OpenZeppelin | security

The Sandbox SAND Staking Audit

December 14, 2022

This security assessment was prepared by OpenZeppelin.

Table of Contents

Summary

Type	Staking
Timeline	From 2022-11-07 To 2022-11-23
Languages	Solidity

Total Issues	26 (18 resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	0 (0 resolved)
Medium Severity Issues	3 (2 resolved)
Low Severity Issues	14 (10 resolved)
Notes & Additional Information	9 (6 resolved)

Scope

We audited the The Sandbox repository at the 095f29c55af04381886d54f2acf63017bb8e18e7 commit.

In scope were the following contracts:

```
contracts/src/solc_0.8
├─ common/
│  ├─ BaseWithStorage/
│  │  └─ ERC2771HandlerV2.sol
│  └─ Libraries/
│     └─ SafeMathWithRequire.sol
└─ defi/
   ├─ ERC20RewardPool.sol
   ├─ StakeTokenWrapper.sol
   ├─ interfaces/
   │  ├─ IContributionCalculator.sol
   │  ├─ IContributionRules.sol
   │  └─ IRewardCalculator.sol
   ├─ rewardCalculation/
   │  └─ TwoPeriodsRewardCalculator.sol
   └─ rules/
      ├─ ContributionRules.sol
      ├─ LockRules.sol
      └─ RequirementsRules.sol
```

System Overview

The set of audited contracts implements a staking pool where users, who meet certain criteria, can stake tokens and earn rewards according to the ratio of their contribution to the total contribution. The system also supports meta-transactions via `ERC2771HandlerV2` contract as well as ownership and role-based access controls using OpenZeppelin's Ownable and AccessControl.

The contract implementing the pool's core functionality is `ERC20RewardPool`. The different configurations that the protocol uses are implemented in the following contracts:

- `TwoPeriodsRewardCalculator` implements the reward calculation logic.

- `ContributionRules` defines how assets (i.e. specific NFTs) owned by users affect the calculation of their rewards.
- `LockRules` defines time buffers for different operations such as: withdrawals, deposits, reward claims and maximum deposit amounts.
- `RequirementRules` implements the criteria that users must meet in order to stake funds in the pool.

Privileged Roles and Trust Assumptions

Within the system, there are a few privileged roles whom users must place their trust in; the `ERC20RewardPool` owner and trusted forwarder, the `TwoPeriodsRewardCalculator` admin and reward distributor, and the `ContributionRules` owner.

The owner of the `ERC20RewardPool` is originally set as the deployer of the contract, however, the owner can transfer the ownership to another address at anytime. The owner also has the capability to set and update the following contract configurations:

- Set and update the stake and reward token addresses.
- Set and update the trusted forwarder address.
- Set and update the parameters in the `LockRules` contract.
- Set and update the parameters in the `RequirementRules` contract.
- Set and update the reward calculation contract.
- Set and update the contribution rules contract.

Additionally, the owner may recover any rewards balance on the contract and transfer them to an arbitrary address; to do so the owner must pause the contract first and if the stake and reward tokens are the same, the staked amount is subtracted from the amount that will be recovered.

Note: Users should be aware that, given these capabilities, it is possible for a compromised owner account to steal staked funds in the scenario where the stake and reward tokens are the same token.

Note: The conditions for setting a new stake or reward token is that the pool's balance of the new token must be equal to or greater than its balance on the current token. Because it is possible to change to a token with less *value*, causing users a loss when withdrawing their stake, this feature should be strictly used only when a token has been migrated to a new version and the old address is now deprecated.

To support meta-transactions and also provide users with a certain amount of gas-free transactions per month a trusted forwarder is used. The address configured as the `_trustedForwarder` in the `ERC20RewardPool` contract can perform operations on behalf

of user's. The current trusted forwarder as of this writing is the Biconomy trusted forwarder at the address 0xf0511f123164602042ab2bcf02111fa5d3fe97cd.

Note: The code of the trusted forwarder was out of scope of this audit.

