Muse Smart Contract Preliminary Audit Report

Project Synopsis

Project Name	**Muse Staking**
Platform	Ethereum, Solidity
Github Repo	
Deployed Contract	Not Deployed
Total Duration	7 Days
Timeline of Audit	**8th April 2021 to 14th April 2021**

Contract Details

Total Contract(s)	5
Name of Contract(s)	Masterchef.sol, CakeToken.sol, MuseToken.sol, SyrupBar.sol, TimeLock.sol
Language	Solidity
Commit Hash	

Contract Vulnerabilities Synopsis

Issues	Open Issues	Closed Issues
Critical Severity	1	0
Medium Severity	6	0
Low Severity	8	0
Information	5	0
Total Found	20	0

Detailed Results

The contract has gone through several stages of the audit procedure that includes structural analysis, automated testing, manual code review etc.

All the issues have been explained and discussed in detail below. Along with the explanation of the issue found during the audit, the recommended way to overcome the issue or improve the code quality has also been mentioned.

A. Contract Name: MasterChef

High Severity Issues

A.1 No significant use of the updateStakingPool function was found. Function appears to be incomplete.

Line no: 209-220

Explanation:

The **Masterchef** protocol has a ***updateStakingPool*** function that appears to be incomplete.

The function, as of now, simply iterates over the **poolInfo array** and updates a local variable, ***points,*** which is not used further in the function.

While this function is also being called in other functions like **add** and **set,** it might not achieve the intended behavior.

```
209     function updateStakingPool() internal {
210         uint256 length = poolInfo.length;
211         uint256 points = 0;
212         for (uint256 pid = 1; pid < length; ++pid) {
213             points = points.add(poolInfo[pid].allocPoint);
214         }
215         // if (points != 0) {
216         //     points = points.div(3);
217         //     totalAllocPoint = totalAllocPoint.sub(poolInfo[0].allocPoint).add(points);
218         //     poolInfo[0].allocPoint = points;
219         // }
220     }
```

Recommendation:

It is recommended to modify the updateStakingPool function accordingly. However, If the **updateStakingPool** function is not supposed to be used adequately in the protocol, it should be removed.

Medium Severity Issues

A.2 Similar LP Token Address can be added more than once.

Line no - 154 to 178

Explanation:

As per the current architecture of the MasterChef contract, the **LP Tokens** added in the pool of this contract should not be repeated. Since the address of an **LP Token** plays an imperative role in the calculation of rewards as well as keeping track of specific LP supply in the pool, the presence of a similar LP Token address more than once will break some of the core functionalities of the contract.

However, the **add()** function at Line 154 allows storing a similar LP Token Address more than once. This will lead to an unexpected scenario where different pools will have a similar LP token address.

```
154     function add(
155         uint256 _allocPoint,
156         IBEP20 _lpToken,
157         bool _withUpdate
158     ) public onlyOwner {
159         bool iswrap;
160 
161         if (
162             IDefiWrap(wrappedContract).erc20ImplementationOf(
163                 address(_lpToken)
164             ) == address(0)
165         ) {
166             iswrap = true;
167         }
168 
169         if (_withUpdate) {
170             massUpdatePools();
171         }
172         uint256 lastRewardBlock = block.number > startBlock
173             ? block.number
174             : startBlock;
175         totalAllocPoint = totalAllocPoint.add(_allocPoint);
176         poolInfo.push(
177             PoolInfo({
178                 lpToken: _lpToken,
```

Recommendation:
If the above-mentioned issue was not considered during the function design, then argument ***_lpToken*** (*LP token address*) passed in the **add()** function must be checked at the very beginning of the function with a **require statement.**

The **require statement** could be designed to check whether or not the passed lpToken address is already present in the contract. Moreover, the **add()** function should only execute if a new lpToken address is passed, thus eliminating any chances of repeating a similar lpToken in more than one pool.
Although the function has been assigned an onlyOwner modifier, including the above-mentioned require statement will ensure an adequate function design.

A.3 Strict Equality should be avoided in Require Statements

Line no - 235

Explanation:
The **require** statement in the ***migrate*** function of the protocol includes a strict equality check that ensures that the **balance** of the previous *LP Token* is exactly similar to the new one.

```
227     // Migrate lp token to another lp contract. Can be called by anyone. We
228     function migrate(uint256 _pid) public {
229         require(address(migrator) != address(0), "migrate: no migrator");
230         PoolInfo storage pool = poolInfo[_pid];
231         IBEP20 lpToken = pool.lpToken;
232         uint256 bal = lpToken.balanceOf(address(this));
233         lpToken.safeApprove(address(migrator), bal);
234         IBEP20 newLpToken = migrator.migrate(lpToken);
235         require(bal == newLpToken.balanceOf(address(this)), "migrate: bad");
236         pool.lpToken = newLpToken;
237     }
```

It is considered a better practise in Solidity to avoid strict equality check on ether or token balances as any slight difference between the balances might revert the entire function.

