PALADIN
BLOCKCHAIN SECURITY

Smart Contract Security Assessment

Final Report

For Farmers Only (Layer 2)

17 October 2021

paladinsec.co

info@paladinsec.co

Table of Contents

Table of Contents 2

Disclaimer 3

1 Overview 4

- 1.1 Summary 4
- 1.2 Contracts Assessed 4
- 1.3 Findings Summary 5
  - 1.3.1 FarmersOnlyMasterChefL2 6
  - 1.3.2 TomatoCoin 7
  - 1.3.3 StakingPool 7

2 Findings 8

- 2.1 FarmersOnlyMasterChefL2 8
  - 2.1.1 Privileged Roles 8
  - 2.1.2 Issues & Recommendations 9
- 2.2 TomatoCoin 20
  - 2.2.1 Token Overview 20
  - 2.2.2 Privileged Roles 20
  - 2.2.3 Issues & Recommendations 21
- 2.3 StakingPool 23
  - 2.3.1 Privileged Roles 23
  - 2.3.2 Issues & Recommendations 24

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 advocation 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.

1 Overview

This report has been prepared for FarmersOnly's Layer 2 farms on the Avalanche 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	Farmers Only
URL	https://farmersonly.farm
Platform	Avalanche
Language	Solidity

1.2 Contracts Assessed

Name	Contract	Live Code Match
MasterChef	FarmersOnlyMasterChefL2.sol	
TomatoCoin	TomatoCoin.sol	
StakingPool	StakingPool.sol	
Source	https://github.com/farmersonlydefi/farmers-contracts/tree/main/Layer%202%20-%20TOMATO	

1.3 Findings Summary

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
🔴 High	2	2	-	-
🟠 Medium	2	2	-	-
🟡 Low	8	7	-	1
🟣 Informational	18	16	-	2
Total	**30**	**27**	**-**	**3**

Classification of Issues

Severity	Description
🔴 **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 with 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 FarmersOnlyMasterChefL2

ID	Severity	Summary	Status
01	HIGH	The reward distribution mechanism is wrongly defined	✓ RESOLVED
02	MEDIUM	Maximum supply cap logic is wrongly defined, causing the token to mint either too much or too little potentially causing deposits and withdrawals to revert	✓ RESOLVED
03	LOW	Setting devAddress and feeAddress to the zero address will break updatePool and deposit functionality respectively	✓ RESOLVED
04	LOW	Adding an EOA or a non-token contract as a pool will break `updatePool`, `massUpdatePools` and `updateEmissionRate`	✓ RESOLVED
05	LOW	Contract contains unused referral code mechanism which could be accidentally set to block deposits	✓ RESOLVED
06	LOW	The `pendingTomato` function will revert if `totalAllocPoint` is zero	✓ RESOLVED
07	INFO	Pools use the contract balance to figure out the total deposits	ACKNOWLEDGED
08	INFO	There are no sanity checks in the `setReferralAddress` function	✓ RESOLVED
09	INFO	`deposit`, `withdraw`, `set`, `emergencyWithdraw`, `setStartTime`, `updateEmissionRate`, `setFeeAddress`, dev and `getHarvestUntil` functions can be made `external`	✓ RESOLVED
10	INFO	`MAX_EMISSION_RATE` and `EMISSION_RATE` can be declared as a constant	✓ RESOLVED
11	INFO	`tomato` can be declared as immutable	✓ RESOLVED
12	INFO	Contracts contains multiple occurrences of code that has been commented out	✓ RESOLVED
13	INFO	The `BONUS_MULTIPLIER` variable is not functional	✓ RESOLVED
14	INFO	The `StartTimeChanged` event wrongly emitted	✓ RESOLVED
15	INFO	The set function error still references the add function	✓ RESOLVED
16	INFO	`msg.sender` is unnecessarily cast to `address(msg.sender)`	✓ RESOLVED
17	INFO	The `rand()` and `isContract()` functions are not used	✓ RESOLVED

1.3.2 TomatoCoin

ID	Severity	Summary	Status
18	LOW	`mint` function can be used to pre-mint large amounts of tokens before ownership is transferred to the Masterchef	ACKNOWLEDGED
19	INFO	`_totalSupply` is sufficient to keep track of the total token supply minted	ACKNOWLEDGED