The `TwoPeriodsRewardCalculator` defines the following roles:

- `DEFAULT_ADMIN_ROLE`: Assigned to the contract's deployer. It is the admin of all roles and can grant and revoke roles to and from addresses respectively.
- `REWARD_DISTRIBUTION`: It can start new campaigns to distribute rewards or modify existing ones and can set a saved reward which will be added to the current campaign's rewards.

The `ContributionRules` owner can set, update, and delete ERC721 and ERC1155 contribution list entries, affecting the multiplier that users will receive on their stake.

Findings

Here we present our findings.

Medium Severity

M-01 Lack of event emission after state changes

The following functions do not emit relevant events after executing sensitive actions that change the state of the contract:

`ERC20RewardPool` contract

- `setRewardToken`
- `setStakeToken`
- `setTrustedForwarder`
- `setContributionRules`
- `setRewardCalculator`
- `recoverFunds`

`TwoPeriodsRewardCalculator` contract:

- `setSavedRewards`

Consider emitting events after sensitive changes take place to facilitate off-chain monitoring of the contract's activity.

Update: *Resolved in PR #831.*

M-02 Claimed rewards can dip into staked tokens

In the contract `ERC20RewardPool`, users that have staked funds for a period of time can claim rewards. The staked funds and rewards may be different tokens or the same token. In the case where the staked funds and the rewards token are the same token, if enough rewards have been accumulated and claimed, it is possible for a claim on rewards to dip into the balance that is reserved for the users' stake. The function `_withdrawRewards`, called by `exit` and `getReward`, does not check that the amount being claimed is less than the amount of *available rewards*. If there was a run on the contract and everyone exited by calling `exit`, some users would not be able to withdraw their stake. This situation can be mitigated by Sandbox Foundation depositing additional funds into the contract; however, without timely

monitoring and deposits, end-users might be impacted due to inability to liquidate their funds immediately.

To ensure that there are enough funds for users to always unstake, consider adding a check in the claim rewards process to ensure the funds are being taken from those set aside as rewards and not another user's stake.

Update: *Resolved in PR #847 and commits 95f27f4 and 0d540db.*

M-03 Rules can be circumvented using NFT flashloans

Within the system there are `ContributionRules` and `RequirementsRules` which dictate the stake multiplier received and the required assets to be allowed to stake. These rules can be circumvented by using protocols that allow flashloaning ERC721s and ERC1155s. Since the rules only check for holding specific assets at the time of an action, it is possible to flashloan a required and/or contributing NFT - thereby maximizing the allowed stake and contribution multiplier, stake some funds, and repay the loaned NFT.

Currently, this doesn't have a large affect as the contracts listed as contribution and requirement rules don't have much depth in the pools/protocols that allow for these flashloans. However, it is possible that at some point in the future, flashloaning NFTs will have more volume and may impact the possibility of the rewards system being abused.

To reduce the possibility of the rewards system's rules being abused, consider requiring alternative rule(s) that would prevent this type of activity.

Update: *Acknowledged, not resolved. The Sandbox team stated:*

> *We acknowledge this issue. We have developed an anti-cheat tracker that will automatically update the contribution of the staker when a LAND is transferred and when the user stakes more than they are allowed to.*

Low Severity

L-01 Lack of input validation

In the contract `ContributionRules`, the function `setERC1155MultiplierList` accepts two arrays, `ids` and `multipliers`, that are expected to be the same length. The function does not validate this assumption, making it possible to have arrays with different sizes. Without this check, it is possible that an id will not have a multiplier set; this will cause any transaction for a user who owns this id to revert until the id and multiplier configuration is updated by the admin of the contract.

In contrast, the function `setERC721MultiplierList` does check that both arrays are equal in size and reverts otherwise.

To prevent a scenario where a user's action can be reverted and to increase consistency, consider checking that the `multipliers` array length is the same as `ids` array length.

Update: *Resolved in PR #832 and commit 66e67c1.*

L-02 `SafeMathWithRequire` results in non-negligible differences for very large numbers

The functionality in the library `SafeMathWithRequire` begins to show non-negligible differences from standard implementations for very large numbers. Each function has its own boundary where these differences begin.

To reduce the potential for misusing this code in the future, consider adding developer docs or comments on the boundaries that these functions work correctly for.

