INFINITY

Axie Infinity Staking Contracts Audit

OpenZeppelin | security

The Axie Infinity team asked us to review and audit their Axie Infinity staking contracts. We looked at the code and now publish our results.

Scope

The Axie Infinity universe is large, but the scope of this audit includes only the contracts that constitute the new staking feature that is yet to be released. We audited commit `20945771e93db096cd034f3e3f4373572d518426` of the axieinfinity/dapss-smart-contracts repository. The scope includes the following files:

- `contracts/staking/manager/ERC20StakingManager.sol`
- `contracts/staking/manager/ERC20StakingManagerProxy.sol`
- `contracts/staking/manager/ERC20StakingManagerStorage.sol`
- `contracts/staking/manager/IERC20StakingManager.sol`
- `contracts/staking/pool/ERC20StakingPool.sol`
- `contracts/staking/pool/ERC20StakingPoolProxy.sol`
- `contracts/staking/pool/ERC20StakingPoolStorage.sol`
- `contracts/staking/pool/IERC20StakingPool.sol`

Note: the repository in question is private at the time of writing, so many hyperlinks will only work for the Axie Infinity team.

In addition, imported code was reviewed to the extent used. All other project files and directories (including tests), along with external dependencies and projects, were excluded from the scope of this audit. External code and contract dependencies were assumed to work as expected.

System overview

Axie Infinity is a competitive NFT-based game played in a Pokemon-inspired universe on the Ethereum blockchain. Players can battle, collect, raise, and build a land-based kingdom for their Axie pets. There is a marketplace where players can deal their Axies, land, or other items, like potions used to breed new Axies.

Axie Infinity is still in early access, but it ranked the #1 Ethereum game based on daily active users. Axie Infinity allows players to "play to earn" using the following methods:

- Competing in PVP battles
- Breed Axies and selling them in the market
- Speculating on rare Axies
- Beginning soon, Axie Infinity will offer a governance token, AXS, that players will be able to earn by staking assets

As already mentioned, only the staking contracts were reviewed as part of this audit. There are two primary staking implementation contracts - one *staking manager* contract, and one *staking pool* contract. A separate staking pool contract must be deployed for each staking token, and the staking manager has administrative controls over all the staking pools.

Users stake a specific ERC20 token, which is assumed to be a trusted token, such as the LP tokens provided by Axie Infinity's DEX pools, into a whitelisted staking pool to accrue AXS rewards based on the `rewardPerBlock` and `fromBlock` settings specified by the staking manager. A `minClaimedBlocks` variable is used to determine how frequently a user can claim rewards and optionally *restake* those rewards in the case where the staking and reward token are the same. Users can unstake a portion, or all, of their staking token, claiming their rewards in the process. Alternatively, a user can use the `emergencyUnstake` function, forfeiting accrued rewards to withdraw all their staked tokens immediately.

Privileged roles

There is an admin role that is used for access control through the `onlyAdmin` modifier to access the following functions:

- `setStakingToken` to define the staking token for a pool contract

- `setStakingManager` to change the staking manager address used to manage a pool
- `setMinClaimedBlocks` to set how frequent a user may collect rewards
- `setFutureBlockRewards` to change the start/end of reward campaigns, by changing the reward per block starting from a specific block
- `whitelistPools` to approve a new pool contract to obtain rewards

Note, because of the proxy pattern, the admin for the proxy contract who can perform upgrades is also the admin for the implementation contracts.

Security model and trust assumptions

The staking contracts are ownable and upgradeable. The admin account can be used to change critical factors, such as changing the manager contract used to control a pool with staked funds. We do not have the details of the admin address, for example, whether it is an EOA, a multi-sig contract, or governance system. Due to the power of the admin account, there is total trust placed in Axie Infinity to act in the best interest of the game and users funds.

In addition, the codebase uses exclusively custom and private libraries for imported code, such as `Math`, `ProxyStorage`, and `IERC20` libraries. The imported libraries have not been reviewed as a part of this audit and are assumed to be trustworthy and work as intended going forward.

Update: *Some of the issues listed below have been either fixed, partially fixed or acknowledged by the Axie Infinity team. We address below the fixes introduced in individual pull requests. Our analysis of the mitigations assumes any pending pull request will be merged, but disregards all other unrelated changes to the codebase.*

Here we present our findings, along with the updates for each one of them.

