PALADIN
BLOCKCHAIN SECURITY

Smart Contract Security Assessment

Final Report

For Dragon Crypto Gaming

(Legend of Aurum Draconis Staking)

25 July 2023

paladinsec.co   info@paladinsec.co

Table of Contents

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.

Paladin retains the right to re-use any and all knowledge and expertise gained during the audit process, including, but not limited to, vulnerabilities, bugs, or new attack vectors. Paladin is therefore allowed and expected to use this knowledge in subsequent audits and to inform any third party, who may or may not be our past or current clients, whose projects have similar vulnerabilities. Paladin is furthermore allowed to claim bug bounties from third-parties while doing so.

1 Overview

This report has been prepared for Dragon Crypto Gaming's The Legend of Aurum Draconis's staking contract 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	The Legend of Aurum Draconis
URL	https://dragoncrypto.io/
Network	Avalanche
Language	Solidity
Preliminary	https://github.com/Dragon-Crypto-Gaming/load-contracts/blob/0282fd04ec4c09f4e2fbe4059d50a1f31ef54ce1/contracts/vaults/TripleRewardsDividendVault.sol
Resolution 1	https://github.com/Dragon-Crypto-Gaming/load-contracts/blob/3278da040b30bd519c71e11fcf9ba1888933f888/contracts/vaults/TripleRewardsDividendVault.sol
Resolution 2	https://github.com/Dragon-Crypto-Gaming/load-contracts/blob/eb57aa2f6a888fd8d3c1d65805cb5754d929894d/contracts/vaults/TripleRewardsDividendVault.sol

1.2 Contracts Assessed

Name	Contract	Live Code Match
`TripleRewardsDividendVault`		

1.3 Findings Summary

Severity	Found	Resolved	Partially Resolved	Failed	Acknowledged (no change made)
🔴 High	15	15	-	-	-
🟠 Medium	6	5	1	-	-
🟡 Low	7	6	-	1	-
🟣 Informational	6	4	1	-	1
Total	**34**	**30**	**2**	**1**	**1**

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 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 TripleRewardsDividendVault

ID	Severity	Summary	Status
01	HIGH	A malicious user can drain the whole vault	✓ RESOLVED
02	HIGH	A malicious user can steal COPPER rewards	✓ RESOLVED
03	HIGH	Owner can change endEmissionTime without updating emissions	✓ RESOLVED
04	HIGH	Reward logic is flawed	✓ RESOLVED
05	HIGH	Owner can steal all tokens	✓ RESOLVED
06	HIGH	Owner can DoS the contract	✓ RESOLVED
07	HIGH	withdraw function is flawed	✓ RESOLVED
08	HIGH	Users might lose rewards during withdraw	✓ RESOLVED
09	HIGH	totalPendingRewards will run out of gas	✓ RESOLVED
10	HIGH	Users can receive rewards retroactively	✓ RESOLVED
11	HIGH	Some depositors will receive much more rewards than intended	✓ RESOLVED
12	MEDIUM	Rewards will be lost if only locked tokens are staked	✓ RESOLVED
13	MEDIUM	`emissionRate` will be changed retroactively	PARTIAL
14	MEDIUM	Deposits might run out of gas	✓ RESOLVED
15	MEDIUM	Consecutive token locks might reset progress	✓ RESOLVED
16	MEDIUM	Gifting of COPPER tokens results in loss of accumulated tokens	✓ RESOLVED
17	LOW	Regular harvest is not possible if user has locked tokens	✓ RESOLVED
18	INFO	Unnecessary use of SafeMath	✓ RESOLVED
19	INFO	Users should not be allowed to deposit 0 tokens	✓ RESOLVED
20	INFO	CEI pattern not adhered to	PARTIAL
21	INFO	Contract does not work with tokens that have a fee on transfer	ACKNOWLEDGED
22	INFO	Typographical issues/minor errors	✓ RESOLVED