Recommendation:
It is recommended to avoid the use of strict equality in **require** statements unless the above-mentioned function design is intentional.

A.4 Contract State Variables are being updated after External Calls. Violation of Check-Effects Interaction Pattern

Line no - 228, 309, 347, 378, 409, 437, 569,

Explanation:

The MasterChef contract includes quite a few functions that update some of the very imperative state variables of the contract after the external calls are being made.

An external call within a function technically shifts the control flow of the contract to another contract for a particular period of time. Therefore, as per the Solidity Guidelines, any modification of the state variables in the base contract must be performed before executing the external call.
Updating state variables after an external call might lead to a potential re-entrancy scenario.

The following functions in the contract updates the state variables after making an external call

- ***deposit()***
- ***withdraw()***
- ***unstakingWrappedToken()***
- ***stakingWrappedToken()***
- ***migrate()***
- ***_claimReward***
- ***emergencyWithdraw() function*** *at Line* ***314-316***

For instance, the **emergencyWithdraw function** makes an external call to the lpToken contract(*Line 573*) to transfer the deposited amount of tokens(***user.amount***) back to the user. However the User struct(***state variable in the contract***) is updated after the external.
This is not considered a secure practice while developing smart contracts in Solidity.

```
569     function emergencyWithdraw(uint256 _pid) public {
570         PoolInfo storage pool = poolInfo[_pid];
571         UserInfo storage user = userInfo[_pid][msg.sender];
572         pool.lpToken.safeTransfer(address(msg.sender), user.amount);
573         emit EmergencyWithdraw(msg.sender, _pid, user.amount);
574         user.amount = 0;
575         user.rewardDebt = 0;
576     }
```

Recommendation:

Modification of any State Variables must be performed before making an external call. Check Effects Interaction Pattern must not be violated.

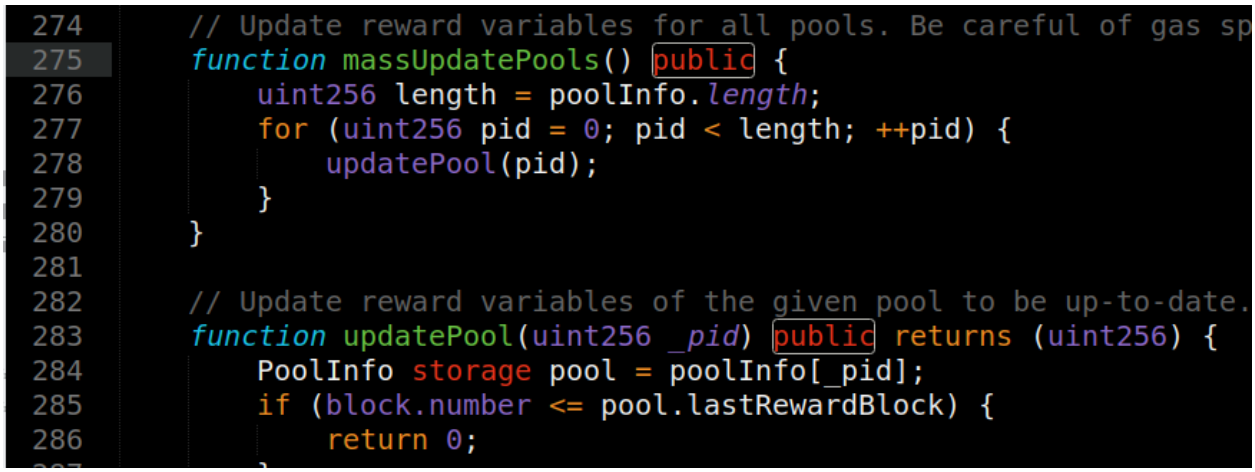
A.5 updatePool and massUpdatePools functions have been assigned a Public visibility

Line no - 275-280, 283-306

Explanation:

The **updatePool** and **massUpdatePools** functions include imperative functionalities as they deal with updating the reward variables of a given pool.

