calculation logic into a dependency used by MasterChef, BaseRewarder, MoeStaking, StableMoe, and VeMoe.

It is inspired by the Synthetix StakingRewards logic and the Sushi masterchef, though the reward debt calculation more closely resembles the original Sushi masterchef. Regardless, the codebase has made significant improvements in code quality compared to Sushi by modularizing this component.



2.18.1 Issues & Recommendations

Issue #29	Gas optimizations
Severity	 INFORMATIONAL
Description	<p><u>L63</u></p> <pre>return lastUpdateTimestamp > timestamp ? 0 : (timestamp - lastUpdateTimestamp) * rewardPerSecond;</pre> <p>This statement still executes advanced arithmetic even if the timestamps are equal, causing wasted gas. Consider inverting the check instead:</p> <pre>return timestamp > lastUpdateTimestamp ? (timestamp - lastUpdateTimestamp) * rewardPerSecond : 0;</pre>
Recommendation	Consider implementing the gas optimizations mentioned above.
Resolution	 RESOLVED

Issue #30	Typographical issues
Severity	 INFORMATIONAL
Description	<p><u>L4, 15 and 19</u></p> <pre>import {Math} from "./Math.sol"; using Math for uint256; uint256 totalDeposit;</pre> <p>These imports and definitions are unused and can be deleted.</p>
Recommendation	Consider fixing the typographical issues.
Resolution	 RESOLVED

2.19 JoeStaking



JoeStaking facilitates the staking of JOE tokens and receiving rewards in JOE or other tokens. The contract owner can set the address of the corresponding JoeStakingRewarder which handles the rewards accounting and claim logic.

2.19.1 Privileged Functions

- `transferOwnership`
- `renounceOwnership`
- `acceptOwnership`
- `setRewarder`



2.19.2 Issues & Recommendations

Issue #31	The owner should not be able to set a previously-used rewarder as the current one
Severity	 INFORMATIONAL
Description	<p>JoeStaking interacts with JoeStakingRewarder which has been deployed by the RewarderFactory. This external contract handles the rewards accounting and claiming logic. The owner has the ability to change the address of the rewarder contract used in case, for example, they want to reward stakers with a different token.</p> <p>However, a potential misconfiguration problem may occur as the owner can set a previously used rewarder contract as the current one after a second one has been used, which would break the rewards accounting and therefore distribute rewards incorrectly due to the old state/storage still being in place in the old rewarder contract instead of starting with a new clear one.</p>
Recommendation	Consider ensuring that a rewarder that has already been used by the JoeStaking contract cannot be added a second time.
Resolution	 ACKNOWLEDGED
	The client is aware of the risk and will not set the same rewarder twice.

Severity

 INFORMATIONAL

Description

If no rewarder contract has been set within `pendingRewards`, the returned value for `rewardToken` is `address(0)` which may be misinterpreted as the native token for the chain since this is the same address used for this purpose.

—

The `_modify` function does not completely follow the Checks-Effects-Interactions pattern since the `PositionModified` event is emitted after the external call to the rewarder contract is made which on its side makes an external call to the untrusted `msg.sender` upon `claim`. Consider emitting the event before the `rewarder.onModify` call.

—

An incorrect comment has been copied from the `MoeStaking` contract on Line 148:

** Will update the veJOE and sJOE positions of the account.*

Recommendation

Consider fixing the typographical issues.

Resolution

 PARTIALLY RESOLVED

The last issue was resolved.



2.20 JoeStakingRewarder

JoeStakingRewarder is the recommended implementation for JoeStaking rewarders. It extends the BaseRewarder contract and simply adds a functionality to query the total supply correctly from the JoeStaking contract.

2.20.1 Privileged Functions

- transferOwnership
- renounceOwnership
- acceptOwnership
- setRewarderParameters
- setRewardPerSecond
- sweep

2.20.2 Issues & Recommendations

Issue #33	Typographical issue
Severity	 INFORMATIONAL
Description	<u>L4</u> <code>import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";</code> This import seems redundant, consider removing it.
Recommendation	Consider fixing the typographical issue.
Resolution	 RESOLVED

2.21 RewarderFactory

RewarderFactory is an upgradeable contract that allows users to create VeMoeRewarder contracts (bribes) and the admin to create MasterChefRewarder and JoeStakingRewarder contracts.

Upon initialization of the contract, an implementation address is set for each of the three types of rewarders. The contract implements various getter functions that can be used by external contracts such as the VeMoe, MasterChef and JoeStaking to verify that the rewarders set and used there have been deployed by the RewarderFactory and therefore the implementation is trusted. The new rewarders are deployed deterministically as immutable minimal proxies.

An important note about this design is that the owner of the contract can change the implementation address of a rewarder type to any arbitrary address at any time and all proxies deployed in the future will use it instead and therefore pass the validation checks.