1.3.2 Second Audit Round

ID	Severity	Summary	Status
23	HIGH	A malicious user can drain all tokens	✓ RESOLVED
24	HIGH	`lastCopperUpdateTime` is not updated properly	✓ RESOLVED
25	HIGH	Copper `rewardDebt` is not updated	✓ RESOLVED
26	HIGH	`topUp` call might result in lost rewards	✓ RESOLVED
27	MEDIUM	Emission update for copper will change rewards retroactively	✓ RESOLVED
28	LOW	Potential overflow due to increase of `copperBalance`	✓ RESOLVED
29	LOW	Very large `_emissionDays` value can result in rewards rounding down	✓ RESOLVED
30	LOW	Lock implementation can run out of gas	✓ RESOLVED
31	LOW	Withdrawals will revert unexpectedly when a user withdraws from multiple locks	✓ RESOLVED
32	LOW	Early return within `updateRewards` might have implications towards locked implementation	FAILED
33	LOW	Unnecessary balance check	✓ RESOLVED
34	INFO	Checks-effects-interactions pattern is not adhered to	✓ RESOLVED

2 Findings

2.1 TripleRewardsDividendVault

`TripleRewardsDividendsVault` is a staking contract that allows users to stake DCAR tokens for reward tokens. As the name indicates, the staker will be rewarded with three different reward tokens: DCAU, DCAR and COPPER tokens. COPPER tokens will only be rewarded for stakers that lock their tokens.

Stakers can choose either a normal deposit or a deposit with lock. For a deposit with lock, users can lock their tokens for one month, three months, six months or one year. However, there are no extra COPPER rewards for longer locks. Once a user has deposited and locked their tokens, they can increase their lock period by depositing more tokens with a higher `_lockPeriod`. Each deposit with a lock will reset the `lockTimestamp` to the current time, which means that all locked tokens will now be locked for the whole locktime again, causing the user to lose their locked progress.

Users can withdraw their unlocked stake at any time, while the locked stake is only withdrawable once the lock is over. COPPER rewards can only be harvested after `lockTime` has passed.

The `emergencyWithdraw` function allows users to withdraw unlocked tokens without receiving any rewards which might be useful in certain cases.

The contract owner can top up the DCAR and DCAU rewards by calling `topUp`, which then recalculates the emission rate for both tokens based on the new balance and the desired emission days.

Due to the large amount of high severity issues (11) we recommend the DCG team to revise the whole codebase and schedule a second audit.

2.1.1 Privileged Functions

- `setDCARToken`
- `setDCAUToken`
- `setCopperEmissionRate`
- `giftCopperRewards`
- `topUp`
- `transferOwnership`
- `renounceOwnership`

2.1.2 Issues & Recommendations

Issue #01	A malicious user can drain the whole vault
Severity	● HIGH SEVERITY
Description	`pendingRewards` uses the `totalStaked` variable as a divisor, however, it uses `user.stakedAmount` and `user.lockedAmount` for the calculation. The issue lies within the sole usage of `totalStaked` as the divisor, as this only accounts for the unlocked deposits. To further illustrate this issue, consider the following PoC: 1. Charles deposits 100 unlocked tokens and 100_000 locked tokens and waits one day. 2. Charles calls `deposit` with an amount of 0 in order to accumulate rewards (the `harvestRewards` function does not work as it is flawed) 3. Charles accumulated rewards based on `((effectiveStaked * (dcarEmissionRate * elapsedTime)) / totalStaked)` `= (100_100 * ( 1 * 86400 )) / 100)` which is highly inflated due to the flawed divisor. If we consider 1 token per second, this should be a maximum of 86400 tokens after one day, however, the result is 86486400 tokens. 4. Charles can effectively drain the whole vault via this PoC. *The actual harvest of the accumulated rewards is not trivial since several functions are flawed — Charles would need to stake unlocked tokens with a higher amount of locked tokens in order to call `withdraw` with a higher amount of unlocked tokens than locked tokens.
Recommendation	Consider using the `totalStaked + totalLocked` amount as the divisor, however, the team should take care that this does not add on any further exploit vectors. The most effective way would be to switch to a known and battle-tested reward calculation like the masterchef logic or the logic which is used for gauges from Solidly forks (these can be topped up as well).
Resolution	✓ RESOLVED