These functions are called within the contract by some crucial functions like **add(), deposit, withdraw** etc.
However, instead of an **internal visibility,** these functions have been assigned a **public** visibility.



```
274     // Update reward variables for all pools. Be careful of gas sp
275     function massUpdatePools() public {
276         uint256 length = poolInfo.length;
277         for (uint256 pid = 0; pid < length; ++pid) {
278             updatePool(pid);
279         }
280     }
281
282     // Update reward variables of the given pool to be up-to-date.
283     function updatePool(uint256 _pid) public returns (uint256) {
284         PoolInfo storage pool = poolInfo[_pid];
285         if (block.number <= pool.lastRewardBlock) {
286             return 0;
```

Is this Intentional?

Since **public** visibility will make the ***updatePool & massUpdatePools function*** accessible to everyone, it would have been a more effective and secure approach to mark these functions as **internal.**

Recommendation:

If both of these functions are only to be called from within the contract, their visibility specifier should be changed from **public** to **internal**.

A.6 Multiplication is being performed on the result of Division

Line no - 263-269, 248-256, *467-473*, 558-563

***Explanation*:**

During the automated testing of the MasterChef.sol contract, it was found that some of the functions in the contract are performing multiplication on the result of a Division.

Integer Divisions in Solidity might truncate. Moreover, this performing division before multiplication might lead to loss of precision.

The following functions involve division before multiplication in the mentioned lines:

- ***pendingMuse*** *at 263-269*
- ***updatePool*** *at 294-302*
- ***_claimReward*** *at 467-473*
- ***getEstimatedReward*** *at 558-563*

```
294         uint256 cakeReward = multiplier
295         .mul(cakePerBlock)
296         .mul(pool.allocPoint)
297         .div(totalAllocPoint);
298         // muse.mint(devaddr, cakeReward.div(10));
299         // muse.mint(address(syrup), cakeReward);
300         pool.accMusePerShare = pool.accMusePerShare.add(
301             cakeReward.mul(1e12).div(lpSupply)
302         );
```

Recommendation:

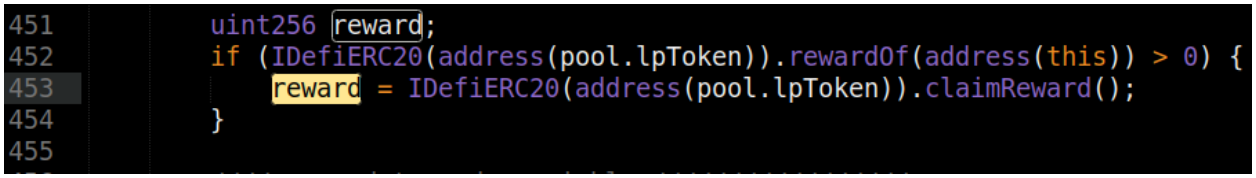
Solidity doesn't encourage arithmetic operations that involve division before multiplication. Therefore the above-mentioned function should be checked once and redesigned if they do not lead to expected results.

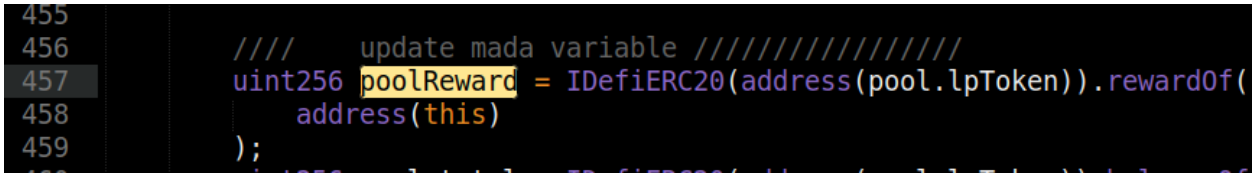
A.7 Local variables Not used effectively

Line no - 451-454, 415, 457, 551

Explanation:

The Masterchef contract includes some functions that include local variables with no significant use.





The above-mentioned line numbers symbolizes the exact lines where such variables can be found.

Recommendation:

If a particular local variable doesn't have a significant use case within the function, it is considered to remove such variables to enhance the function design and code readability.

Low Severity Issues

A.8 Return Value of an External Call is never used Effectively

Line no - 327, 363, 396, 506

Explanation:

The external calls made in the above-mentioned lines do return a boolean value that indicates whether or not the external call made was successful.

These boolean return values can be used in the function as a check to ensure that the further execution of the function is only allowed if the external is successfully made.

However, the MasterChef contract never uses these return values throughout the contract.

Recommendation:

Effective use of all the return values from external calls must be ensured within the contract.