1.3.3 StakingPool

ID	Severity	Summary	Status
20	HIGH	If the staked token is equal to the reward token, the owner can drain the staked tokens with `emergencyRewardWithdraw`	✓ RESOLVED
21	MEDIUM	The `deposit`, `withdraw` and `emergencyWithdraw` functions do not have reentrancy guards	✓ RESOLVED
22	LOW	The `setRewardPerTime` function can be set to an excessive emission rate	✓ RESOLVED
23	LOW	The `updatePool` function will revert if `totalAllocPoint` is zero	✓ RESOLVED
24	LOW	The `setRewardPerTime` should include an `updatePool(0);` before the change	✓ RESOLVED
25	INFO	The `deposit` and `withdraw` functions can be made external	✓ RESOLVED
26	INFO	`stakeToken`, `rewardToken` and `startTime` can be declared as a `immutable`	✓ RESOLVED
27	INFO	The `poolInfo` is unnecessarily declared as an array making `totalAllocPoint` redundant as well	✓ RESOLVED
28	INFO	The `msg.sender` is unnecessarily cast to `address(msg.sender)`	✓ RESOLVED
29	INFO	Lack of events for `setRewardPerTime` and `setBonusEndTime`	✓ RESOLVED
30	INFO	Typographical errors	✓ RESOLVED

2 Findings

2.1 FarmersOnlyMasterChefL2

FarmersOnlyMasterChefL2 is a modified Panther Masterchef that gives rewards based on timestamp instead of blocks. The deposit fees are capped at 4%.

This Masterchef also contains a lockup mechanism that does a harvests half the rewards while and the other half stays locked up. This is different from the original Panther Masterchef where the rewards can be harvested completely after the lockup expires. The maximum harvest interval can be set to 2 days.

The initial emission rate is set to 0.0035 Tomato tokens per second and can be increased to at most 0.007 Tomato tokens per second.

2.1.1 Privileged Roles

The following functions can be called by the owner of the contract:

- `set`
- `setStartTime`
- `addPool`
- `setReferralAddress`
- `setReferralCommissionRate`
- `dev`
- `setFeeAddress`
- `renounceOwnership`
- `transferOwnership`

2.1.2 Issues & Recommendations

Issue #01	The reward distribution mechanism is wrongly defined
Severity	● HIGH SEVERITY
Location	Line 1731 `uint256 totalRewards = pending.add(user.rewardLockedUp.div(2));`
Description	The client has indicated that when rewards are unlocked, they want to lock 50% of the harvestable tokens. However, if users do not lock up any tokens, their full harvest will be encompassed within the `pending` variable and not the `user.rewardLockedUp` variable. This latter variable is relocked partially in the current code, but `pending` is currently distributed to the user completely. Therefore we expect users to completely circumvent this tokenomic design by waiting to harvest only after their lockup timer expires.

Recommendation	Consider adjusting the functionality to also relock the pending rewards for half of their value.
	`uint256 totalRewards = pending.add(user.rewardLockedUp);`
	`uint256 rewardsToLockup = totalRewards.div(2);`
	`uint256 rewardsToDistribute =`
	`    totalRewards.sub(rewardsToLockup);`
	`// reset lockup`
	`totalLockedUpRewards = totalLockedUpRewards.sub(`
	`    user.rewardLockedUp).add(rewardsToLockup);`
	`user.rewardLockedUp = rewardsToLockup;`
	`user.nextHarvestUntil = getPoolHarvestInterval(_pid);`
	`// send rewards`
	`safeTomatoTransfer(msg.sender, rewardsToDistribute);`
	`payReferralCommission(msg.sender, rewardsToDistribute);`

Resolution	✓ RESOLVED The recommended code has been implemented.

Issue #02	**Maximum supply cap logic is wrongly defined, causing the token to mint either too much or too little potentially causing deposits and withdrawals to revert**
Severity	● MEDIUM SEVERITY

Location	Line 1139
	`uint256 public constant MAX_SUPPLY = 7777 ether;`
	
	Lines 1275-1278
	`function getMultiplier(uint256 _from, uint256 _to) public view returns (uint256) {`
	`    if (tomato.totalSupply() + totalLockedUpRewards >= MAX_SUPPLY) return 0;`
	`    return _to.sub(_from).mul(BONUS_MULTIPLIER);`
	`}`