2.21.1 Privileged Functions

- `transferOwnership`
- `renounceOwnership`
- `acceptOwnership`
- `createRewarder`
- `setRewarderImplementation`

2.21.2 Issues & Recommendations

Issue #34	Blockchain reorg can be abused by a malicious actor to steal funds from creators of VeMoeRewarder contracts
------------------	--

Severity

 MEDIUM SEVERITY

Description

The RewarderFactory deterministically deploys immutable minimal proxies. For calculating the salt used by the create2 opcode when deploying the new contracts, the rewarderType and rewarders.length variables are encoded and hashed:

```
bytes memory immutableData = abi.encodePacked(token, pid);
bytes32 salt =
keccak256(abi.encodePacked(uint8(rewarderType),
rewarders.length));
rewarder =
IBaseRewarder(ImmutableClone.cloneDeterministic(address(impl
ementation), immutableData, salt));
rewarders.push(rewarder);
_rewarderTypes[rewarder] = rewarderType;
rewarder.initialize(msg.sender);
```

A well-known attack vector is block reorg attacks where adversaries make use of blockchains reordering blocks and transactions.

In the case of the RewarderFactory, consider the following scenario:

1. Alice submits a transaction to create a VeMoeRewarder contract with token X and pool ID Y.
2. Alice sends \$100,000 of value of token X to fund the newly created rewarder.
3. Bob notices that a blockchain reorg is happening right while Alice executes these actions so he also calls the RewarderFactory.createRewarder function.
4. Blocks and transactions get reordered due to the blockchain reorg, so now Bob's transaction happened before Alice's first transaction.

-
5. Since Bob is now deployed at the same address that Alice intended to (because all parameters used to calculate the salt, `rewarderType` and `rewarders.length`, were the same), Alice's second transaction will fund Bob's rewarder instead of her own. The difference is that Bob's address was used as the `msg.sender` and owner when initializing the contract on Line 136 instead of Alice's, and that it was not included in the parameters used to calculate the salt hash. Therefore, the address remains the same.
-


Recommendation Consider including all parameters in the calculation of the salt hash that are used to configure the VeMoeRewarder proxy upon deployment and initialization. These include the token address, `pid`, `rewarderType`, and most importantly `msg.sender`.

Note that the salt should also **NOT** be dependent on the length of the `rewarders` array as the attacker will be able to simply execute the same attack making the victim's rewarder address impossible to create as the length has changed.

Resolution



The salt hash now depends on the `msg.sender` address (bribe creator) and a corresponding nonce number which increments with each rewarder deployment.


Issue #35**Deployers of VeMoeRewarder contracts can abuse privileged functions to grief users****Severity** INFORMATIONAL**Description**

The factory allows any account to create VeMoeRewarder contracts with custom parameters such as the pool ID and reward token.

VeMoeRewarder comes with a set of privileged functionalities such as `setRewarderParameters`, `setRewardPerSecond`, `stop` and `sweep`. Those can be abused by a malicious actor by, for example, advertising a generous bribe, and then stopping and sweeping any funds before users harvest their rewards.

Recommendation

Given that the VeMoe contract implements the `emergencyUnsetBribes` function, any potential griefing attack can easily be mitigated by simply removing the malicious bribe. Users should be very careful when deciding what bribes to use as anyone can create one with arbitrary parameters such as the reward token even though the implementation will be a known one.

Resolution ACKNOWLEDGED

This design is intended.



Severity

 INFORMATIONAL

Description

There is no validation that the pool ID pid exists within the Moe MasterChef contract. Consider implementing a check that the passed pid is less than the MasterChef _farms array's length.

—

The pool ID pid must have a value of 0 when creating a JoeStakingRewarder contract. However, such check is missing in the _clone method.

—

The comment on Line 14 is not completely accurate as the admin can now also create JoeStakingRewarder contracts.

Recommendation

Consider fixing the typographical issues.

Resolution

 RESOLVED

The second and third recommendations were implemented. The first one was kept as-is in case the owner wants to create a rewarder before creating the farm and add the rewarder at the same time.

2.22 VestingContract

VestingContract is a simple vesting contract where the owner of the Moe MasterChef contract can set the beneficiary of the funds who is the only account that can release the vested tokens.

The owner has the ability to revoke the vesting at any time, which will result in any funds still unvested being returned back to him and allow the beneficiary to withdraw the tokens that are already vested. The beneficiary will also be able to directly withdraw any funds deposited to the contract in the future.

An important note about this design is that the MasterChef contract owner can basically withdraw all funds at any time by setting the beneficiary to his own address and then calling the revoke and release functions.