Critical severity

None.

High severity

[H01] Incorrect total staking amount calculation

The `syncPool` function in the `ERC20StakingManager` contract synchronizes the state of a given whitelisted pool before a user stakes, unstakes, restakes, or claims pending rewards from it. Synchronizing a pool consists of two main operations:

- Updating the block that the pool was last synchronized, by updating the `lastSyncedBlock` variable of the `Pool` struct.
- Updating the accumulated rewards per share, stored in the `accumulatedPerShare` variable of the `Pool` struct. This variable keeps track of the rewards per share accumulated over a period of time, measured in blocks, and it is calculated by summing the total rewards of several periods of time, and dividing by the total staking amount in the pool in each of these periods.

However, the total staking amount is calculated using the `balanceOf` function of the staking token to check the total pool's balance, which can be modified by anyone by performing an ERC20 transfer operation to the pool contract. This can lead to the following misbehaviors:

- If the `getStakingTotal` function does not return 0 because someone transfers one or more tokens to the pool after it is whitelisted, the `syncPool` function will not terminate execution early, and rewards will be included in the calculation of the `accumulatePerShare` variable. Since these rewards per share are also going to be included in the `creditedRewards` of future stakers, they will not be distributed, and therefore will get stuck.
- The `_total` value will be greater than the amount staked by users, and therefore the `accumulatedPerShare` value for every block period will be lower than expected, which means that fewer reward tokens will be distributed to stakers.
- The number of reward tokens held by the `ERC20StakingManager` contract will be greater than the rewards distributed to stakers, and since there is no mechanism to withdraw reward tokens from the manager, this difference in amount will get locked in the contract.

Consider tracking the total staking amount in stake and unstake operations in a state variable, and retrieving this value in the `getTotalStaking` function instead of using the `balanceOf` function.

Update: *Fixed in PR#38. The Axie Infinity team added a `_stakingTotal` internal variable to track the total staking amount on stake and ustake operations.*

Medium severity

[M01] Reward tokens softlocked after `emergencyUnstake`

The `emergencyUnstake` function from the `ERC20StakingPool` contract lets users withdraw their stake, even if the system if paused. This function will reset the `creditedRewards` and `debitedRewards` variables from the `rewardInfo` variable of the `ERC20StakingManager` contract for a particular pool and user.

Given that the rewards per share are accumulated by blocks in the `accumulatedPerShare` variable on each pool sync, and given that this variable is inversely proportional to the total amount staked, once someone unstakes their tokens through the `emergencyUnstake` function, the reward tokens that correspond to this user will get locked.

For example, with an interval of 1000 blocks, 200 rewards/block being released, and only two users participating in the pool (for the sake of simplicity):

- Alice deposits 500 tokens in block = 0
- Bob deposits 500 tokens in block = 0

After 1000 blocks, Alice decides to `emergencyUnstake` her tokens. Since the pool is synced before every `stake`/`unstake` operation, the `accumulatedPerShare`, in this case, will be:

`rewards = (toBlock - fromBlock) * rewardPerBlock = (1000 - 0) * 200 = 200,000` and:
`accumulatedPerShare = 200,000 / 1,000 = 200`

A total of 200,000 rewards are to be claimed, but 100,000 (500 * 200) will not be claimable because Alice removed her tokens through the `emergencyUnstake` function. Bob then claims his 100,000 reward tokens, leaving the pool with 100,000 tokens softlocked.

This is a softlock, because in the next `BlockReward` period, the admin can just transfer less reward tokens to the manager, so that the balance of reward tokens in the manager is the sum of the new amount and the softlocked amount.

Consider adding a way of withdrawing excess reward tokens in the staking manager that remain because of `emergencyUnstake` calls.

Update: Mitigated. The `ERC20StakingManager` contract inherits from the `Withdrawable` contract. This contract was not part of the original audit, but was taken into consideration in the review of the fixes provided by the Axie Infinity team to validate that the locked tokens are withdrawable by the administrator of the staking contracts._

[M02] Lack of event emission after sensitive actions

The following functions do not emit relevant events after executing sensitive actions.