Description	The maximum supply cap is currently completely wrongly defined since `MAX_SUPPLY` does not match the actual maximum supply of 7,000 tokens. Furthermore the maximum supply limit is currently included in `getMultiplier`, which means it will only set the supply to zero once the supply is matched or exceeded. The latter can never happen because the token will revert any attempts that exceed the supply so it will only set the multiplier to zero if the supply has been exactly reached which is extremely unlikely since there is no specific logic to adjust the reward rate to this. Finally, `totalLockedUpRewards` is added to the `totalSupply` in the check which causes double counting since these tokens have already been minted and are thus also accounted for in `totalSupply`. This all will likely cause the Masterchef to severely malfunction once the total supply is nearly reached, potentially causing deposits and withdrawals to revert if they try to mint amounts that would exceed the total supply. This last part would require people to use `emergencyWithdraw` until the developer sets the allocation points to zero (as `updateEmissionRate` would revert as well). ❗ The current version of `getMultiplier` also uses raw addition instead of SafeMath. As we recommend removing this section, we've not created an explicit issue for this.

Recommendation	Consider using a normal `getMultiplier` function and instead adjusting the reward rate directly in the `updatePool` function.

getMultiplier function

```
function getMultiplier(uint256 _from, uint256 _to) public view
returns (uint256) {
        return _to.sub(_from);
}
```

updatePool adjustment

```
uint256 tomatoReward = multiplier
    .mul(tomatoPerSecond)
    .mul(pool.allocPoint)
    .div(totalAllocPoint);

if(tomato.totalSupply() >= MAX_SUPPLY) {
    tomatoReward = 0;
}else if (tomato.totalSupply().add(tomatoReward.mul(11).div(10))
>= MAX_SUPPLY) {
    tomatoReward =
(MAX_SUPPLY.sub(tomato.totalSupply()).mul(10).div(11));
}
if(tomatoReward > 0) {
    tomato.mint(devAddress, tomatoReward.div(10));
    tomato.mint(address(this), tomatoReward);

    pool.accTomatoPerShare =
pool.accTomatoPerShare.add(tomatoReward.mul(1e18).div(lpSupply));
}
pool.lastRewardTime = block.timestamp;
```

Resolution	✓ RESOLVED Within the token, the maximum supply has been adjusted to 7,777 to match the maximum supply denoted in the Masterchef. The recommended code has furthermore been implemented.

Issue #03	**Setting `devAddress` and `feeAddress` to the zero address will break `updatePool` and deposit functionality respectively**
Severity	● LOW SEVERITY
Description	The `updatePool` function mints tokens to the `devAddress`, but minting will revert if it is made to the zero address. Transferring tokens to the zero address will revert the transactions as well. Deposits will thus break if the `feeAddress` is ever set to the zero address, and deposit-based harvests will break as well.

Recommendation	To prevent this from ever happening by accident and to limit governance risks, consider adding a requirement like
	`require(_devAddress != address(0), “!nonzero”);`
	to the `dev` function, and
	`require(_feeAddress != address(0), “!nonzero”);`
	to the `setFeeAddress` function.
	❗ It may also be better to rename the dev function as `setDevAddress()`.

Resolution	✓ RESOLVED

Issue #04	Adding an EOA or a non-token contract as a pool will break `updatePool`, `massUpdatePools` and `updateEmissionRate`
Severity	● LOW SEVERITY
Description	updatePool will always call `balanceOf(address(this))` on the token of the pool, and will fail if the token is not an actual token contract address.
Recommendation	Consider simply adding a test line in the add function. If the token does not exist, this will make sure the add function fails. `_lpToken.balanceOf(address(this));`
Resolution	✓ RESOLVED The recommended check has been added.

Issue #05	Contract contains unused referral code mechanism which could be accidentally set to block deposits
Severity	● LOW SEVERITY
Description	The contract contains logic for a referral fee. This fee is however not settable to a value greater than zero thus it is not functional. However, this functionality could block deposits if the referral contract is ever set to a malfunctioning contract.
Recommendation	Consider removing all referral fee logic.
Resolution	✓ RESOLVED All referral related logic has been removed.