Issue #02	A malicious user can steal COPPER rewards
Severity	● HIGH SEVERITY
Description	`lockTimestamp` is only updated for two scenarios: 1. For the first locked deposit 2. For deposits that have an increased `lockPeriod` This opens the possibility for the following exploit: 1. Charles deposits 1 WEI with a `lockPeriod` of 1 year. 2. Charles waits 365 days - 10 seconds, then deposits 1_000_000 tokens with `lock=true` and `lockPeriod` as 1 year. 3. Charles now waits 10 seconds and calls `harvestRewards` and receives all rewards. He immediately dumps his 1_000_000 tokens. 4. Charles has successfully received the COPPER rewards for a full year while only having 1 wei locked for that time. Additionally, `user.lastCopperUpdate` is only updated for the first deposit.
Recommendation	The fix is non-trivial. `lockTime` could be updated for every deposit, however, that would make all unlock progress redundant. We recommend switching to a masterchef-like reward system where all rewards are accumulated correctly for each period and only paid out at the end of the lock time.
Resolution	✓ RESOLVED

Issue #03	Owner can change endEmissionTime without updating emissions
Severity	🟥 HIGH SEVERITY
Description	Within the topUp function, emissions are only updated when tokens are being sent to the contract, however, endEmissionTime can still be updated. If the owner accidentally or intentionally calls this function with no token amounts but valid _emissionDays, this might result in a side-effect where emissions are still distributed but have not been covered by the contract owner, resulting in stakers bearing the loss of the distributed rewards. After a further review of this issue it will be upgraded to high risk due the following: When the owner decides to just top up the vault with DCAU tokens, it will adjust the emission rate for DCAU only. However, it will still increase the endEmissionTime which then will result in a drainage of DCAR tokens.
Recommendation	Consider not increasing the emissionEndTime without updating the emission rates.
Resolution	✓ RESOLVED

Issue #04	Reward logic is flawed
Severity	● HIGH SEVERITY
Description	Whenever Paladin identifies a reward logic that differs from the Synthetix staking rewards / masterchef logic, we find that bugs are often introduced. The same applies here as well and we will illustrate it with the following PoC: 1. Charles deposits 100 tokens at timestamp 1000 2. Bob deposits 100 tokens at timestamp 1000 as well 3. `emissionRate` is 1 token/second 4. Charles withdraws 100 token at timestamp 1100 5. Charles therefore receives 50 reward tokens 6. Bob withdraws/harvests at timestamp 1100 as well 7. Bob now receives 100 reward tokens Consider another PoC which reflects the flawed logic with the assumption of 1 token / second: 1. Charles deposits 100 token at timestamp 100 2. Charles waits 100 seconds and deposits again 100 token at TS = 200 3. Charles now accumulated 100 tokens as reward which is assigned to the rewardDebt 4. Charles waits again 100 seconds 5. Charles deposits again 100 tokens, however, there will be no accumulated rewards due to the flawed logic, effectively resulting in lost rewards for the last 100 seconds

	A third PoC will be included as well in order to highlight the importance of switching to a successfully working logic: 1. Charles deposits 100 tokens 2. Charles waits 100 seconds 3. Charles calls deposit with `_amount=0` 4. Charles' `rewardDebt` is now 100 5. Charles wants to claim and calls `harvestRewards`, however the function reverts due to an underflow within the `_pendingRewards` calculation This issue applies for the COPPER reward logic as well Due to the modified logic, there might be further issues introduced as well, however, in order to mitigate any potential side-effects the simplest, solid solution would be to switch to the masterchef logic.
Recommendation	We highly recommend switching to an established solution for reward calculation as we assume that the current reward logic will not work even if there are partial fixes applied.
Resolution	✓ RESOLVED

Issue #05	Owner can steal all tokens
Severity	● HIGH SEVERITY
Description	Tthe owner has the ability to change DCAR. This can be abused for to steal all tokens: 1. Change DCAR to a dummy token 2. Deposit a huge amount 3. Change DCAR back to the correct tokens 4. Withdraw all tokens Another method is to change the DCAU to DCAR which then effectively distributes more DCAR as desired, effectively draining users' staked tokens. A third method which is an extension of the second method is: 1. Change the reward tokens to dummy tokens 2. Call `topUp` with huge amounts in order to artificially increase the emission rates 3. Change both tokens back to the original tokens which results in an immense reward distribution of both tokens.
Recommendation	Consider preventing DCAU and DCAR to be changed.
Resolution	✓ RESOLVED