Update: *Acknowledged, not resolved. The Sandbox team stated:*

> *We will update the respective documentation to make it clear to users/developers that discrepancies can happen.*

L-03 maxStakeOverall is the effective maxStake if set

In the contract `RequirementsRules.sol`, the function `_maxStakeAllowedCalculator` is used to calculated the maximum amount a specific user can stake. Each requirement rule has an amount of stake that is allowed based on the assets, ERC721s and ERC1155s, that a user owns. There are two components that are used in the calculation of a user's effective `maxStake`:

- the sum of the amounts from the intersection of the ERC1155 and ERC721 assets the user owns and the respective requirements lists; lets call this amount `userMaxStakeFromAssets`.
- the state variable `maxStakeOverall`; this is meant to be used when a user has no ERC1155 or ERC721 that are on the requirements list.

The max stake defaults to `maxStakeOverall` but is changed to `userMaxStakeFromAssets` if a user's `userMaxStakeFromAssets` is greater than 0 and less than `maxStakeOverall`.

Given this logic, if there are requirements lists, the `maxStakeOverall` is set, and a user's `userMaxStakeFromAssets` is lower than the `maxStakeOverall`, it is simple for that user to remove the restricting amount set by `userMaxStakeFromAssets` by using an address with no assets on the requirements list. Effectively, if the `maxStakeOverall` is set, it becomes the maximum stake amount for *all* users.

To prevent users from removing the restricting stake amount set by the requirements rules, consider adding checks that limit the `maxStakeOverall` to less than the lowest amount possible from meeting a requirement rule.

Update: *Acknowledged, not resolved. The Sandbox team stated:*

> *We updated the Dev docstring mentioning maxStakeOverall value to be attentive when setting this value to not benefit those who do not own any assets. Adding checks to the code to compare and check the requirements lists is very complex and not worth doing since this case is very unlikely to happen.*

L-04 Missing docstrings

Throughout the codebase there are several parts that do not have docstrings. For instance:

- Line 15 in `ERC2771HandlerV2.sol`
- Line 19 in `ERC2771HandlerV2.sol`
- Line 326 in `ERC20RewardPool.sol`
- Line 9 in `StakeTokenWrapper.sol`
- Line 5 in `IContributionRules.sol`
- Line 9 in `TwoPeriodsRewardCalculator.sol`
- Line 124 in `TwoPeriodsRewardCalculator.sol`
- Line 12 in `ContributionRules.sol`
- Line 71 in `ContributionRules.sol`
- Line 87 in `ContributionRules.sol`
- Line 95 in `ContributionRules.sol`
- Line 103 in `ContributionRules.sol`
- Line 128 in `ContributionRules.sol`
- Line 157 in `ContributionRules.sol`
- Line 161 in `ContributionRules.sol`
- Line 171 in `ContributionRules.sol`
- Line 181 in `ContributionRules.sol`
- Line 197 in `ContributionRules.sol`
- Line 213 in `ContributionRules.sol`
- Line 217 in `ContributionRules.sol`
- Line 221 in `ContributionRules.sol`
- Line 242 in `ContributionRules.sol`
- Line 259 in `ContributionRules.sol`
- Line 275 in `ContributionRules.sol`
- Line 10 in `LockRules.sol`
- Line 91 in `LockRules.sol`
- Line 98 in `LockRules.sol`
- Line 105 in `LockRules.sol`
- Line 113 in `LockRules.sol`
- Line 123 in `LockRules.sol`
- Line 133 in `LockRules.sol`
- Line 11 in `RequirementsRules.sol`
- Line 102 in `RequirementsRules.sol`
- Line 109 in `RequirementsRules.sol`
- Line 153 in `RequirementsRules.sol`

- Line 179 in `RequirementsRules.sol`
- Line 189 in `RequirementsRules.sol`
- Line 199 in `RequirementsRules.sol`
- Line 215 in `RequirementsRules.sol`
- Line 231 in `RequirementsRules.sol`
- Line 235 in `RequirementsRules.sol`
- Line 239 in `RequirementsRules.sol`
- Line 263 in `RequirementsRules.sol`
- Line 280 in `RequirementsRules.sol`
- Line 286 in `RequirementsRules.sol`
- Line 299 in `RequirementsRules.sol`
- Line 311 in `RequirementsRules.sol`
- Line 328 in `RequirementsRules.sol`

To ensure documentation is generated correctly, consider adding docstrings and NatSpec to adhere to the Solidity Style Guide on NatSpec style.