Issue #06	The `pendingTomato` function will revert if `totalAllocPoint` is zero
Severity	● LOW SEVERITY
Description	In the `pendingTomato` function, at some point a division is made by the `totalAllocPoint` variable. If all pools have their rewards set to zero, this variable will be zero as well. The requests will then revert with a division by zero error.
Recommendation	Consider only calculating the accumulated rewards since the `lastRewardBlock` if the `totalAllocPoint` variable is greater than zero. This check can simply be added to the existing check that verifies the `block.number` and `lpSupply`, like so: `if (block.timestamp > pool.lastRewardTime && lpSupply != 0 && totalAllocPoint > 0) {`
Resolution	✓ RESOLVED

Issue #07	Pools use the contract balance to figure out the total deposits
Severity	● INFORMATIONAL
Description	As with pretty much all Masterchefs, the total number of tokens in the Masterchef contract is used to determine the total number of deposits. This can cause dilution of rewards when people accidentally send tokens to the masterchef. More severely, because the native token is constantly minted, this will cause severe dilution on the native token pool.
Recommendation	Consider adding an `lpSupply` variable to the `PoolInfo` that keeps track of the total deposits. Each `lpToken.balanceOf(address(this))` query can then be replaced with this `lpSupply` as well.
Resolution	● ACKNOWLEDGED

Issue #08	**There are no sanity checks in the `setReferralAddress` function**
Severity	INFORMATIONAL
Description	A lot of functionality can break if the referral address is updated to a value that is not a referral contract.
Recommendation	Consider removing all referral related functionality since this code is unused.
Resolution	✓ RESOLVED

Issue #09	**`deposit`, `withdraw`, `set`, `emergencyWithdraw`, `setStartTime`, `updateEmissionRate`, `setFeeAddress`, `dev` and `getHarvestUntil` functions can be made `external`**
Severity	INFORMATIONAL
Description	The above functions can be changed from public to external. Apart from being a best practice when the function is not used within the contract, this can lead to a lower gas usage in certain cases.
Recommendation	Consider making these functions external.
Resolution	✓ RESOLVED

Issue #10	**`MAX_EMISSION_RATE` and `EMISSION_RATE` can be declared as a constant**
Severity	● INFORMATIONAL
Location	Variables that do not change throughout the contract can be marked as constant to signal this to third-party reviewers. This also reduces gas usage. ❗ The variable `EMISSION_RATE` might furthermore confuse third-party reviewers as it is in-fact only the initial emission rate. The real emission rate is encoded within the `tomatoPerSecond` variable.
Description	Consider marking the aforementioned variables as constant. Furthermore consider renaming `EMISSION_RATE` to `INITIAL_EMISSION_RATE`.
Recommendation	`MAX_EMISSION_RATE` and `EMISSION_RATE` can be declared as a constant
Resolution	✓ RESOLVED

Issue #11	**`tomato` can be declared as `immutable`**
Severity	● INFORMATIONAL
Description	Variables that are not changed throughout the contract but only set in the constructor can be marked as immutable to signal this to third-party reviewers. This furthermore reduces gas usage.
Recommendation	Consider marking the variable as immutable.
Resolution	✓ RESOLVED

Issue #12	Contracts contains multiple occurrences of code that has been commented out
Severity	● INFORMATIONAL

Description	Throughout the contract, there are multiple code sections which have been commented out. This makes it hard to read and can introduce confusion.
	
	Example Location
	`Lines 1123-1124`
	`// uint256 minHarvestInterval;  // Harvest interval in seconds`
	`// uint256 maxHarvestInterval;  // Harvest interval in seconds`

Recommendation	Consider removing the obsolete sections of code.
Resolution	✓ RESOLVED

Issue #13	The `BONUS_MULTIPLIER` variable is not functional
Severity	● INFORMATIONAL
Description	The `BONUS_MULTIPLIER` variable is set to 1 and can not be modified, making it not functional. This could cause unnecessary confusion for third-party reviewers that try to understand the codebase.
Recommendation	Consider removing it.
Resolution	✓ RESOLVED

Issue #14	The `StartTimeChanged` event wrongly emitted
Severity	INFORMATIONAL
Description	The `StartTimeChanged` event is emitted at the wrong code location causing it to emit multiple times or even never when the start time is updated.
Recommendation	Consider moving it outside the `for` loop.
Resolution	✓ RESOLVED

Issue #15	The `set` function error still references the add function
Severity	INFORMATIONAL
Location	`Line 1261` `require(_harvestInterval <= MAXIMUM_HARVEST_INTERVAL, "add: invalid harvest interval");`
Description	The `set` function still references `add` which is presumably an error made through copy-pasting the above code from the `add` function.
Recommendation	Consider re-writing the error message as follows: "set: invalid harvest interval".
Resolution	✓ RESOLVED