Issue #06	Owner can DoS the contract
Severity	HIGH SEVERITY
Description	Similar to above, the owner can change the DCAU token to any token which can result in a DoS of the contract. The owner can also cause a DoS state with regards to locked tokens whenever the DCAR token is changed because this would prevent users from withdrawing locked tokens. Another method to DoS the contract is by increasing `copperEmissionRate` which might result in an overflow within `pendingCopperRewards`, or calling `giftCopperRewards` with a huge amount in order to induce an overflow within `claimCopperRewards`.
Recommendation	Consider preventing the change of DCAR and DCAU and setting an upper limit for `copperEmissionRate`.
Resolution	✓ RESOLVED

Issue #07	`withdraw` function is flawed
Severity	● HIGH SEVERITY

Description

The withdraw logic is as follows:

1. Check if the locked amount is higher than the requested amount:
   `bool lock = user.lockedAmount >= _amount;`
2. If it is, a withdraw is only allowed if `lockTime` has passed:

```
if (lock) {
    require(block.timestamp >=
getLockExpiry(user.lockPeriod, user.lockTimestamp),
"Lock period has not passed yet");
claimCopperRewards();
}
```

This means that a user with a higher locked than unlocked stake will not be able to use the normal `withdraw` function in order to withdraw unlocked tokens, resulting in the need to call `emergencyWithdraw` which allows users to withdraw without rewards. However, the contract has a `harvestRewards` function which is exactly meant for that case. Unfortunately, it is flawed as well.

```
if (user.lockedAmount > 0) {
    require(block.timestamp >=
getLockExpiry(user.lockPeriod, user.lockTimestamp),
"Lock period has not passed yet");
claimCopperRewards();
}
```

The user is left with the choice to either burn the reward or wait until `lockTime` has passed.

Recommendation

Consider revising the withdraw `function` to include the option to withdraw either unlocked tokens or locked tokens. Both options should be strictly separated in order to not encounter any collisions.

Resolution	✓ RESOLVED

Issue #08 | **Users might lose rewards during withdraw**

Severity | ● HIGH SEVERITY

Description

The `withdraw` function allows users to withdraw tokens + harvest rewards. However, regular rewards are only harvested if the user has unlocked tokens staked:

```
if (user.stakedAmount > 0) {
    dcarTotal += dcarPending + user.rewardDebtDCAR;
    dcauTotal += dcauPending + user.rewardDebtDCAU;
 }
```

This will cause users to lose their rewards in the following scenarios:

1. Charles locks 100 tokens for 1 month
2. Charles decides to withdraw 50 tokens after this month
3. Since Charles only has locked tokens, `dcarTotal` and `dcauTotal` will not be increased but the tokens will be withdrawn and `user.lastUpdateTime` will be updated
4. Charles effectively lost all rewards besides the COPPER rewards

A second scenario is also present:

1. Charles stakes 100 tokens
2. Charles waits 100 seconds
3. Charles withdraws his stake and expects the rewards to be withdrawn as well

However, a flaw in the `withdraw` logic causes Charles to lose all his rewards:

```
} else {
   totalStaked -= _amount;
   user.stakedAmount -= _amount;
}
dcarTotal += _amount;
}
if (user.stakedAmount > 0) {
   dcarTotal += dcarPending + user.rewardDebtDCAR;
   dcauTotal += dcauPending + user.rewardDebtDCAU;
}
```

	As highlighted in the section above, `user.stakedAmount` is in fact deducted before the check happens which effectively results in Charles losing all his rewards.
Recommendation	Consider also taking into account accumulated rewards with locked tokens. Additionally, consider accounting for the rewards before the amount has been deducted. In any case, we highly recommend a total revision of the whole function.
Resolution	✓ RESOLVED

Issue #09	**`totalPendingRewards` will run out of gas**
Severity	● HIGH SEVERITY
Description	`totalPendingRewards` is used by the `topUp` function to calculate the reward amount which is still assigned to users. However, this only works up to a certain point as the loop will eventually run out of gas, effectively preventing the owner from adding any further rewards.
Recommendation	Consider switching to a different logic than aggregating all users in a huge array.
Resolution	✓ RESOLVED

Issue #10	Users can receive rewards retroactively
Severity	● HIGH SEVERITY
Description	If the contract has no assigned rewards yet, a user can still deposit which sets `user.lastTimestamp = block.timestamp`. A user can thus wait until the owner adds rewards to the contract without executing another deposit in order to receive rewards retroactively.
Recommendation	Consider switching to the well-known masterchef logic and update all pools before making any change to the `emissionRate`.
Resolution	✓ RESOLVED