- The `setStakingToken` function of the `ERC20StakingPoolStorage` contract, after changing the value of the `stakingToken` state variable.
- The `setStakingManager` function of the `ERC20StakingPoolStorage` contract, after changing the value of the `stakingManager` state variable.
- The `setMinClaimedBlocks` function of the `ERC20StakingManagerStorage` contract, after changing the value of the `minClaimedBlocks` state variable.
- The `syncPool` function of the `ERC20StakingManagerStorage` contract, after updating the pool's `accumulatedPershare` and/or `lastSyncedBlock` attributes.
- The `whitelistPools` function of the `ERC20StakingManagerStorage` contract, after whitelisting a pool.
- The `allocateRewards` function of the `ERC20StakingManagerStorage` contract, after updating the user's `rewardInfo`.

Consider emitting events after sensitive changes take place (including the first event emission in the constructor when appropriate), to facilitate tracking and notify off-chain clients following the contracts' activity

Update: *Fixed in PR#39. We additionally suggest to assess whether the new events introduced may benefit from adding indexes to some of their variables*.

[M03] Fragile upgradeability implementation

The Axie Infinity staking contracts implement an upgradeability pattern where both the proxy and the implementation contracts have mirrored storage layouts. The proxy implements functions to perform upgrades between different implementation versions, delegates calls to the implementation, and holds the storage of it, whereas the implementation only defines the logic of the contract, leaving its storage unused. During our review of the code, we identified the following issues:

- There are no checks in the `deployment.ts` script to identify whether the proxy and the implementation have the same storage layout before the first deployment is done. Both the proxies and the implementations follow a non-trivial inheritance chain, which could lead to different c3 linearization at compilation time. For instance, if the `Pausable` and `ProxyStorage` contracts are swapped in the `ERC20StakingPoolStorage` contract definition, its storage layout and the `ERC20StakingPoolProxy` storage layout will be different and the storage may become corrupted. It has to be taken into consideration that storage layout can become corrupted *between* upgrades if the corresponding checks are not performed.

- There are no checks in the `deployment.ts` script of function selectors for collisions between the proxy and the implementation. Clashing happens among functions with either equal or different names. Every function that is part of a contract's public ABI is identified, at the bytecode level, by a 4-byte identifier. The identifier depends on both the arity and name of the function, but since it is a 4 bytes truncation, there is a possibility that two different functions with different names may have the same identifier, which will make the function on the implementation unreachable.

Consider using the OpenZeppelin Contracts library, which provides a set of contracts to manage several different upgradeability patterns, and consider using the OpenZeppelin Upgrades library, which provides functionality to check storage layout changes.

Alternatively, consider implementing a script to check that both the storage layout of the the proxy and the implementation are the same before the deployment, and between upgrade operations. Additionally, consider implementing a script to check that there are no function selector collisions between the proxy and the implementation, or consider implementing a similar pattern to the transparent proxy.

Update: *Acknowledged. The Axie Infinity team will carry on with the current state for now.*

[M04] Missing documentation and docstrings

The README.md of the project has no information about what is the purpose of the project and how to use it. README files on the root of git repositories are the first documents that most developers often read, so they should be complete, clear, concise, and accurate.

Moreover, most of the functions throughout the codebase lack Nat Spec comments for parameters and return values.

This hinders reviewers' understanding of the code's intention, which is fundamental to correctly assess not only security, but also correctness. Additionally, docstrings improve readability and ease maintenance. They should explicitly explain the purpose or intention of the functions, the scenarios under which they can fail, the roles allowed to call them, the values returned and the events emitted.

Consider thoroughly documenting all functions (and their parameters) that are part of the contracts' public API. Functions implementing sensitive functionality, even if not public, should be clearly documented as well. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec). Consider increasing the amount of documentation created and shared with the game developers, reviewers, and gaming community.

Additionally, consider following Standard Readme to define the structure and contents for the README file, including an explanation of the core concepts of the repository, the usage workflows, the public APIs, instructions to test and deploy it, and how the code relates to other key components of the project.