Update: *Resolved in PR #833 and commit e824166.*

L-05 Use role-based access control instead of Ownable

The `ERC20RewardPool` contract has many functions and capabalities that are reserved for the owner of the contract. While using an owner is a good practice in some cases, in cases where there are multiple separate privileged functionalities, it may be better to use role-based access control (RBAC). RBAC can reduce the potential for hacks via a single compromised account, and divides the privileges and responsibilities between multiple trusted entities instead of a single owner.

To reduce the risk of a single point of failure via the owner account, consider implementing RBAC in the rewards pool contract.

Update: *Acknowledged, not resolved. The Sandbox team stated:*

> *We previously used role-based access control on this contract, but as this contract is built from several contracts that also use Ownable, we decided to remain with just the Ownable to avoid confusion between contracts and roles.*

L-06 Unnecessary code

Throughout the codebase, there are instances where unnecessary code adds complexity and confusion to the functionality and flow. For example:

- In the function `_withdrawStake`, the code on line 332 is unnecessary since the same code has already been executed in the modifier `antiWithdrawCheck` on line 66 without any other side effects that would change the state happening in between.
- In the function `stake`, the code on line 281 is unnecessary since the same code has already been executed in the modifier `antiDepositCheck` on line 78 without any other side effects that would change the state happening in between.
- The `else` branch in `_maxStakeAllowedCalculator` is unnecessary since line 367 sets `maxAllowed` to `maxStakeOverall` already.
- The variable `_totalBal` and line 322 is unnecessary since the `_totalBal` is reset to 0 every iteration of the loop and `balanceId` is the only amount added to it so it will be equal to the `balanceId`.

To improve readability and reduce code complexity, consider removing any unnecessary code.

Update: *Resolved in PR #834 and commit 438cb45.*

L-07 Consistency of calculating `rewardsAvailable`

In the contract `ERC20RewardPool`, if the `rewardToken` and the `stakeToken` are the same contract, the function `getRewardsAvailable` calculates the available rewards by using the `stakeToken` variable to get the balance of the ERC20RewardPool contract while the function `recoverFunds` uses the `rewardToken` variable to get the same balance.

For consistency and clarity, consider using the same variable names to calculate the rewards for both instances of this case.

Update: *Resolved in PR #835 and commit aa7d872.*

L-08 Require statements with multiple conditions

Throughout the codebase there are `require` statements that requires multiple conditions to be satisfied. For instance:

- The `require` statement on line 108 of `ContributionRules.sol`.
- The `require` statement on line 134 of `ContributionRules.sol`.
- The `require` statement on line 118 of `RequirementsRules.sol`.
- The `require` statement on line 159 of `RequirementsRules.sol`.

To simplify the codebase and to raise the most helpful error messages for failing `require` statements, consider having a single condition per require statement.

Update: *Resolved in PR #836 and commits c0dceec and bf8ec0e.*

L-09 Simplify `isContractAndAdmin` modifier

The modifier `isContractAndAdmin` in the ERC20RewardPool contract unnecessarily reimplements the `onlyOwner` modifier that is inherited through `Ownable`.

For clarity and conciseness, consider removing the only owner check, renaming this modifier, and adding `onlyOwner` directly to the functions that need it.

Update: *Resolved in PR #837 and commit 1e294b5.*

L-10 Unnecessary variable

Within the `StakeTokenWrapper` contract, the `__gap` array is defined in a system that is not using the upgradable pattern. This type of variable is usually used in order to reserve storage slots for future state variables when a contract can be upgraded.

Since this contract is not being used in an upgradable manner, consider removing the `__gap` array.