Issue #11	Some depositors will receive much more rewards than intended
Severity	● HIGH SEVERITY

Description	Within `pendingRewards`, the elapsed time since last user rewards info update is calculated in the following way:
	`if (block.timestamp > endEmissionTime) {`
	`    elapsedTime = user.lastUpdateTime > endEmissionTime ? user.lastUpdateTime : endEmissionTime - user.lastUpdateTime;`
	`} else {`
	`    elapsedTime = block.timestamp - user.lastUpdateTime;`
	`}`
	When the end of rewards emission is in the past and the last update for the user has been made after this, the last update timestamp is returned as the elapsed time when 0 should be returned instead.
	This can be abused for example by early depositors calling the `deposit` function twice before `topUp` is called so that `endEmissionTime` is still 0 and they get an extremely high amount of rewards.

Recommendation	Consider replacing `user.lastUpdateTime` with `0` on the above line in `pendingRewards`.
Resolution	✓ RESOLVED

Issue #12	Rewards will be lost if only locked tokens are staked
Severity	● MEDIUM SEVERITY
Description	`pendingRewards` sets the pendings to zero whenever `totalStaked = 0`. However, if there are locked tokens staked, the reward for these tokens will be stuck in the contract.
Recommendation	Consider using `totalStaked + totalLocked` as the check variable.
Resolution	✓ RESOLVED

Issue #13	`emissionRate` will be changed retroactively
Severity	● MEDIUM SEVERITY
Description	Both `emissionRates` will be changed during the `topUp` function. The owner can either increase the rate by topping up rewards / using lower `emissionDays` than the previous value or decrease it by increasing the `emissionDays` without properly topping up the rewards. However, both actions will change `emissionRates` retroactively which means that users that still have a pending reward will have this reward either decreased or increased. The same issue applies for the update of the copper emissions.
Recommendation	Consider switching to a reputable reward logic which uses an `accPerShare` variable and updating this variable before adjusting the emission rates.
Resolution	● PARTIALLY RESOLVED Changing the `copperEmissionRate` will still change rewards retroactively.

Issue #14	Deposits might run out of gas
Severity	● MEDIUM SEVERITY
Description	The `deposit` function loops over all staked users to check whether the address has already deposited tokens. However, this loop will run out of gas at some point.
Recommendation	Consider a change of this logic — an `enumerableSet` might be one solution.
Resolution	✓ RESOLVED

Issue #15	Consecutive token locks might reset progress
Severity	● MEDIUM SEVERITY

Description	Each consecutive token lock will be handled in the following clause:
	
	`require(_lockPeriod >= user.lockPeriod, "Cannot reduce lock period");`
	`if (_lockPeriod > user.lockPeriod) {`
	`    user.lockPeriod = _lockPeriod;`
	
	`    user.lockTimestamp = block.timestamp;`
	`}`
	
	This sets `user.lockTimestamp` to the current timestamp when a user decides to choose a higher `lockPeriod`. While we agree that it is necessary to prevent an exploit where users deposit a small token amount and later deposit a huge token amount in order to account for the passed `lockTime` for the huge token amount, this logic completely resets the progress from the previous deposit.
	
	Another flaw which was introduced due to this logic is the deposit of locked tokens with a similar `lockPeriod`, which causes `lockTimestamp` to not be updated effectively, allowing the user to abuse the system by depositing a small amount first and a large amount later.

Recommendation	Consider switching to masterchef-like logic where all past reward accumulations are properly accounted for, in order to not accidentally reset a user’s progress.
Resolution	✓ RESOLVED

Issue #16	Gifting of COPPER tokens results in loss of accumulated tokens
Severity	● MEDIUM SEVERITY
Description	`giftCopperRewards` allows the owner to increase the COPPER balance of arbitrary addresses. However, it also updates the last copper update time: `user.lastCopperUpdate = block.timestamp;` Whenever this time is updated, all accumulated rewards are effectively erased from the used address.
Recommendation	Consider removing this update in order to not reset the users progress, and consider the potential side-effects of doing this as well.
Resolution	✓ RESOLVED