Update: *Acknowledged. The Axie Infinity team will carry on with the current state for now.*

Low severity

[L01] Unnecessary pool synchronization

The `syncPool` function from the `ERC20StakingManager` contract is unnecessarily being called in:

- The `syncRewardInfo` function. This function is being called from the `stake`, `unstake` and `unstakeAll` functions of the `ERC20StakingPool` contract through the `_stake` and `_unstake` functions, which implement the `poolSynced` modifier and therefore trigger the pool synchronization beforehand.

- The `allocateRewards` function. This function is being called from the `restakeRewards` and `claimPendingRewards` functions of the `ERC20StakinPool` contract, which also implement the `poolSynced` modifier and therefore trigger the pool synchronization beforehand.

Even though this second call will finalize its execution early, it unnecessarily consumes more gas and hinders reviewers' understanding of the pool synchronization mechanism.

Consider removing any extra unnecessary function calls.

Update: *Not fixed. The Axie Infinity team says:*

> In `syncRewardInfo` and `allocateRewards` methods, we are not sure that other implementations of `IERC20StakingPool` that call those methods have synced the pool or not so that we will keep syncing those.

[L02] Confusing usage of the `token` and `rewardToken` attributes from the `Pool` struct

The `Pool` struct of the `IERC20StakingManager` interface defines the `token` and `rewardToken` fields to store the addresses of the staking token and reward token of a particular pool, respectively.

Some confusing behaviors that the Axie Infinity team may want to review are:

- If the staking token address is updated by calling the `setStakingToken` function from the `ERC20StakingPoolStorage` contract, this change will not be reflected in the `token` field from the manager contract, since there is no mechanism to update it, and therefore these two variables will get out of sync.

- There is no mechanism to update the reward token address, as with the staking token.

- Even though it is accessible within the `ERC20StakingManager` contract, the reward token address is not being used in the `allocateRewards` function, and an external call is being performed instead.

- The staking token address is used to check whether a pool is whitelisted in the staking manager contract, but there is no mechanism to remove it from the whitelist if needed, or if the `stakingToken` address from the `ERC20StakingPoolStorage` contract is updated to the zero address. This means that users may be able to interact with certain functions such as `getPendingRewards`, which may revert with a confusing error message triggered by the `getStakingTotal` function, instead of failing because the pool is not whitelisted anymore.

Consider reviewing these behaviors and modifying them appropriately to avoid hindering the experience of users, auditors and, external contributors.

Update: *Fixed in PR#39. The Axie Infinity team removed the `setStakingToken` function from the `ERC20StakingPoolStorage` contract, and mitigated the misbehaviors mentioned above.*

[L03] Assuming that all staking and reward tokens have 18 decimals

Throughout the `ERC20StakingManager` contract, calculations are performed assuming that the staking token and the reward token will always have 18 decimals of precision, for instance:

- When calculating the user's `creditedRewards` in the `syncRewardInfo` function and in the `allocateRewards` function

- When calculating the rewards per share in the `syncPool` function and in the `_getPendingRewards` function

- When calculating the pending rewards value to be returned by the `_getPendingRewards` function

Consider performing these calculations using the decimal precision of the corresponding token. Alternatively, if these tokens are *always* going to have 18 decimals of precision, consider defining a constant variable for this and every other constant value used, giving it a clear and self-explanatory name. Additionally, for complex values, inline comments explaining how they were calculated or why they were chosen are highly recommended. Following Solidity's style guide, constants should be named in `UPPER_CASE_WITH_UNDERSCORES` format, and specific public getters should be defined to read each one of them.

Update: *Acknowledged. The Axie Infinity team says that they will only use tokens with 18 decimals.*

[L04] Naming issues

To favor explicitness and readability, several parts of the contracts may benefit from better naming.

Our suggestions are:

In `ERC20StakingManger.sol`:

- `getTotalRewards` to `getIntervalRewards`, or `getBlockRewards`
- `_block` to `_initialBlock` or `_intervalInitialBlock`
- `_total` to `_stakingTotal` or `_totalStakingAmount`

In `ERC20StakingManagerStorage.sol`:

- `rewardInfo` to `userRewardInfo`

In `IERC20StakingManager.sol`:

- `token` to `stakingToken`
- `accumulatedPerShare` to `accumulatedRewardsPerShare`

Update: *Fixed in PR#40*.