A.9 External Visibility should be preferred

Explanation:

Those functions that are never called throughout the contract should be marked as ***external*** visibility instead of ***public*** visibility.

This will effectively result in Gas Optimization as well.

Therefore, the following function must be marked as **external** within the contract:

- ***updateMultiplier***
- ***add***
- ***set***
- ***setMigrator***
- ***migrate***
- ***deposit***
- ***withdraw***
- ***stakingWrappedToken***
- ***unstakingWrappedToken***
- ***emergencyWithdraw***

Recommendation:
If the PUBLIC visibility of the the above-mentioned is not intended, then EXTERNAL visibility should be preferred.

A.10 Comparison to boolean Constant

Line no: 487

Description:
Boolean constants can directly be used in conditional statements or require statements.
Therefore, it's not considered a better practice to explicitly use **TRUE or FALSE** in the **require** statements.

```
486             );
487             require(xfer == true, "ERR_TRANSFER");
488         }
489
```

Recommendation:
The equality to boolean constants must be removed from the above-mentioned line.

A.11 No Events emitted after imperative State Variable modification Line no -145

Explanation:
Functions that update an imperative arithmetic state variable contract should emit an event after the updation.
The **updateMultiplier** function modifies a crucial state variable, i.e, **BONUS_MULTIPLIER** in the Masterchef contract but doesn't emit any event after that.

```
144     function updateMultiplier(uint256 multiplierNumber) public onlyOwner {
145         BONUS_MULTIPLIER = multiplierNumber;
146     }
```

Since there is no event emitted on updating these variables, it might be difficult to track it off chain.

***Recommendation*:**
An event should be fired after changing crucial arithmetic state variables.

Informational

A.12 Coding Style Issues in the Contract

Explanation:

Code readability of a Smart Contract is largely influenced by the Coding Style issues and in some specific scenarios may lead to bugs in the future.

```
Parameter MasterChef.add(uint256,IBEP20,bool)._lpToken (flatContracts/FlatMasterChef.sol#1622) is not in mixedCase
Parameter MasterChef.add(uint256,IBEP20,bool)._withUpdate (flatContracts/FlatMasterChef.sol#1623) is not in mixedCase
Parameter MasterChef.set(uint256,uint256,bool)._pid (flatContracts/FlatMasterChef.sol#1658) is not in mixedCase
Parameter MasterChef.set(uint256,uint256,bool)._allocPoint (flatContracts/FlatMasterChef.sol#1659) is not in mixedCase
Parameter MasterChef.set(uint256,uint256,bool)._withUpdate (flatContracts/FlatMasterChef.sol#1660) is not in mixedCase
Parameter MasterChef.setMigrator(IMigratorChef)._migrator (flatContracts/FlatMasterChef.sol#1689) is not in mixedCase
Parameter MasterChef.migrate(uint256)._pid (flatContracts/FlatMasterChef.sol#1694) is not in mixedCase
Parameter MasterChef.getMultiplier(uint256,uint256)._from (flatContracts/FlatMasterChef.sol#1706) is not in mixedCase
Parameter MasterChef.getMultiplier(uint256,uint256)._to (flatContracts/FlatMasterChef.sol#1706) is not in mixedCase
Parameter MasterChef.pendingMuse(uint256,address)._pid (flatContracts/FlatMasterChef.sol#1715) is not in mixedCase
Parameter MasterChef.pendingMuse(uint256,address)._user (flatContracts/FlatMasterChef.sol#1715) is not in mixedCase
Parameter MasterChef.updatePool(uint256)._pid (flatContracts/FlatMasterChef.sol#1749) is not in mixedCase
```

During the automated testing, it was found that the **Masterchef** contract had quite a few code style issues.

Recommendation:

Therefore, it is recommended to fix the issues like naming convention, indentation, and code layout issues in a smart contract.

A.13 NatSpec Annotations must be included

Description:

Smart contract does not include the NatSpec annotations adequately.

Recommendation:

Cover by NatSpec all Contract methods.

A.14 Commented codes must be wiped-out before deployment

Explanation

The Masterchef contract includes quite a few commented codes in the protocol. This badly affects the readability of the code.

Recommendation:

If the commented instances of code are not required in the current version of the contract, then those codes must be removed before deployment.

B. Contract Name: Timelock

Low Severity Issues

B.1Absence of Zero Address Validation

Line no- 43, 76

Description:

The **Timelock** Contract includes some functions that update some of the imperative addresses in the contract like ***admin, pendingAdmin etc***.