Issue #17	Regular `harvest` is not possible if user has locked tokens
Severity	● LOW SEVERITY
Description	`harvestRewards` is meant for users to harvest their accumulated rewards. However, if a user has locked tokens, they are unable to call this function if the `unlockTime` has not been reached: `if (user.lockedAmount > 0) { require(block.timestamp >= getLockExpiry(user.lockPeriod, user.lockTimestamp), "Lock period has not passed yet"); claimCopperRewards(); }` This `require` statement will therefore always revert the whole function
Recommendation	Consider not reverting for that case by simply not calling `claimCopperRewards`.
Resolution	✓ RESOLVED

Issue #18	Unnecessary use of SafeMath
Severity	● INFORMATIONAL
Description	The contract still imports and uses SafeMath occasionally, however, the contract is compiled with Solidity 0.8.0 which makes SafeMath redundant.
Recommendation	Consider removing SafeMath.
Resolution	✓ RESOLVED

Issue #19	Users should not be allowed to deposit 0 tokens
Severity	● INFORMATIONAL
Description	Allowing users to be able to deposit 0 tokens is not recommended as it might result in unexpected behavior in the staking flow.
Recommendation	Consider reverting if users try to deposit 0 tokens.
Resolution	✓ RESOLVED

Issue #20	CEI pattern not adhered to
Severity	● INFORMATIONAL

Description	The contract contains sections that do not adhere to the CEI standard:
	`} else {`
	`totalStaked += _amount;`
	` user.stakedAmount += _amount;`
	` } // checks-effects-interactions pattern`
	`dcarToken.safeTransferFrom(msg.sender, address(this), _amount);`
	`}`
	
	`user.lastUpdateTime = block.timestamp; user.rewardDebtDCAR += dcarPending;`
	`user.rewardDebtDCAU += dcauPending;`
	Even though the comment indicates the CEI pattern, it is in fact not used. The transfer of the token should be at the end of the function

Recommendation	Consider implementing the checks-effects-interactions pattern throughout the whole codebase.
Resolution	● PARTIALLY RESOLVED

Issue #21	Contract does not work with tokens that have a fee on transfer
Severity	● INFORMATIONAL
Description	The contract does not properly account for tokens that have a fee on transfer. However, since this contract is only designed for DCAU and DCAR this issue will just be informational.
Recommendation	If the team plans to use such tokens in future, consider implementing logic that accounts for tokens with a fee on transfer. Otherwise, this issue can simply be resolved with an acknowledgement from the team.
Resolution	● ACKNOWLEDGED

Issue #22	Typographical issues/minor errors
Severity	● INFORMATIONAL

Description	Line 48
	`event EmergencyWithdraw(address indexed user, uint256 amount, bool lock);`
	
	This event is unused, consider using it.
	
	Line 228
	`require(user.lockedAmount > 0, "No locked tokens to claim rewards from”);`
	
	This check is redundant as the logic related to when this function is called already ensures this.
	
	Line 150
	`function getUserCopperBalance(address user) public view returns (uint256) {`
	
	`user` could be renamed to `_user` to be consistent with the naming convention in the codebase.
	
	Line 371
	`require(user.copperBalance >= amount, "Not enough Copper balance");`
	
	The above check in spendCooper is redundant since the subtraction on the next line will revert if this condition is not met.
	
	Line 379
	`* @notice Get pending rewards for a user`
	
	The natspec comment for `totalPendingRewards` is incorrect.

Lines 418, 423, 427

```
dcarEmissionRate = (currentDCARBalance + _dcarAmount) /
(_emissionDays * 86400); // Convert days to seconds
dcauEmissionRate = (currentDCAUBalance + _dcauAmount) /
(_emissionDays * 86400); // Convert days to seconds
endEmissionTime = block.timestamp + _emissionDays * 1 days;
```

Consider using only 1 days or only 86400 for consistency.

Line 426

```
lastUpdateTime = block.timestamp;
```

The global variable `lastUpdateTime` is updated in `topUp` but is never used.

Recommendation	Consider fixing the typographical issues.
Resolution	✓ RESOLVED

2.2 Second Audit Round

Due to the large number of issues within the first audit round, a second audit round was scheduled with the goal to identify any potential issues that can arise within the resolution round.