Update: *Resolved in PR #838 and commit a630894.*

L-11 Potential for incorrect values to be returned

The `TwoPeriodsRewardCalculator` contract defines two periods of rewards each with their own finish time, `finish1` and `finish2`, and rates, `rate1` and `rate2`. The first

period is meant to represent the current rewards campaign and the second period is meant to represent the next campaign. This structure allows the second campaign to begin before the admin of the contract has updated it to the current campaign resulting in a fluid transition to the next reward campaign. The contract also defines the functions `getRate` and `getFinish` that should return the current campaign's finish time and rate, however, these functions return fixed values `rate1` and `finish1` respectively.

While this makes sense in most cases when the current campaign is defined by `finish1` and `rate1`, there are times after a campaign finishes but before the new campaign is updated by the admin where `rate2` and `finish2` are the current campaign. Due to this, it is possible for these functions to return an incorrect rate and finish time.

To ensure the current campaign's finish time and rates are returned, consider adjusting these functions to return a value based on the current time and the time of the campaigns.

Update: *Resolved in PR #839 and commits b6d56e2 and 26dee9c.*

L-12 Incorrect docstring comment format

Throughout the codebase there are several contracts and/or functions that have some docstrings that are not in the correct comment format or are not directly above the intended contract and/or function. For instance:

- Line 8 in `ERC2771HandlerV2.sol`
- Line 431 in `ERC20RewardPool.sol`
- Line 437 in `ERC20RewardPool.sol`
- Line 70 in `TwoPeriodsRewardCalculator.sol`
- Line 75 in `TwoPeriodsRewardCalculator.sol`
- Line 80 in `TwoPeriodsRewardCalculator.sol`
- Line 85 in `TwoPeriodsRewardCalculator.sol`
- Line 92 in `TwoPeriodsRewardCalculator.sol`
- Line 99 in `TwoPeriodsRewardCalculator.sol`
- Line 109 in `TwoPeriodsRewardCalculator.sol`
- Line 118 in `TwoPeriodsRewardCalculator.sol`
- Line 131 in `TwoPeriodsRewardCalculator.sol`
- Line 136 in `TwoPeriodsRewardCalculator.sol`

To ensure documentation is generated correctly, consider updating the comment format to adhere to the Solidity Style Guide on documentation style.

Update: *Resolved in PR #840 and commit 3df5afe.*

L-13 Id and contract limits can be constant

In the contract RequirementsRules.sol, the state variables `idsLimit` and `contractsLimit` cannot be updated and therefore can be constant variables, reducing the required reads on storage.

Consider updating these state variables to constant variables.

Update: *Resolved in PR #841 and commit cb1bb14.*

L-14 Naming consistency

The naming for actions on the rewards pool is confusing and not consistent with traditional naming. For example:

- `stake` and `withdraw` should be either `stake` and `unstake` or `deposit` and `withdraw`.
- `getRewards` should be `claimRewards` as the `get..` prefix usually indicates read only getters.
- `exit` should be `withdrawAndClaimRewards` or `unstakeAndClaimRewards`.
- `restartRewards` should be `accumulateAndRestartRewards` as this is what the function does.
- `earned` should be `rewardsAvailable`.
- `getRewardsAvailable` should be `totalRewardsAvailable`.

For clarity, consider renaming the above mentioned functions to better describe their functionality.

Update: *Acknowledged, not resolved. The Sandbox team stated:*

> *As this will have impact in other pieces of the solution (Frontend and Backend) we will keep the naming as it is.*

Notes & Additional Information

N-01 Duplicate code

The functions `getERC1155MaxStake` and `getERC721MaxStake` are exactly the same code as `checkAndGetERC1155Stake` and `checkAndGetERC721Stake` respectively without the `require` statements.

For consistency, clarity, and conciseness, consider creating an internal helper function to remove the duplicated code.

Update: *Acknowledged, not resolved. The Sandbox team stated:*

> *As the usage of the functions are different and changing them will have huge impact on the contract functionality, we will acknowledge this issue for now.*

N-02 Unused named return variables

Named return variables are a way to declare variables that are meant to be used inside a function body and returned at the end of the function. This is an alternative to the explicit `return` statement to provide function outputs.

Throughout the codebase there are the following instances of unused named return variables:

- In the return statement on line 19 of the `ERC2771HandlerV2.sol` contract.
- In the return statement on line 441 of the `ERC20RewardPool.sol` contract.