Issue #16	`msg.sender` is unnecessarily cast to `address(msg.sender)`
Severity	● INFORMATIONAL
Location	Line 1353 (example) `pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);`
Description	The `msg.sender` is casted to `address(msg.sender)` throughout the contract when used with `pool.lpToken.safeTransfer()`. This is unnecessary.
Recommendation	Consider replacing it with `msg.sender`.
Resolution	✓ RESOLVED

Issue #17	The `rand()` and `isContract()` functions are not used
Severity	● INFORMATIONAL
Description	The `rand()` and `isContract()` functions are not used anywhere in the contract. This could cause unnecessary confusion for third-party reviewers that try to understand the codebase.
Recommendation	Consider removing them.
Resolution	✓ RESOLVED

2.2 TomatoCoin

The Tomato token is a simple BEP20 token with a maximum supply of 7,777 tokens. It allows for Tomato tokens to be minted when the `mint` function is called by the owner of the contract, which at the time of deployment would be the FarmersOnly team. Users should therefore carefully inspect that this account has been correctly initialized to the Masterchef.

2.2.1 Token Overview

Address	TBC
Token Supply	7,777
Decimal Places	18
Transfer Max Size	None
Transfer Min Size	None
Transfer Fees	None

2.2.2 Privileged Roles

The following functions can be called by the owner of the contract:

- `mint`
- `renounceOwnership`
- `transferOwnership`

2.2.3 Issues & Recommendations

Issue #18	**`mint` function can be used to pre-mint large amounts of tokens before ownership is transferred to the Masterchef**
Severity	● LOW SEVERITY
Description	The `mint` function could be used to pre-mint tokens for legitimate uses including, but not limited to, the injection of initial liquidity, token presale, or airdrops; however, this function may also be used to pre-mint and dump tokens when the token contract has been deployed but before ownership is set to the Masterchef contract. This risk is prevalent amongst less-reputable projects, and any pre-mints can be prominently seen on the Blockchain.
Recommendation	Consider being forthright if this `mint` function has been used by letting your community know how much was minted, where they are currently stored, if a vesting contract was used for token unlocking, and finally the purpose of the mints. Users should also carefully inspect which wallet or contract the owner of the token is.
Resolution	● ACKNOWLEDGED This issue will be marked as resolved once ownership has been transferred.

Issue #19	**`_totalSupply` is sufficient to keep track of the total token supply minted**
Severity	INFORMATIONAL
Description	Both `MAXCAP` and `_totalSupply` are incremented by the amount of tokens minted, and decremented by the amount of tokens burnt. This makes `MAXCAP` redundant.
Recommendation	Consider removing `MAXCAP`, and keeping track of the token supply using `_totalSupply`. If the emissions should stop when the `MAXCAP` is minted (eg. on burns new emissions should not be possible again), consider removing the `MAXCAP` reduction on burn.
Resolution	ACKNOWLEDGED

2.3 StakingPool

The StakingPool is a contract based on the Masterchef but with the distinction that it supports only a single staking pool, and is presumably a way to get dividends by staking the native token. This single pool is defined during the creation of the contract.

The owner has privileges to withdraw all reward tokens at any time through `emergencyRewardWithdraw`. The owner cannot withdraw stake tokens unless they are send to the contract by accident, in this case they can skim the non-staked amount with the `skimStakeTokenFees` function.

2.3.1 Privileged Roles

The following functions can be called by the owner of the contract:

- `setRewardPerTime`
- `setBonusEndTime`
- `skimStakeTokenFees`
- `emergencyRewardWithdraw`

2.3.2 Issues & Recommendations

Issue #20	**If the staked token is equal to the reward token, the owner can drain the staked tokens with `emergencyRewardWithdraw`**
Severity	HIGH SEVERITY
Description	The contract contains functionality to withdraw the reward token from the contract. Although this is not desirable, it could be useful if the contract was deployed wrongly to move the reward tokens to another contract. However, if this contract is used to distribute reward tokens that are the same as the staked tokens, this `emergencyRewardWithdraw` function could be used to withdraw the staked tokens as well, as they would be identical to the rewards tokens.
Recommendation	Consider adding a requirement that the staked token is not equal to the rewards token in the constructor. `require(_stakeToken != _rewardToken, “Does not support same currency pools”);`
Resolution	✓ RESOLVED The recommended safeguard has been added to the constructor.