2.2.1 Issues & Recommendations

Issue #23	A malicious user can drain all tokens
Severity	● HIGH SEVERITY

Description	Whenever `emergencyWithdraw` is called, a user's `rewardDebt` will be reset:
	`user.rewardDebtDCAR = 0;`
	`user.rewardDebtDCAU = 0;`
	However, the user still has a locked stake which then can be abused to steal all tokens when claiming (invoking `deposit`/`withdraw`):
	`dcarPending = (userEffectiveTotalStaked * _accDCARPerShare) / WAD - user.rewardDebtDCAR; dcauPending = (userEffectiveTotalStaked * _accDCAUPerShare) / WAD - user.rewardDebtDCAU;`
	Since `rewardDebt` was set to zero beforehand, a user can call the `withdraw` or `deposit` function which then uses the locked stake to drain all tokens.
	This process can be repeated until all tokens are stolen.

Recommendation	Consider setting the `rewardDebt` to correspond to the current `rewardPerShare` and the locked amount of the user.
Resolution	✓ RESOLVED `rewardDebt` is now set to the corresponding locked amount.

Issue #24	`lastCopperUpdateTime` is not updated properly
Severity	● HIGH SEVERITY

Description	Within `updateRewards`, `lastCopperUpdateTime` is not updated if the reward rate is zero:
	`if (totalLocked > 0 && copperEmissionRate > 0 && block.timestamp >= lastCopperUpdateTime) {`
	`    uint256 secondsElapsedCopper = block.timestamp - lastCopperUpdateTime;`
	`    accCOPPERPerShare += secondsElapsedCopper * copperEmissionRate * WAD / totalLocked;`
	`    lastCopperUpdateTime = block.timestamp;`
	`}`
	This will then distribute rewards based on `lastCopperUpdateTime` whenever the reward rate is increased, resulting in a huge amount of rewards.

Recommendation	Consider updating `lastCopperUpdateTime` under all circumstances.
Resolution	✓ RESOLVED However during the attempt to fix issue #10, there was a bug implemented which impacts this logic as well. There could now be a state where `accCopperPerShare` is increased but `lastCopperTime` not updated. Consider reversing the change for #10.

Issue #25	Copper `rewardDebt` is not updated
Severity	● HIGH SEVERITY
Description	Within the `withdraw` function, `rewardDebtCopper` is not updated after the locked amount is decreased, which will result in underflow whenever the `pendingCopperRewards` function is invoked: `copperPending = (user.lockedAmount * _accCOPPERPerShare) / WAD - user.rewardDebtCOPPER;` `lockedAmount` will be smaller while the `rewardDebtCopper` is still based on the pre-withdraw `lockedAmount`.
Recommendation	Consider updating the `rewardDebtCopper` to comply with the decreased `lockBalance`.
Resolution	✓ RESOLVED

Issue #26	`topUp` call might result in lost rewards
Severity	● HIGH SEVERITY

Description	Whenever `topUp` is called and the current time is larger than `emissionEndTime`, `lastUpdateTime` will be set to `block.timestamp`:
	
	`if (block.timestamp > endEmissionTime) {`
	    `lastUpdateTime = block.timestamp;`
	`}`
	
	This will result in A loss of rewards if the rewards have not been updated and the `lastUpdateTime` is less than `endEmissionTime`.

Recommendation	Consider updating the pools as first call within the function to ensure all rewards have in fact been allocated.
Resolution	✓ RESOLVED

Issue #27	Emission update for copper will change rewards retroactively
Severity	● MEDIUM SEVERITY
Description	`setCopperEmissionRate` changes the copper emission rate. However, there is no `updateRewards` call executed before the emission rate is changed which will result in an emission rate change retroactively.
Recommendation	Consider calling `updateRewards` before the emission rate is changed.
Resolution	✓ RESOLVED

Issue #28	Potential overflow due to increase of `copperBalance`
Severity	● LOW SEVERITY
Description	The contract owner can increase the COPPER balance of any address without any limitations, if the balance is increased by a huge amount this will potentially overflow in `claimCopperRewards` resulting in a DoS. This issue is only rated as low severity because the user can manually spend/decrease the copper balance.
Recommendation	Consider implementing an upper limit for the COPPER balance increase as well as a timelock feature per address to prevent any abuse of this functionality.
Resolution	✓ RESOLVED