[L05] Outdated Solidity version in use

An outdated Solidity version, `0.5.17`, is currently in use. As Solidity is now under a fast release cycle, make sure that the latest version of the compiler is being used at the time of deployment (presently `0.8.7`).

Update: *Not fixed. The Axie Infinity team says:*

> Our toolchains were built around 0.5.x. We would like to stick with this version until we can update our toolchains.

[L06] Multiple getters for the same state variable

Some contracts in the codebase contain multiple public getter functions for the same state variable.

For example:

- The `getStakingToken` function from the `ERC20StakingPool` contract. Since the `stakingToken` variable is public, Solidity will automatically generate a public `stakingToken` function.
- The `getStakingAmount` function from the `ERC20StakingPool` contract. Since the `stakingAmount` variable is public, Solidity will automatically generate a public `stakingAmount` function.

Moreover, the `stakingToken` variable is being accessed within the `ERC20StakingPool` contract through a getter function in several places, for instance, in the `_stake`, `_unstake`, and `emergencyUnstake` functions, but this is not necessary since this variable is defined as a storage variable in the `ERC20StakingPoolStorage` parent contract and therefore is accessible within `ERC20StakingPool`.

To favor encapsulation and explicitness, ensure that there is at most one publicly exposed getter for each contract state variable.

Update: *Fixed in PR#41*.

Notes & Additional Information

[N01] Incorrect function visibility

The following public functions are not being called internally in their respective contracts:

- The `getPoolInfo`, `getPendingRewards`, and `getBlockReward` functions in the `ERC20StakingManger` contract.

- The `getPendingRewards`, and `getStakingTotal` in the `ERC20StakingPool` contract.

Consider declaring these functions as external instead of public.

Update: *Fixed in PR#41*.

[N02] Misleading comments

There are some misleading comments throughout the codebase. For example:

- There are incorrect parameter names in comment for `RewardClaimed` event.

- In the `IERC20StakingManager` interface is repeatedly stated: "Requirement: the method caller is whitelisted {IERC20StakingPool} contract." In fact, the caller can be any address, but the `_pool` parameter has to be whitelisted.

- The *requirement* comment for the `syncPool` function is not strict enough. It states, "The `_pool` address is whitelisted." In fact, the pool address must be whitelisted, and the function caller must be a whitelisted `_pool`. This is the opposite error of the previous bullet point. The comments should be swapped to improve clarity.

- The requirement for the `setFutureBlockReward` function states "The `_fromBlock` should be equal to or larger than current block.", but the implementation requires `_fromBlock` to be strictly greater than `block.number`.

- The `@dev` tag in the `getPoolInfo` function references the `getPendingRewards` function of the `IERC20StakingManager` interface instead of the `getPoolInfo` function.

Consider reviewing all the comments in the codebase to correct errors and improve clarity.

Update: *Partially fixed in PR#42. The "requirement" comments in the `IERC20StakingManager` contract are still misleading.*

[N03] Not using `interface` keyword

The `IERC20StakingManager` and `IERC20StakingPool` contracts *look* like interfaces, but they do not use the `interface` keyword, instead defined as regular contracts. Consider changing the contracts to interfaces to improve semantics and better restrict the potential functionality.

Update: *Fixed in PR#42.*

[N04] Repeated code

The `restakeRewards` and `claimPendingRewards` functions are almost equal (they both allocate the user's rewards), with the difference that the `restakeRewards` function re-stakes the rewards.

To favor reusability, consider refactoring all repeated operations throughout the codebase into private or internal functions.

Update: *Fixed in PR#42.*

[N05] Typographical errors

We identified the following typographical errors:

- "or" should be "of"
- "distibute" should be "distribute"

Consider correcting these errors and using a linter or IDE plugin to automatically check for typographical errors.

Update: *Fixed in PR#42.*

[N06] Unnecessary imports

We've identified the following unnecessary imports:

- `IERC20` in `ERC20StakingManagerStorage.sol`.
- `IERC20StakingPool` in `ERC20StakingPoolStorage.sol` (already imported and used in `ERC20StakingPool.sol`).

Consider removing unnecessary imports to improve code clarity and reduce maintenance.

Update: *Fixed in PR#42*.

Conclusions

No criticals and 1 high severity issues were found. Some changes were proposed to follow best practices and reduce the potential attack surface.