Consider either using or removing any unused named return variables.

Update: *Resolved in PR #842 and commit 3647686.*

N-03 Non-explicit imports are used

Non-explicit imports are used inside the codebase, which reduces code readability and could lead to conflicts between names defined locally and the ones imported. This is especially

important if many contracts are defined within the same Solidity files or the inheritance chains are long.

Within `StakeTokenWrapper.sol`, global imports are being used. For instance:

- line 5 of `StakeTokenWrapper.sol`
- line 6 of `StakeTokenWrapper.sol`

Following the principle that clearer code is better code, consider using named import syntax (`import {A, B, C} from "X"`) to explicitly declare which contracts are being imported.

Update: *Resolved in PR #842 and commit 43de7c3.*

N-04 Internal constant variable can be private

The constant variable DECIMALS_18 is only used in the file where it is declared.

Consider changing the visibility of this variable to private.

Update: *Resolved in PR #843 and commit 7d07ba7.*

N-05 Constants not using `UPPER_CASE` format

Throughout the codebase there are constants not using `UPPER_CASE` format. For instance:

- The `idsLimit` constant declared on line 18 in `ContributionRules.sol`
- The `contractsLimit` constant declared on line 19 in `ContributionRules.sol`
- The `maxMultiplier` constant declared on line 20 in `ContributionRules.sol`
- The `timeLockLimit` constant declared on line 12 in `LockRules.sol`
- The `amountLockLimit` constant declared on line 13 in `LockRules.sol`

To increase readability and adhere to the Solidity Style Guide, consider writing all constant variable names in the `UPPER_CASE` format.

Update: *Resolved in PR #844 and commit f24f100.*

N-06 Use of outdated Solidity version

Throughout the codebase there are `pragma` statements that use an outdated version of Solidity. For instance:

- The `pragma` statement on line 3 of `ERC2771HandlerV2.sol`
- The `pragma` statement on line 2 of `SafeMathWithRequire.sol`
- The `pragma` statement on line 3 of `ERC20RewardPool.sol`
- The `pragma` statement on line 3 of `StakeTokenWrapper.sol`
- The `pragma` statement on line 3 of `IContributionCalculator.sol`
- The `pragma` statement on line 3 of `IContributionRules.sol`
- The `pragma` statement on line 3 of `IRewardCalculator.sol`
- The `pragma` statement on line 3 of `TwoPeriodsRewardCalculator.sol`
- The `pragma` statement on line 3 of `ContributionRules.sol`
- The `pragma` statement on line 3 of `LockRules.sol`
- The `pragma` statement on line 3 of `RequirementsRules.sol`

Consider taking advantage of the latest Solidity version to improve the overall readability and security of the codebase.

Update: *Acknowledged, not resolved. The Sandbox team stated:*

> *As we are still on the 0.8.x version of solidity and all the other contracts of The Sandbox ecosystem use the same version, we will keep this version for now.*

N-07 TODO comments in the codebase

The following instances of TODO comments were found in the codebase. These comments should be tracked in the project's issue backlog and resolved before the system is deployed:

- Lines 163 and 164 of `TwoPeriodsRewardCalculator.sol`

During development, having well described TODO comments will make the process of tracking and solving them easier. Without that information these comments might age and important information for the security of the system might be forgotten by the time it is released to production.

Consider tracking all instances of TODO comments in the issues backlog and linking each inline TODO to the corresponding backlog entry. Resolve all TODOs before deploying to a production environment.

Update: *Acknowledged, not resolved. The Sandbox team stated:*

> *There are only 2 TODOS in the code that are still relevant. So, we will keep it as it is.*

N-08 Typographical errors

Throughout the codebase there were a few typographical errors. For instance:

- line 75 of the `ContributionRules` contract: "check if the calculated multipliers *exceed* the limit".
- line 115 of the `ContributionRules` contract: "Limiting the size of the array *(iterations)* to avoid the risk of DoS.".
- line 144 of the `ContributionRules` contract: "Limiting the size of the array *(iterations)* to avoid the risk of DoS.".
- line 136 of the `RequirementsRules` contract: "Limiting the size of the array *(iterations)* to avoid the risk of DoS.".
- line 170 of the `RequirementsRules` contract: "Limiting the size of the array *(iterations)* to avoid the risk of DoS.".