However, during the automated testing of the contact it was found that no *Zero Address Validation* is implemented on the following functions while updating the above-mentioned addresses of the contract:

- ***constructor***
- ***setPendingAdmin***

Recommendation:

A ***require*** statement should be included in such functions to ensure no zero address is passed as arguments.

B.2 Redundant State Variable Update

Line no: 45

Explanation

The **Timelock** Smart contract involves redundant updating of its State Variable, i.e., ***admin_initialized.***

```
39      constructor(address admin_, uint
40          require(delay_ >= MINIMUM_DEL
41          require(delay_ <= MAXIMUM_DEL
42
43          admin = admin_;
44          delay = delay_;
45          admin_initialized = false;
46      }
```

A boolean variable is, by-default initialized to FALSE. Hence, such state variables do not need to be initialized explicitly.

Recommendation:

Redundant initialization of state variables should be avoided.

C. Contract Name: SyrupBar

Low Severity Issues

D.1 Return Value of an External Call is never used Effectively

Line no - 34, 35

Explanation:

The external calls made in the above-mentioned lines do return a boolean value that indicates whether or not the external call made was successful.

These boolean return values can be used in the function as a check to ensure that the further execution of the function is only allowed if the external is successfully made.

```
31      function safeMuseTransfer(address _to, uint256 _amount)
32          uint256 cakeBal = muse.balanceOf(address(this));
33          if (_amount > cakeBal) {
34              muse.transfer(_to, cakeBal);
35          } else {
36              muse.transfer(_to, _amount);
37          }
38      }
```

However, the **SyrupBar** contract never uses these return values throughout the contract.

Recommendation:

Effective use of all the return values from external calls must be ensured within the contract.

Informational

D.2 Coding Style Issues in the Contract

Explanation:

Code readability of a Smart Contract is largely influenced by the Coding Style issues and in some specific scenarios may lead to bugs in the future.

```
Parameter SyrupBar.mint(address,uint256)._to (flatContracts/FlatSyrupbar.sol#1114) is not in mixedCase
Parameter SyrupBar.mint(address,uint256)._amount (flatContracts/FlatSyrupbar.sol#1114) is not in mixedCase
Parameter SyrupBar.burn(address,uint256)._from (flatContracts/FlatSyrupbar.sol#1119) is not in mixedCase
Parameter SyrupBar.burn(address,uint256)._amount (flatContracts/FlatSyrupbar.sol#1119) is not in mixedCase
Parameter SyrupBar.safeMuseTransfer(address,uint256)._to (flatContracts/FlatSyrupbar.sol#1135) is not in mixedCase
Parameter SyrupBar.safeMuseTransfer(address,uint256)._amount (flatContracts/FlatSyrupbar.sol#1135) is not in mixedCase
Variable SyrupBar._delegates (flatContracts/FlatSyrupbar.sol#1151) is not in mixedCase
```

During the automated testing, it was found that the **SyrupBar** contract had quite a few code style issues.

Recommendation:

Therefore, it is recommended to fix the issues like naming convention, indentation, and code layout issues in a smart contract.

D. Contract Name: Muse Token

Low Severity Issues

E.1 Invalid Error Messages in the Require Statements

Explanation:

The error messages in the Must Token contract includes wrong statements as it shows **CAKE** instead of **MUSE.**

```
115         require(signatory != address(0), "CAKE::delegateBySig: invalid signature");
116         require(nonce == nonces[signatory]++, "CAKE::delegateBySig: invalid nonce");
117         require(now <= expiry, "CAKE::delegateBySig: signature expired");
118         return _delegate(signatory, delegatee);
```

Recommendation:

It is recommended to update the error messages of the **require** statements in the Muse Token contract and include right statements to avoid any confusions.

Informational

E.2 Coding Style Issues in the Contract

Explanation:

Code readability of a Smart Contract is largely influenced by the Coding Style issues and in some specific scenarios may lead to bugs in the future.

```
Parameter MuseToken.mint(address,uint256)._to (flatContracts/FlatMuse.sol#873) is not in mixedCase
Parameter MuseToken.mint(address,uint256)._amount (flatContracts/FlatMuse.sol#873) is not in mixedCase
Variable MuseToken._delegates (flatContracts/FlatMuse.sol#885) is not in mixedCase
```

During the automated testing, it was found that the **Muse Token** contract had quite a few code style issues.

Recommendation:

Therefore, it is recommended to fix the issues like naming convention, indentation, and code layout issues in a smart contract.

E. Contract Name: Cake Token

No significant Issues Found

Automated Test Results