Issue #21	**The `deposit`, `withdraw` and `emergencyWithdraw` functions do not have reentrancy guards**
Severity	MEDIUM SEVERITY
Description	The above functions do not have reentrancy guards and can cause the drainage of the rewards and staking token if ever an ERC-777 or similar token is added as the staking token. This is because such tokens allows the sender and recipient to execute arbitrary code during the sending and receiving of the token which in this case can be used to receive rewards multiple times since `rewardDebt` does not adhere to checks-effects-interactions and withdraw multiple times in `emergencyWithdraw` because this function does not adhere to the checks-effects-interactions pattern either. ❗ Furthermore, the `EmergencyWithdraw` event has zero as the parameter because the variable is reset.
Recommendation	Consider marking the these functions to implement a `nonReentrant` modifier. Consider updating the event to emit the correct withdrawn amount, this could be done by adhering to checks-effects-interactions.
Resolution	✓ RESOLVED `deposit`, `withdraw` and `emergencyWithdraw` have been protected against reentrancy vectors. Additionally, the `emergencyWithdraw` event has been fixed.

Issue #22	The `setRewardPerTime` function can be set to an excessive emission rate
Severity	● LOW SEVERITY
Description	The `setRewardPerTime` can be set to an excessive emission rate which could allow the contract to emit a very large amount of tokens in a short time, this possibility might be bad for investor confidence.
Recommendation	Consider adding a maximum cap to the above function. `require(_rewardPerTime <= MAXIMUM_REWARD_PER_TIME, “too large”);`
Resolution	✓ RESOLVED A maximum of 0.01 tokens per second has been set.

Issue #23	The `updatePool` function will revert if `totalAllocPoint` is zero
Severity	● LOW SEVERITY
Description	In the `updatePool` function, at some point a division is made by the `totalAllocPoint` variable. If all pools have their rewards set to zero, this variable will be zero as well. The requests will then revert with a division by zero error.
Recommendation	Consider adding an `totalAllocPoint != 0` check in the if statement in `updatePool`.
Resolution	✓ RESOLVED

Issue #24	The setRewardPerTime should include an updatePool(0); before the change
Severity	● LOW SEVERITY
Description	When the owner updates the emission rate, this could also affect the emission rate since the previous update (deposit or withdrawal) has occurred causing it to affect rewards in hindsight.
Recommendation	Consider including an `updatePool(0);` before the change in `setRewardPerTime`.
Resolution	✓ RESOLVED

Issue #25	The deposit and withdraw functions can be made external
Severity	● INFORMATIONAL
Description	The deposit and withdraw functions can be changed from public to external. Apart from being a best practice when the function is not used within the contract, this can lead to a lower gas usage in certain cases.
Recommendation	Consider making these functions external.
Resolution	✓ RESOLVED

Issue #26	`stakeToken`, `rewardToken` and `startTime` can be declared as a immutable
Severity	● INFORMATIONAL
Description	Variables that are not changed throughout the contract but only set in the constructor can be marked as `immutable` to signal this to third-party reviewers. This furthermore reduces gas usage.
Recommendation	Consider marking the above variables as immutable.
Resolution	✓ RESOLVED

Issue #27	The poolInfo is unnecessarily declared as an array making totalAllocPoint redundant as well
Severity	● INFORMATIONAL
Description	The poolInfo is unnecessarily declared as an array since there is only one pool, this makes the whole allocPoint logic unnecessary as well since the one pool will always take all emissions. ❗ In addition, the `massUpdatePools` function is never used
Recommendation	Consider changing the `poolInfo` to `PoolInfo poolInfo;` and removing all allocation point related logic.
Resolution	✓ RESOLVED

Issue #28	The msg.sender is unnecessarily cast to address(msg.sender)
Severity	● INFORMATIONAL
Location	`Line 888 (example)` `safeTransferReward(address(msg.sender), currentRewardBalance);`
Description	The `msg.sender` is cast to `address(msg.sender)` throughout the contract when used with safeTransferReward(). This is unnecessary.
Recommendation	Consider replacing it with simply `msg.sender`.
Resolution	✓ RESOLVED

Issue #29	Lack of events for setRewardPerTime and setBonusEndTime
Severity	● INFORMATIONAL
Description	Important functions should emit events to keep a track record of when and how they have been called.
Recommendation	Consider adding events to the above functions.
Resolution	✓ RESOLVED

Issue #30	Typographical errors
Severity	● INFORMATIONAL
Description	The comments still contain the usage of "blocks" to indicate time in various locations while the contract no longer use block numbers to keep track of time.
Recommendation	Consider fixing the above typographical errors.
Resolution	✓ RESOLVED

PALADIN
BLOCKCHAIN SECURITY