Consider correcting the above and any other typos in favor of correctness and readability.

Update: *Resolved in PR #845 and commit 4dc192b.*

N-09 Gas optimizations

Possible gas cost improvements were found throughout the codebase. In particular:

- Within a given function, the `_msgSender()` value can be stored on the stack to reduce the number of calls to and extraction of the sender; see `stake`, `withdraw`, `exit`, `getReward` in `ERC20RewardPool` and `_stake` and `_withdraw` in `StakeTokenWrapper.sol`.
- Within `StakeTokenWrapper`, balances can be updated using the shorthand `+=`.
- `_updateContribution` can reduce gas by not accessing `_totalContributions` more than once by subtracting and adding the old and new contributions respectively and applying the shorthand `+=`.
- `_maxStakeAllowedCalculator`:
  - The nested if block can check `maxAllowed` instead of reading from storage.
  - calculate `maxStakeERC721 + maxStakeERC1155` once as a `totalMaxStake`.

- Within `ContributionRules.sol` and `RequirementRules.sol`, there are for loops that read the length of a state variable as part of the condition on each iteration. It is cheaper to save the length to the stack than to access the state variable on each iteration.
- Within `ERC20RewardPool.sol`, `ContributionRules.sol`, and `RequirementRules.sol`, there are for loops that increment the loop iteration counter outside of an `unchecked` block. Since solidity version 0.8.0, arithmetic operations automatically check for over- and underflows which costs gas. Since it is highly unlikely that a positively incrementing `uint256` variable starting from 0 will overflow within a loop, the increment could be wrapped in an `unchecked` block.
- Within `ERC20RewardPool.sol`, `ContributionRules.sol`, and `RequirementRules.sol`, there are variables that are initialized to their default values.

To reduce the gas consumption during the execution of the code, consider updating the code to be more performant.

Update: *Resolved in PR #846 and commit 6c2e606.*

Conclusions

No critical or high severity issues were found. Three (3) medium issues were found. Some changes were proposed to follow best practices and reduce the potential attack surface.

Appendix

Monitoring recommendations

While audits help in identifying code-level issues in the current implementation and potentially the code deployed in production, we encourage The Sandbox team to consider incorporating monitoring activities in the production environment. Ongoing monitoring of deployed contracts helps identify potential threats and issues affecting production environments. Hence, with the goal of providing a complete security assessment we want to raise several actions addressing trust assumptions and out-of-scope components that can benefit from on-chain monitoring.

Consider monitoring:

- Ownership changes via `OwnershipTransferred` event, changes to roles via `RoleAdminChanged`, `RoleGranted` and `RoleRevoked` events and changes in the contract's state via `Paused` or `Unpaused` events. Any unexpected modification could signal a privileged account being compromised.
- Changes to critical system settings, via `TimelockClaimSet`, `TimelockDepositSet`, `TimeLockWithdrawSet`, and `AmountLockClaimSet` events for lock rules settings and `MaxStakeOverallSet`, `ERC11551RequirementListDeleted`, and `ERC721RequirementListDeleted` for changes in staking requirement rules.
- Changes to contribution rules via `ERC1155MultiplierListSet`, `ERC721MultiplierListSet`, `ERC1155MultiplierListDeleted`, `ERC721MultiplierListDeleted`, `ERC721MultiplierLimitSet`, and `ERC1155MultiplierLimitSet` events as they impact the amount of rewards that users may get.
- Pool liquidity levels, especially when the reward token is the same as the stake token. Insufficient liquidity can have negative effects for users such as rewards not being paid.
- Changes to campaigns via `InitialCampaign`, `NextCampaign` and `UpdateCampaign` events.
- Total unclaimed rewards; this value is important to ensure the pool's `rewardToken` balance is enough to cover unclaimed rewards.
- Transactions from the admin or owner addresses; unintended or unscheduled transactions can signal a potentially compromised account.