Issue #29	Very large `_emissionDays` value can result in rewards rounding down
Severity	● LOW SEVERITY
Description	The `topUp` function allows the setting of `_emissionDays` which declares the length of the following reward period. However, if this value is too large, it might round down the `emissionRates` to zero.
Recommendation	Consider setting a reasonable upper limit for this variable.
Resolution	✓ RESOLVED

Issue #30	Lock implementation can run out of gas
Severity	● LOW SEVERITY
Description	Users can have unlimited locked positions which poses a risk whenever a user with a large amount of locks executes a large withdrawal. The execution logic loops over all existing positions until the remaining variable becomes zero. This logic might run out of gas if the user tries to withdraw a huge amount. Additionally, the contract could also run out of gas if the only unlocked position is very far at the end of the array. However, the likelihood for this case is almost zero since in case of locked positions no gas-consuming logic is executed within the loop.
Recommendation	Consider setting an upper limit for the amount of lock positions a user can create.
Resolution	✓ RESOLVED The client has implemented a limit for the total user lock amount — this variable will be decreased whenever a lock is deleted.

Issue #31	**Withdrawals will revert unexpectedly when a user withdraws from multiple locks**
Severity	● LOW SEVERITY

Description	Whenever users withdraw locked tokens, a loop over all locked positions will get executed, which results in a position being removed if it was cleared and the length of the array is then decreased.
	`function removeLock(address _user, uint _index) internal {`
	`    Lock[] storage locks = userLocks[_user];`
	`    locks[_index] = locks[locks.length - 1];`
	`    locks.pop();`
	`}`
	The problem is that the length of the array is cached before the for loop and therefore the variable will not be updated when the actual array length decreases in `removeLock`.
	`uint256 totalLocks = userLocks[msg.sender].length;`
	`for (uint i = 0; i < totalLocks && remaining > 0; ) {`
	Therefore, for each unlocked position, the `totalLocks` value will be off by 1 and at some point will revert due to reading a non-existent position which was removed in previous iterations.

Recommendation	Consider reducing `totalLocks` by 1 each time `removeLock` is called.
Resolution	✓ RESOLVED

Issue #32	**Early return within `updateRewards` might have implications towards locked implementation**
Severity	● LOW SEVERITY

Description	`updateRewards` returns whenever the `lastUpdateTime` is larger or equal to `block.timestamp`:
	
	`if (block.timestamp <= lastUpdateTime) {`
	`    return;`
	`}`
	
	This will then not update `accCopperPerShare` which will result in users losing COPPER rewards.
	
	However, at the current point, we could not identify a state where this would result in the mentioned negative side-effects but we still recommend mitigating this issue.

Recommendation	Consider executing the logic for the locked/copper implementation before the potential return.
Resolution	● FAILED RESOLUTION While `accCopperPerShare` is updated, the early return will prevent the update of `lastCopperUpdateTime`. As previously mentioned, due to the fact that such a state will not exist and an issue was introduced during the attempt to fix it, we recommend reversing this change and keep it as it was.

Issue #33	Unnecessary balance check
Severity	● LOW SEVERITY
Description	The following check ensures that a staker has at least the minimum amount of locked or staked tokens: `require(user.stakedAmount >= _amount || user.lockedAmount >= _amount, "Not enough balance to withdraw”);` While this check itself is redundant because the accounting underflows, in Solidity versions below 0.8.0, this check could have been bypassed (if the additional checks within the certain flows would not have been present) because the check does not additionally check if a withdraw is from a locked or unstaked portion. For example, users could withdraw a huge amount of unstaked tokens, as long as their `lockedAmount` is large enough.
Recommendation	Consider removing this functionality since it serves no purpose, however, this issue should emphasise the need for attention to detail when developing smart contracts.
Resolution	✓ RESOLVED

Issue #34	Checks-effects-interactions pattern is not adhered to
Severity	● INFORMATIONAL
Description	The contract contains sections that do not adhere to the CEI standard. L483 `require(dcarEmissionRate <= MAX_DCAR_EMISSIONS, "DCAR emission rate too high"); require(dcauEmissionRate <= MAX_DCAU_EMISSIONS, "DCAU emission rate too high”);` This check should be executed before the transfer of the tokens.
Recommendation	Consider following the adhering to the CEI pattern.
Resolution	✓ RESOLVED

PALADIN

BLOCKCHAIN SECURITY