2.22.1 Privileged Functions

- release
- setBeneficiary
- revoke



2.22.2 Issues & Recommendations

Issue #37	The owner will receive less funds than intended when vesting is revoked
-----------	---

Severity

 MEDIUM SEVERITY

Description

When the MasterChef contract owner decides they want to cancel the vesting, they can call the `revoke()` function which will calculate the amount that has been vested already and send the remainder back to the owner:

```
function revoke() public virtual {
    [...]
    uint256 balance = _token.balanceOf(address(this));
    uint256 vested = _vestingSchedule(balance + released(),
    block.timestamp);

    _revoked = true;

    _token.safeTransfer(owner, balance - vested);
    [...]
}

function _vestingSchedule(uint256 total, uint256 timestamp)
internal view virtual returns (uint256) {
    if (_revoked) return total;

    if (timestamp < start()) {
        return 0;
    } else if (timestamp >= end()) {
        return total;
    } else {
        return (total * (timestamp - start())) / duration();
    }
}
```

An error has been made when determining the amount of tokens that has to be sent to the owner. The local variable `vested` represents the amount of tokens that have been already vested and released by the beneficiary address as well as the already vested, but not yet released (claimed) tokens. Therefore, the correct formula would be `balance + released() - vested` instead of only `balance - vested` done on Line 163.

The impact is that fewer tokens will actually be returned to the MasterChef contract owner, and therefore more tokens will be reserved for the beneficiary to release.

Additionally, after some point, the `revoke()` function will become inaccessible since the subtraction `balance - vested` will begin reverting due to an underflow exception being thrown.

Recommendation Consider implementing the following changed on Line 163:
`_token.safeTransfer(owner, balance + released() - vested);`

Resolution



The recommendation was implemented.



Issue #38**The MasterChef contract owner can be renounced, potentially resulting in locked funds****Severity** INFORMATIONAL**Description**

The Moe MasterChef contract inherits OpenZeppelin's OwnableUpgradeable contract which includes the `renounceOwnership` method. If executed, the `revoke()` function of the `VestingContract` will become inaccessible since there will not exist any valid caller:

```
address owner = Ownable(_masterChef).owner();
if (msg.sender != owner) revert
VestingContract__NotMasterChefOwner();
```

Additionally, the `_beneficiary` address is not set upon initialization of the contract and therefore remains equal to `address(0)` unless the owner updates it. In the case that funds are sent to the `VestingContract` and the MasterChef contract owner calls `renounceOwnership` before setting the `_beneficiary` address, any funds deposited to the contract will be lost/locked. However, the likelihood of this scenario happening is very low.

Recommendation

Consider initializing the `_beneficiary` address variable in `VestingContract.initialize` as well as implementing a zero-address check in the `setBeneficiary` function.

Resolution RESOLVED

`renounceOwnership` was disabled in all relevant contracts.

Severity

 INFORMATIONAL

Description

The contract uses the variable `_released` to track the amount of tokens that have been vested and then released (claimed) by the `_beneficiary` address.

The type of this variable is `uint88` which has been chosen so that the 3 storage variables used by the contract could fit in 1 single storage slot (a well-known gas optimization). However, this imposes the risk of an arithmetic overflow exception being thrown if the amount of tokens released exceeds `type(uint88).max` which is equivalent to roughly 300 million tokens with 18 decimals.

So, based on the decimals and value of the token that will be used for the vesting contract, this amount can either be sufficiently high or too low.

There is also a way in which a malicious `_beneficiary` address may manipulate the value of the `_released` variable to reach its upper limit more quickly by simply releasing (claiming) tokens and then transferring them back directly to the contract so that they can release them again and that way count/add them twice towards the value of the `_released` variable. However, such manipulation will not result in any profit for the attacker and can also be executed only with a low value token or one with a high number of decimals.

Recommendation

Consider whether this could become a risk. If so, consider using `uint256` as the type of the variable `_released`.

Resolution

 RESOLVED

The recommendation was implemented.

Description

Some of the functions currently marked as public can be changed to external as they are not called internally within the contract:

- `token()`
- `masterChef()`
- `beneficiary()`
- `revoked()`
- `setBeneficiary()`
- `release()`
- `revoke()`

—

The comparison operator `<=` can be used instead of `<` on Line 177 to save gas from skipping the execution of the vested amount calculation.

—

Within `release()`, `msg.sender` can be used instead of the storage variable `_beneficiary` when transferring the token amount and emitting the event in order to save gas.

—

The following zero-value checks can be added to prevent from unexpected values used during functions execution:

- `require(duration_ != 0)` in the contract's constructor
- `require(amount != 0)` in `release()`
- `require(balance + released() - vested != 0)` in `revoke()`

revoke() can currently be called more than once which results in emitting an unnecessary event and executing a 0 value transfer. Consider allowing the owner to call revoke() only once while the _revoked variable is still set to false.

Recommendation Consider fixing the typographical issues.

Resolution



Most of the issues were resolved.





PALADIN
BLOCKCHAIN SECURITY